

Modeling and Verification of an Interrupt System in $\mu\text{C}/\text{OS-III}$ with TMSVL

Jin Cui, Zhenhua Duan^(✉), Cong Tian, and Nan Zhang

ICTT and ISN Laboratory, Xidian University,
Xi'an 710071, People's Republic of China
zhenhua_duan@126.com

Abstract. Interrupt mechanism is a useful means to ensure timely response to asynchronous events in real-time systems. Modeling and verification of the correctness of interrupt systems are important in practice. This paper proposes an efficient way to formalize the interrupt mechanism in TMSVL. We apply TMSVL to model and verify a timer interrupt application running under $\mu\text{C}/\text{OS-III}$. To do so, the real-time system is formalized in TMSVL, and properties to be verified are specified by projection temporal logic (PTL) formulas or TMSVL statements. Then a model checker built in the toolkit MSV is employed to check whether or not the model satisfies the properties automatically.

Keywords: Real-time systems · Interrupt · Schedulability · $\mu\text{C}/\text{OS-III}$ · Model checking

1 Introduction

$\mu\text{C}/\text{OS-III}$ is a widely used open source real-time operating system (OS) [10, 13]. It is a portable, ROM-able, scalable, preemptive, and real-time deterministic multitasking kernel. It has the following features: (1) priority-based preemptive scheduling; (2) synchronization and communication between tasks, for instance, mutual exclusion semaphores with built-in priority ceiling protocol to prevent priority inversions; (3) interrupt and time management. $\mu\text{C}/\text{OS-III}$ is used in a wide variety of industries such as avionics, medical equipments, industrial controls, and so on.

A real-time system using $\mu\text{C}/\text{OS-III}$ which performs reliably and safely is vital. It is important to ensure the correctness, reliability and safety of such a system. At present, there are several approaches that can be used to improve these properties of real-time systems. For instance, (1) simulation and testing based approaches; (2) verification approaches such as theorem proving and model checking. However, simulation and testing try to find out bugs of a system based on enumeration of either simulation environments or test cases [9]. As Dijkstra pointed out “testing can only find the presence of errors never their

This research is supported by the NSFC Grant Nos. 61133001, 61322202, 61420106004, 91418201, and 61272117.

absence” [5]. Verification is a strict mathematical approach which can be used to prove whether or not a system satisfies a property by means of theorem proving or model checking. However, theorem proving needs involvement of manual efforts while model checking suffers from so called state explosion problem [11]. In addition, the model of a system and the property to be verified are defined using different notations. This makes the verification process complicated. To overcome this problem, Timed Modeling, Simulation and Verification Language (TMSVL) [7] which can be used to model, simulate and verify real-time systems is introduced.

TMSVL [8] is a timed version of MSVL [7]. A unified model checking approach has been carried out for TMSVL, that is, a system is modeled as M and the desired property is specified as ϕ in the same formalism, thus whether $\models M \rightarrow \phi$ can be checked effectively. A supporting toolkit MSV is developed to model a system in terms of TMSVL programs, to simulate a system by executing of a path of the model, and to verify properties of the system by means of the unified model checking. In this paper, we utilize TMSVL to formalize the timer interrupt application running under $\mu\text{C}/\text{OS-III}$. First, we formalize a general interrupt mechanism in TMSVL with a derived structure. Then this structure is applied to formalize a timer interrupt system. At last, the correctness and timeliness properties are specified in TMSVL and verified with the unified model checker MSV. 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 automatically be performed using MSV.

There are several techniques which are used to verify systems developed under similar OS. In [4], a method is presented for converting the Trampoline kernel code into formal models for the model checker SPIN and a series of experiments using an incremental verification approach has been used to improve the performance of the verification. While the formal language PROMELA as an input of SPIN does not aim at modeling real-time systems. Thus time-dependent properties may not be verified in SPIN. Timed automata [1] are extended from Büchi automata by introducing the real-valued clock variables. It is a useful formalism to model real-time systems. In [15], a distributed fault-tolerant real-time application is modeled with timed automata. In [14], timed automata are used to model a real-time operating system compliant with an OSEK/VDX standard. Timed automata are also used to model primitives of Ravenscar run-time kernel for Ada [12]. However, the variable used to measure the execution time of tasks is an integer. This violates the characteristic of continuous time. A well-known tool for model checking timed automata is UPPAAL [2]. While systems and properties are not described in the same formalism. Since the desired property is usually described in CTL, LTL [11] or TCTL [3].

The paper is organized as follows. The next section introduces TMSVL language and a formalization of the interrupt mechanism. Section 3 gives an overview on $\mu\text{C}/\text{OS-III}$. In Sect. 4, we show how a timer interrupt application running under $\mu\text{C}/\text{OS-III}$ can be modeled and verified in TMSVL. 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 [6]. It is an executable subset of PTL (Projection Temporal Logic) [7], for the MSVL statements are defined by basic PTL formulas. A toolkit named MSV has been developed to preform simulation, modeling and verification automatically.

However, MSVL is inefficient to describe time constraints since it abstracts away from time, retaining only the sequence of states. For instance, a process p starting at time point t_1 and ending at t_2 cannot be expressed in MSVL with reals. It can be expressed by $len()$ statement in MSVL if the time increments of each two successive states are identical. However, this makes the time increment be the least one of all the successive states, which unnecessarily increases the number of states for the model and thus reduces the efficiency for model checking. To avoid this defect, we extend MSVL by making time explicit and introduce a time constraint statement in the form of $(t_1, t_2)p$ to describe the situation where a process p starts at time point t_1 and ends at t_2 . The extended MSVL is named TMSVL [8] with variables T and Ts being used to describe time and time increment, respectively. Meanwhile, we have extended the toolkit MSV with TMSVL thus verification of real-time systems can automatically be performed using MSV.

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:

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

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

Elementary statements of TMSVL are defined as follows:

- | | |
|-----------------------------|---|
| 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 | $(tp_1, \dots, tp_m) \text{ prj } (tp)$ |

MSVL statements are included first. 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 . $tp_1 \wedge tp_2$ means that tp_1 and tp_2

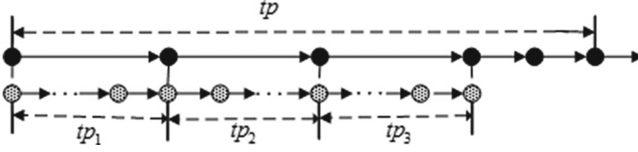


Fig. 1. An example of projection structure

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 language like *C* and *Java*.

Projection statement $(tp_1, \dots, tp_m) \text{ prj } tp$ means tp is executed in parallel with $tp_1; tp_2; \dots; tp_m$ over an interval obtained by taking endpoints of the intervals over which tp_1, \dots, tp_m are executed. An endpoint denotes the first or the last state of an interval. Taken $(tp_1, tp_2, tp_3) \text{ prj } tp$ as an example. We assume tp_3 terminates before tp . The construct of $(tp_1, tp_2, tp_3) \text{ prj } tp$ is depicted in Fig. 1.

If tp_1, \dots, tp_m are identical, we usually use $((tp_1)^m) \text{ prj } (tp)$ to represent $(tp_1, \dots, tp_m) \text{ prj } (tp)$ for simplicity. $((tp_1)^{\otimes}) \text{ prj } (tp)$ means m can be any non-negative integers, and \otimes is named projection-star.

2.2 Interrupt in TMSVL

In real-time embedded applications, interrupt-driven systems are widely adopted due to strict timing requirements. Interrupt mechanism is an effect way to handle events like a request to an external device or timed detection and control. When an interrupt is triggered, the processor stops executing the current task and switches to handle the specified program, namely interrupt service routine (ISR) or interrupt handler. The process including responding to interrupt and recovering from interrupt can be modeled by a derived statement $q \text{ when } b \text{ do } p$, which is defined as follows.

$$q \text{ when } b \text{ do } p \stackrel{\text{def}}{=} ((\text{if } b \text{ then } p \text{ else skip})^{\otimes}, r \wedge \varepsilon) \text{ prj}(q; r \wedge \varepsilon) \wedge \text{halt}(r)$$

Here q represents the process which may be interrupted, p is the interrupt handler and b indicates that the interrupt is triggered and can be processed. The definition of interrupt is a conjunction of *projection* and *halt* statements. The *projection* statement describes the relation between the main process q and the interrupt handler p . When the interrupt is triggered, b becomes true, p is performed, and only when p is finished, q is resumed. Otherwise, q is executed. *skip* is a MSVL statement which means the length of an interval is one. An interval is a non-empty (possibly infinite) sequence of states. The length of an interval is the number of sates minus one. Interrupt is processed only when q does not terminate. This is realized by introducing the auxiliary proposition r

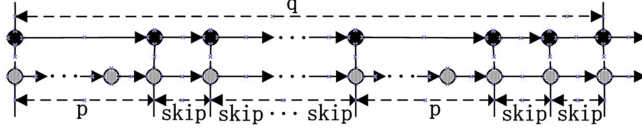


Fig. 2. The structure of interrupt in TMSVL

which neither occurs in q nor p . Whenever q is terminated, $r \wedge \varepsilon$ is attached to the end state of the execution of q . This enables statement $halt(r)$ to handle the termination since $halt(r)$ ensures when r becomes true, q is terminated. Thus, the execution of whole *projection* statement ends. An example of the interrupt structure is illustrated in Fig. 2. In addition, p can also be an interrupt structure for the case of interrupt nesting.

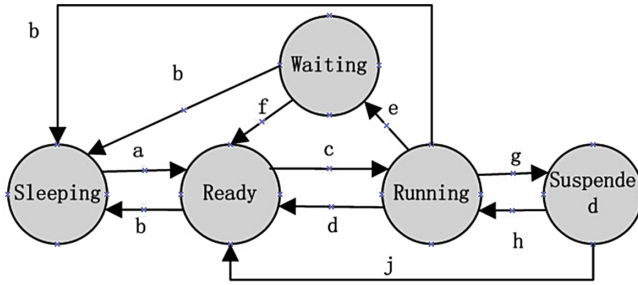
3 $\mu C/OS$ -III Overview

3.1 Tasks Management

The functionality of a real-time application is usually achieved by a number of different priority tasks coordinated by the scheduler of the operating system. In $\mu C/OS$ -III, a task has five states: sleeping, ready, running, waiting and suspended. How states of tasks change is shown in Fig. 3. Sleeping state refers to that a task resides in memory only in the form of code, and is not known by the operating system. When a task is created by a program, it is registered in the operating system and turns the sleeping state into ready state. A ready task can also be deleted by a user program, which will lead the state to change to sleeping state. If a ready task obtains the processor, it will be executed and in the running state. A running task may turn into waiting state when it waits for the occurrence of some events, or turn into ready state when it is preempted by a higher priority task, or become suspended when an interruption occurs. A task in waiting state can either become ready when the waited events occur or turn into sleeping when it is deleted. A task in suspended state turns into ready state if the interrupt handler makes a higher priority task ready, otherwise, the interrupted task is resumed.

3.2 Events Management

$\mu C/OS$ -III provides events management for task synchronization and communication. An event can be a request for common resource such as hardware device and buffer, or release of the common resource. The process that a task requests or releases common resource is managed by the operating system. Furthermore, the operating system manages these events with appropriate mechanisms such as semaphore, mutex semaphores, event flag groups, mailbox and message queues according to different characteristics of common resources.



a: a task is created; b: a task is deleted; c: a ready task gets the processor; d: a task is preempted; e: a task waits for the occurrence of some events; f: the waited events have occurred; g: a task is interrupted; h: the interrupted task is resumed j: the interrupt makes a higher priority task ready

Fig. 3. State transition graph of tasks

3.3 Interrupt Management

In $\mu\text{C}/\text{OS-III}$, interrupt has higher priority than tasks. Interrupt mechanism is adopted to deal with asynchronous events timely. When an interrupt request is received, if interrupt is not disabled, the OS suspends the current running task and switches to the corresponding interrupt handler. At the end of the interrupt handler, the OS scheduler puts the highest priority ready task into running state instead of just returning to the interrupted task.

Timer interrupt is a fundamental hardware condition for tasks synchronization. It is the heart of $\mu\text{C}/\text{OS-III}$. By setting the hardware timer, the interrupt arrives every 10~200 ms. The corresponding ISR is timer tick ISR. It does the following jobs: (1) increasing the timer by one time tick; (2) traversing the task control block and decreasing the delayed tasks by one time tick. Turning the waiting tasks with the delay being zero and waiting for no event to ready state; (3) conducting task scheduling. Interrupt is enabled until the end of a single execution of the timer tick ISR. Without timer interrupt, multi-task scheduling operating systems do not exist, neither do real-time systems.

3.4 OS Kernel Service

Kernel is an important part of OS. Its primary function is task scheduling. The task scheduling function mainly does the following work: (1) finding the highest priority ready task; (2) if the highest priority ready task is not the current running task, performing a task switching, namely, making the processor available for the highest priority ready task and saving the context of the preempted task. There are two schedulers in $\mu\text{C}/\text{OS-III}$, that is, task level and interrupt level scheduler. Interrupt level scheduling occurs at the end of interrupt processing, whereas task level scheduling is triggered in the following conditions: (1) a task is added or deleted, or the priority of a task is changed; (2) a task delays itself, or the delay time is decreased to zero; (3) the event a task requests occurs.

4 Modeling and Verification of a Timer Interrupt Application

In this section, we utilize TMSVL to verify the correctness and timeliness of an abstract timer interrupt application running under $\mu\text{C}/\text{OS-III}$.

The application consists of three tasks $Task_0$, $Task_1$, and $Task_2$. We assume the priority of each task is 5, 10, and 14. The larger value is corresponding to lower priority. Therefore, $Task_0$ has the highest priority, followed by $Task_1$ and finally $Task_2$. $Task_0$ and $Task_2$ need to access the mutex semaphore sem when they are executed. The pseudo-code of the three tasks is given in Fig. 4.

| $Task_0()$ | $Task_1()$ | $Task_2()$ |
|--|---|---|
| <pre> { sem=OSMutexCreate(pri, err); while(1) { pend(sem); Comp₀; post(sem); OSTimeDly(n₀) ; } }</pre> | <pre> { while(1) { Comp₁; OSTimeDly(n₁) ; } }</pre> | <pre> { while(1) { pend(sem); Comp₂; post(sem); OSTimeDly(n₂) ; } }</pre> |

Fig. 4. Tasks pseudo-code

$Task_0$ creates sem by invoking the system function $OSMutexCreate()$. The first parameter namely pri is the priority to use when accessing the mutex semaphore, the second parameter err indicates the error message, and the return value is the created semaphore. We should specify pri a priority that is higher than any of the tasks competing for sem to prevent the priority inversion. This is the essence of the priority ceiling protocol. Thus the value of pri is lower than 5. $Task_0$ and $Task_2$ request sem before performing $Comp_0$ or $Comp_2$ and release sem after finishing $Comp_0$ or $Comp_2$. Then the tasks invoke the system function $OSTimeDly()$ to cede the processor and turn to the waiting state. The parameter of $OSTimeDly()$ is the number of time ticks the tasks block themselves. The three tasks are loop structures. Each new execution circle of the three tasks except for the first one is preceded by a delay of n_i ($i = 0, 1, 2$) time ticks.

In order to verify the abstract application, we assume the three tasks are created at the same time ST . The correctness and timeliness properties depend both on the design of the application and hardware environment. We assume the timer interrupt is triggered every 20 ms, and the time it takes to handle the timer interrupt is 0.2 ms.

```

struct m_task{
    int rd, ex, wait, spd and
    float C, D, ac, acD, dly
};
struct m_task Task[3]

```

Fig. 5. Data type for tasks

4.1 Modeling of the Application

A new data type *m_task* is defined in Fig. 5 to store task information for the modeling and verification purpose. In *m_task*, *rd*, *ex*, *wait*, *spd* are integer variables. *rd* = 1 if a delay of a task finishes, *rd* = 0 otherwise; *ex* = 1 if a task is running, *ex* = 0 otherwise; *wait* = 1 if a task is at the state of waiting, *wait* = 0 otherwise; *spd* = 1 if a task is suspended by interrupt, *spd* = 0 otherwise. *C*, *D*, *ac*, *acD*, *dly*, *d* are float variables. *C* denotes the time it takes to perform a computation. *D* is the deadline of each computation of a task. *ac* stores the accumulated running time of a task in the current period. *acD* denotes the accumulated delay time of a task in the current period. *dly* is the time a task delays each time.

Task[3] is an array of *m_task* type, and *Task*[*i*] is corresponding to *Task_i*. Other notions and their meanings are given below (*i* is an integer and *i* = 0, 1, 2, *N* is a constant and *N* = 2):

- *d*[*i*]: a real variable, with a non-negative value denoting the remainder of the delay time if *Task_i* is delayed; otherwise, *d*[*i*] is an infinity.
- *d*[*N* + 1]: a real variable, if the interrupt has been triggered at the current state, *d*[*N* + 1] represents the time needed for finishing the interrupt handler; otherwise, it indicates the time needed for the next arrival of an interrupt.
- *d*[*N* + 2]: a non-negative real variable indicating the time required for finishing the remaining part of a running task. If there is no running task at the current state, it is an infinity.
- *inp*: a boolean variable. *inp* = 1 indicates that the current request of the interrupt is still standing while *inp* = 0 indicates that the current request of the interrupt is fulfilled.
- *runTaskNum*: an integer variable, with a non-negative value indicating the highest ready task's subscript. That is, *runTaskNum* = *i* if *Task_i* is running, *runTaskNum* = −1 otherwise.
- *ST* and *ET*: non-negative real variables indicating the start and end time of the tasks.

The TMSVL model of the above application is defined as follows:

$$M \stackrel{\text{def}}{=} \text{clock}(e_T, e_{Ts}) \wedge TsSet \wedge (\text{while}(T < ET)\{Q\} \wedge P) \text{when}(b) \text{do}(ISR)$$

Each TMSVL program is a conjunction of $clock(e_T, e_{Ts})$ and statements. $clock(e_T, e_{Ts})$ initializes T and Ts , the current 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 the TMSVL program.

The module $TsSet$ sets the time step Ts to eliminate states without events to reduce the state space for model checking. Events including the occurrence of the interrupt, the start and end of interrupt processing, the completion of a computation, and the start and finish of a task delay. At every state, the time needed for the occurrences of those events are calculated and they are stored in the array d . Ts is always set to the minimum element of d .

$(\text{while}(T < ET)\{Q\} \wedge P) \text{when}(b) \text{do}(ISR)$ is the TMSVL interrupt structure. It is used to describe the relation between the tasks scheduling and the hardware interrupt. Q denotes a task scheduling. $\text{while}(T < ET)\{Q\}$ means Q is executed repeatedly and terminated at time point ET . P denotes the tasks module, and ISR denotes the interrupt handler. The notion b is a boolean condition here, which means that the timer interrupt is triggered and the system does not disable the interrupt.

The TMSVL model of the scheduler is shown in Fig. 6. Lines 8 to 11 show the scheduling of $Task_2$ which has the lowest priority. If $Task_2$ is ready and $Task_0$ and $Task_1$ are not ready or executing, $Task_2$ can be executed and it cannot be preempted, which is realized by the *while* statement at Line 10. As long as $Task_2$ does not finish computation, namely $Task[2].ac < Task[2].C$, the scheduler will not schedule other tasks. The scheduler makes $Task_0$ be the running task by setting $runTaskNum$ to 0 if $Task_0$ is ready, which is shown in Lines 2 to 3; if $Task_0$ is not ready but $Task_1$ is ready, $Task_1$ will be scheduled. When none of the three tasks is ready, $runTaskNum$ is set to -1 .

```

Q  $\stackrel{\text{def}}{=}$ 
1. //Selecting the ready task with the highest priority
2.   if(Task[0].rd=1)
3.     then{runTaskNum=0}
4.   and
5.   if(Task[0].rd=0 and Task[1].rd=1)
6.     then{ runTaskNum=1}
7.   and
8.   if(Task[0].rd=0 and Task[1].rd=0 and Task[2].rd=1)
9.     then{runTaskNum=2 and skip;
10.        while(Task[2].ac<Task[2].C){ runTaskNum=2 and skip }
11.        }
12.   else { skip }
13.   and
14.   if( Task[0].rd=0 and Task[1].rd=0 and Task[2].rd=0)
15.     then{runTaskNum=-1 }

```

Fig. 6. TMSVL model of the scheduler

$P_i \stackrel{\text{def}}{=} \begin{array}{l} 1. \text{ while}(T < ET) \\ 2. \quad \{ \text{ await}(\text{runTaskNum}=i); \\ 3. \quad \quad \text{Task}[i].ac=0 \text{ and} \\ 4. \quad \quad \text{while}(\text{Task}[i].ac < \text{Task}[i].C) \\ 5. \quad \quad \{ \text{ if}(\text{inp}=0 \text{ and } \text{runTaskNum}=i) \\ 6. \quad \quad \quad \text{then}\{\text{Task}[i].ac:=\text{Task}[i].ac+Ts \text{ and } \text{Task}[i].ex=1 \text{ and } \text{Task}[i].wait=0\} \\ 7. \quad \quad \quad \text{else}\{ \text{Task}[i].ex=0 \text{ and skip} \} \}; \\ 8. \quad \quad \text{if}(\text{Task}[i].ac=\text{Task}[i].C) \\ 9. \quad \quad \text{then}\{(T, T+\text{Task}[i].dly)\text{keep(next Task}[i].acD=\text{Task}[i].acD+Ts \text{ and} \\ 10. \quad \quad \quad \text{Task}[i].rd=0 \text{ and } \text{Task}[i].ex=0 \text{ and } \text{Task}[i].wait=1); \\ 11. \quad \quad \quad \text{if}(i=2)\text{then}\{\text{await}(\text{runTaskNum}!=0)\}; \\ 12. \quad \quad \quad \text{Task}[i].acD=0 \text{ and } \text{Task}[i].rd=1 \text{ and empty} \} \\ 13. \} \end{array}$

Fig. 7. TMSVL model of each task

The task module is a parallel of tasks. P can be expressed with the parallel statement below:

$$P \stackrel{\text{def}}{=} \parallel_{i=1}^N P_i$$

where P_i is the TMSVL model of $Task_i$ and it is defined in Fig. 7. The structure of P_i is a *while* loop. In the loop body, the task is waiting for its opportunity to run which is shown in Line 2. If $Task_i$ gets the opportunity to run, namely $runTaskNum = i$, the *await* statement terminates and the statement following starts work, wherein $Task[i].ac$ is set to zero at Line 3. Otherwise, the *await* statement will not terminate and the statement after it cannot work. The value of $runTaskNum$ is determined in the scheduler Q . If $Task_i$ runs, the accumulated execution time of $Task_i$ at the next state is the sum of $Task[i].ac$ and Ts at the current state. When $Task_i$ starts a new circle, $Task[i].ac$ is set to zero. If $Task_i$ finishes an execution of a circle, that is, $Task[i].ac = Task[i].C$, it will delay $Task[i].dly$ time units, during which the task is at the waiting state. For $Task_0$ and $Task_1$, they will be ready again and repeat the above steps after delaying $Task[0].dly$ and $Task[1].dly$ time units. While for $Task_2$, after delaying $Task[2].dly$ time units, only when *sem* is released by $Task_0$ can it be in ready state, which is shown in Line 11.

The TMSVL model of *ISR* is given in Fig. 8. When the OS begins to handle interrupt, the running task turns to the suspended state and this is shown in Lines 1 to 4. The interrupt handler takes 0.0002s, during this period, the running task is suspended by setting $Task[runTaskNum].ex$ to 0 and $Task[runTaskNum].spd$ to 1. When the interrupt processing is finished, the interrupted task leaves the suspended state and a task scheduling is triggered which is shown in Lines 5 to 7.

$ISR \stackrel{\text{def}}{=} \begin{array}{l} 1. (T, T+0.0002) \text{ (temp=runTaskNum and} \\ 2. \quad \text{keep(if(temp!=-1)} \\ 3. \quad \text{then\{ Task[temp].spd=1 and Task[temp].ex=0\}} \\ 4. \quad \text{));} \\ 5. \quad Q \text{ and} \\ 6. \quad \text{if(temp!=-1)} \\ 7. \quad \text{then\{ Task[temp].spd=0\}} \end{array}$

Fig. 8. TMSVL model of the ISR

4.2 Verification of the Application

Verification is based on constructing Normal Form Graph (NFG) [6]. Given a TMSVL program p , we can construct a graph named NFG that explicitly illustrates the state space of the program. An NFG is a directed graph, denoted as $G = \langle V, E \rangle$, 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 execution paths of the corresponding TMSVL program.

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 check 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 based on the toolkit MSV for supporting verification of TMSVL programs. The following properties are verified in MSV.

(1) Safety: when the lowest priority task, namely $Task_2$ is running, it should not be preempted by $Task_0$ or $Task_1$.

This can be expressed by p_1 as follows:

$$\square (Task[2].ac < Task[2].C \wedge Task[2].ac > 0 \rightarrow (inp = 1 \vee Task[2].ex = 1))$$

p_1 means that once $Task_2$ starts executing, namely, $Task[2].ac > 0$ and $Task[2].ac < Task[2].C$, it is either suspended by the interrupt ($inp = 1$) or at the executing state ($Task[2].ex = 1$). $inp = 1 \vee Task[2].ex = 1$ indicates that $Task_0$ and $Task_1$ are not executed. Since at each state, at most one task is executing ($\square (Task[0].ex + Task[1].ex + Task[2].ex \leq 1)$), which can be verified first. When interrupt occurs, none of the three tasks can execute, this can be expressed as $\square (inp = 1 \rightarrow Task[0].ex + Task[1].ex + Task[2].ex = 0)$ and has been verified using MSV.

(2) Timeliness property: the three tasks can always finish in their deadline periodically.

This can be expressed by p_2 below:

$$(0, ST)true; \wedge_{i=0}^2 (Task[i].rd = 1 \wedge Task[i].ac = 0 \rightarrow \{0, Task[i].D\}true; Task[i].ac = Task[i].C)^+$$

p_2 means that from the time point ST , once $Task_i$ is ready implies $Task_i$ finishes in $Task[i].D$ time units. $\{0, Task[i].D\}$ is the delay operation, and is derived from time constraint statement. $\{0, Task[i].D\}true; p$ means p holds in $Task[i].D$ time units from the current time point. $+$ is the *chop-plus* operator, and is derived from the sequential operator $(;)$. $(p)^+$ means p repeats for any positive number of times.

Verification Results and Analysis. In order to verify the two properties, we set the computation time for the three tasks with $Task[0].C = 0.4$, $Task[1].C = 0.6$ and $Task[2].C = 0.8$, and the delay time with $Task[0].dly = 0.8$, $Task[1].dly = 0.6$ and $Task[2].dly = 0.4$. Meanwhile, we set the deadline for the three tasks with $Task[0].D = 1.2$, $Task[1].D = 1.5$ and $Task[2].D = 1.8$. We assume the start time of the application is at $T = 0$, that is, $ST = 0$.

The verification process with the extended toolkit MSV is conducted. With the system model and desired properties as input, we can eventually know whether or not the property is valid on the system model. The correctness property is verified and there is no path in the NFG, which indicates the property is valid.

Figure 9 is the verification result of the timeliness property. There are 3521 nodes and 3521 edges on the counterexample. Each node represents a program

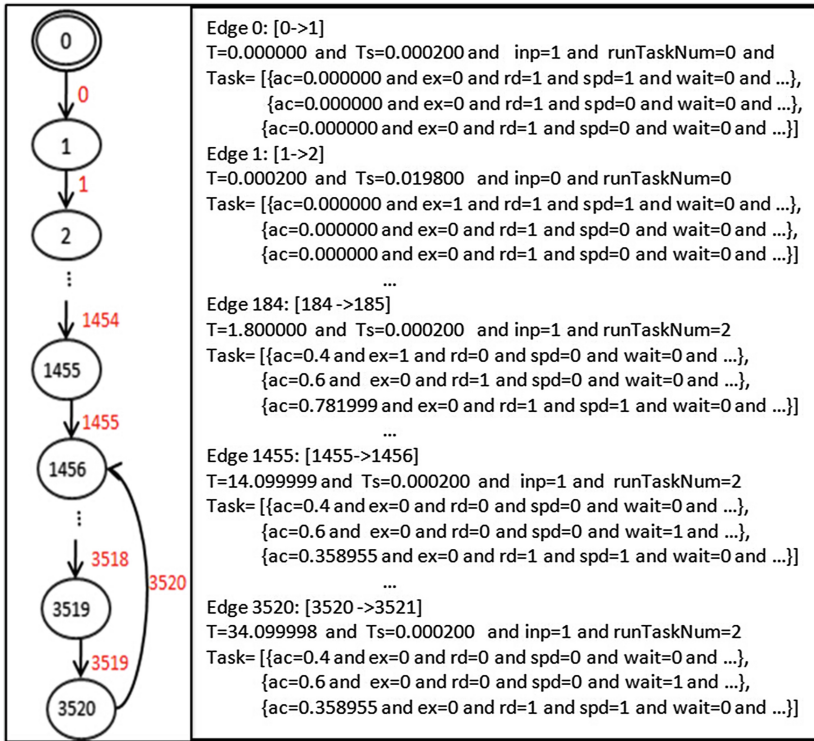


Fig. 9. Verification result of p_2

while each edge represents a state which shows the executing of the application at different time. The root node is a double circle, it represents the TMSVL program of the application. By executing the program that a node represents, two part can be obtained: the current part and future part where the current part is an outgoing edge of that node and points to the node which represents the future part. For example, edge 0 and node 1 are the executing results of node 0. The edges are state formulas while the nodes are temporal formulas which need to be further executed. In Fig. 9, we can see that edge 0 represents the state that $T = 0$, and all the three tasks are not delayed since $rd = 1$ for the three tasks. While $Task_0$ is the ready task with the highest priority, therefore $runTaskNum = 0$. $inp = 1$ indicates that the interrupt request occurs and at the second state, namely edge 2, the interrupt request is fulfilled. Then $Task_0$ starts executing. We can see that the time interval Ts is changeable and the time is measured in seconds.

In Fig. 9, edge 18 shows that when $T = 1.8$ s, the accumulated execution time of $Task_2$ is not equal to $Task[2].C$, that is, $Task_2$ does not finish in the given deadline $Task[2].D = 1.8$. Thus the timeliness property is violated in this case.

The three tasks start at the same time $T = 0$. $Task_0$ executes first, which is finished at $T = 0.4042$ s. Then it waits 0.8s, during this duration, $Task_1$ is executing. It finishes after 0.6006s, namely at $T = 1.0102$. Then $Task_1$ waits 0.6s and $Task_2$ gets the opportunity to run. Once $Task_2$ starts running, it is suspended by the interrupt but it cannot be preempted, for $Task_2$ needs to access the mutex semaphore, which makes its priority raise and be higher than other tasks. After 0.8016s, it is finished and its priority is recovered. Then it turns to waiting state at $T = 1.8182$. Therefore, $Task_2$ finishes after starting for 1.8182 time units, which is greater than the given deadline. This is consistent with the verification result of our toolkit.

5 Conclusion

We present a unified model checking approach to verify real-time systems running under $\mu C/OS-III$. The timer interrupt mechanism of $\mu COS-III$ is formalized in TMSVL. As a case study, a multi-task application with timer interrupt is provided and verified in TMSVL. With the toolkit MSV, the TMSVL program can be executed and the desired property of the system can be verified automatically. The mechanism that time intervals are adjustable for modeling improves the efficiency of verification. In the near future, we will further investigate modeling and verification techniques of time delay, timeout and other aspects of $\mu C/OS-III$ applications on the basis of TMSVL.

References

1. Alur, R., Dill, D.L.: A theory of timed automata. *Theor. Comput. Sci.* **126**(2), 183–235 (1994)
2. Behrmann, G., David, A., Larsen, K.G.: A tutorial on uppaal. *Lect. Notes Comput. Sci.* **4**(12), 200–236 (2004)
3. Bozga, M., Daws, C., Maler, O., Olivero, A., Tripakis, S., Yovine, S.: KRONOS: a model-checking tool for real-time systems (Tool-presentation for FTRTFT 1998). In: Ravn, A.P., Rischel, H. (eds.) FTRTFT 1998. LNCS, vol. 1486, pp. 298–302. Springer, Heidelberg (1998)
4. Choi, Y.: Model checking trampoline OS: a case study on safety analysis for automotive software. *Softw. Test. Verif. Reliab.* **24**(1), 38–60 (2014)
5. Dijkstra, E.W.: Notes on structured programming. *Structured Programming*, pp. 1–82. Academic Press Ltd., New York (1972)
6. Duan, Z., Tian, C.: A unified model checking approach with projection temporal logic. In: Liu, S., Maibaum, T., 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, J.C.: An approach to program testing. *ACM Comput. Surv. (CSUR)* **7**(3), 113–128 (1975)
10. Labrosse, J.J.: *uC/OS-III. The Real-Time Kernel*. Micrium Press, Weston (2009)
11. Baier, C., Katoen, J.-P.: *Principles of Model Checking*. The MIT Press, Cambridge (2008)
12. Lundqvist, K., Asplund, L.: A raven-scar-compliant run-time kernel for safety-critical systems. *Real-Time Syst.* **24**(1), 29–54 (2003)
13. Lv, M., Guan, N., Deng, Q., Ge, Y., Wang, Y.: Static worst-case execution time analysis of the $\mu\text{C}/\text{OS-II}$ real-time kernel. *Front. Comput. Sci. China* **4**(1), 17–27 (2010)
14. Waszniowski, L., Hanzlek, Z.: Formal verification of multitasking applications based on timed automata model. *Real-Time Syst.* **38**(1), 39–65 (2008)
15. 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)

Structured Object-Oriented Formal Language and
Method

5th International Workshop, SOFL+MSVL 2015, Paris,
France, November 6, 2015. Revised Selected Papers
Liu, S.; Duan, Z. (Eds.)

2016, VIII, 219 p. 90 illus. in color., Softcover

ISBN: 978-3-319-31219-4