MATEC Web of Conferences **35**, 03003 (2015)
DOI: 10.1051/matecconf/20153503003

Formalizing Real-Time Embedded System into Promela

Punwess Sukvanich[1,a], Arthit Thongtak [1], Wiwat Vatanawood [1]

[1]*Department of Computer Engineering, Faculty of Engineering, Chulalongkorn University, Bangkok, Thailand*

Abstract. We propose an alternative of formalization of the real-time embedded system into Promela model. The proposed formal model supports the essential features of the real-time embedded system, including system resource-constrained handling, task prioritization, task synchronization, real-time preemption, the parallelism of resources via DMA. Meanwhile, the model is also fully compatible with the partial order reduction algorithm for model checking. The timed automata of the real-time embedded system are considered and transformed into Promela, in our approach, by replacing time ticking into the repeated cycle of the timed values to do the conditional guard to enable the synchronization among the whole system operations. Our modeling approach could satisfactorily verify a small real-time system with parameterized dependent tasks and different scheduling topologies..

1 Introduction

In general, the real-time embedded systems are actively found and exploited in safety critical applications such as aerospace, automotive and medical industries. However, for the last two decade, real-time embedded systems and their applications are expanded to the fast-evolving industries such as telecommunication, multimedia and consumer electronics [1]. The environment of those new coming applications required cost efficient and shorter time-to-market.

Every real-time embedded system has the same correctness, safety, and liveness requirements, and it has to be developed under strict time constraints [2-4]. The correctness of the time-critical system seriously depends on the time logical results that are produced within any exact period [5]. Using run-time testing, simulation, and traditional verification, are still not sufficient to verify the correctness of these mentioned systems because infinitely time of their operations are not covered. Typical real-time embedded systems are designed to run infinitely. It means the infinite set of inputs.

The usage of ω-automata is considered to accept these infinite execution patterns[6]. A ω-automata is known as a finite state automaton that runs infinitely rather than finitely [7] and one of the extensions of ω-automata is timed automata. A timed automaton is a finite automaton with a finite set of the real-valued clock [7]. It is one of the most popular methods for modeling a real-time system. The timed automata approach will be used to specify, design, and verify the correctness of the real-time systems [8, 9].

Technically, the real-valued clock in the real-time embedded system will be replaced with the repeated cycle of a clock variable which being handling with a *Ticking* process. Any model checking tool handling the finite state automata should accept and do the formal verification of the mentioned timed automata of the real-time system[10]. Because of the mentioned techniques, the use of model checking could replace simulation or run-time testing on the actual system[1].

2 Background

In a real-time system, there is two type of tasks. First, a periodic task required a strict timing requirement. This task has a severe constraint that the task has to execute periodically at the precise time. The second ones are the event-based tasks which are less strict to the timing requirements. The execution of the event based tasks could be delayed in case the expected resources are not available. A system is called hard real-time system when it consists of at least one periodic task. In a hard real-time system, tasks have to satisfy the strict timing constraint. Each task will have the variation of execution time depending on its initial state, input data, and system environment. The set of all possible execution path has to be computed and expectedly verified to assure the correctness of its timing constraints.

Timing analysis is one of the key methods to detect problems. Timing analysis is one of the key methods to detect problems and is used to determine the maximum periods of each task on its execution times. The duration of execution time depends on the execution path. If the system control flow is straightforward, the timing analysis will easily perform. The task relationship could be a dependency. A dependent task and dynamic scheduling cause the timing analysis more complicated. In practice, the control flow would depend on the state and hard or else it is impossible to be determined [11].

[a] Corresponding author: punwess.s@student.chula.ac.th

3 Literature Review of Implementation of the Real-time System in Promela.

Promela is a process modeling language introduce in [6, 12, 13]. It was used to model the logic of parallel systems. Among the processes, they communicate with buffered message exchanges or rendezvous operations and shared global variables. SPIN is an explicit model checker for Promela. SPIN verifies a Promela model to ensure correctness using LTL formulas or never claims. There are several works of implementation of timed automata into Promela such as mentioned in [14-16]. A time slice defined as tick representing a granule duration of real-valued process time in the real-time systems are proposed in [14, 16]. A variable called tick is defined as the time index using a positive integer. The hard real-time system requires any task to operate at any particular value of tick. The process prioritization was implemented for the real-time system as well such as in [14]. . It is also a built-in feature in SPIN distribution 6.2.0. The higher priority process always takes over lower priority unless it is blocked [14]. Unfortunately, it is not compatible with the Partial Order Reduction (POR). Whenever POR is used, the process priority will be disabled. The POR algorithm is the primary strength of the SPIN model checker [17], especially for the discrete timed model. Without the POR, it is almost impossible to verify full state space and handle with the state space explosion.

In SPIN, the execution of statements is asynchronous and interleaved. To model real-time system that required a synchronization might have a problem with a task that can accidentally execute before its time slice. In [16, 17] , the timeout statement that is a predefined statement in SPIN is used to create the synchronization of tasks without using the rendezvous operation.

4 Our Approach of Formalizing the Real-Time Embedded System

Several required features of the real-time embedded system are preemptive, priority based scheduling, and resources management. The use of logic blocking is the easiest way and very lightweight to cope with the prioritization feature. The most valuable resource for the real-time system is the computation because of the lack of processing power and the bottleneck of the transmission between them via buses. Fig. 1 shows our case study of a health tracker device. The device is a heart rate tracker implemented by a small real-time embedded system with a small display where health information is displayed.

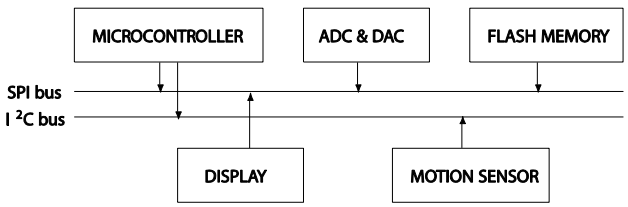

Figure 1. The diagram of the case-study system.

The device consists of the low-power 16-bit microcontroller. The microcontroller has I2C and SPI peripheral buses that are used to interface with other peripherals. The device is not safety critical, but it is still useful for a case study. This device wakes up and has the CPU burst for a short duration at 20Hz, which is a timing period of 50 milliseconds. The device is defined as a hard real-time system because it consists of periodic tasks.

5 Implementation

We create a set of parameters of task model that can represent the required parameter from the real-time embedded system. The required parameters are a name, resource requirement, type of task, execution duration, state phase, task priority and task period. In Table 1. we create a set of parameters of case study device that can represent the required processes.

Table 1. Task parameters in model.

Name	CPU	I^2C	SPI	Periodic	Duration	Priority
Wear	O			True	5	10
Act	O		O	True	7	9
RTC	O	O		True	8	8
Samp	O		O	True	7	7
Power	O		O	True	2	4
HRCal	O			False	2	4
HR	O			False	6	3
Flash	O			False	5	5
Pedo	O			False	2	4
Dsp	O			False	5	4

5.1 Real-valued time

We define the real-valued clock of the system as a synchronization primitive as well as resource management. The one millisecond time slice is defined and a tiny portion of time which is smaller than one millisecond will be rounded up to one millisecond. We define a minor loop that has 50 ticks in a total of 50 milliseconds. For each loop, the microcontroller would wake up from sleep and then process any required task and then goes deep sleep again. A tick counter will start from the first tick (tick=0) and is increased by one until its 50th tick (tick=49), the tick counter will reset to 0 (line 9 of fig.2) and repeatedly start again as mentioned. Our proposed model could reduce the number of states to four states per tick. The total state space will be then reduced as well as its complexity.

```
1   #define isMyPhase (((Tick - (myTask.phase)) % myTask.period == 0))
2   mtype = {Open, Close, Execute, Schedule}
3   proctype Ticking() {
4   do::((ClockState == Open)&& timeout) ->ClockState=Schedule;
5      ::((ClockState == Schedule)&& timeout) ->ClockState = Close;
6      ::((ClockState == Close)&& timeout) ->ClockState = Execute;
7      ::((ClockState == Execute) && timeout) ->ClockState = Open;
8            Tick++;          /* advance to next tick */
9       if :: (Tick > (TotalTickPerLoop)) -> Tick = 0;
10         :: else;
11      fi;   /* implement reset all resources here */
12  od;}
```

Figure 2. Promela model of Tick process

5.2 Ticking process

We propose an alternative of *Ticking process*, which handles the clock mechanism of the system. A Ticking process contains the infinitely often execution of changing of the ClockState variable. The ClockState contains four states that are Open, Schedule, Close, and Execute (shown at line 4-7 of Fig. 2). Every task in the real-time system uses ClockState to synchronize themselves with another task. The timeout statement should appear between each clock state to ensure that every task will work under the same ClockState. The tick could only be proceed if all four state values of ClockState are sequentially parsed.

5.2.1 Open State of ClockState

The first state value of our clock mechanism is defined as an Open state. The Open state allows all tasks in the system to check if the current tick is their arrival time or not. Initially, every task stays waiting until the guard condition called *isMyPhase* is true shown at line 9 in Fig.4. The guard condition will block any particular task that has unsatisfied values of the predefined parameters 'period' and 'phase'. An event task is setting period to 0 (period=0) and it can be active at any time slice if available. A task arrival time can start at any tick possible by setting phase parameter. For example, a periodic task that executes every ten ticks if the period parameter is set to 10 (period=10). Any task that would ready to execute (*isMyPhase*=true) if its arrival time will be setting to active (*isTaskActive*=true) shown at line 9 of Fig. 4).

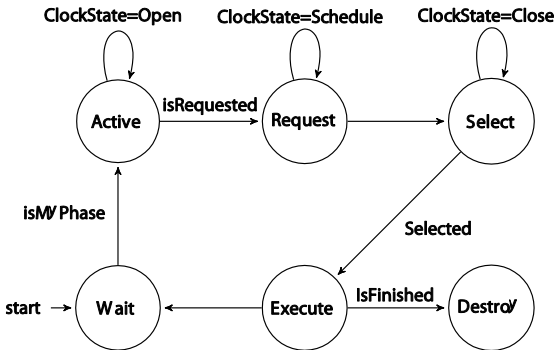

Figure 3. State machine of Task

An Active task will request a resource every time when ClockState is in Open state even the resource is not available. Active task can request resource by setting its *is_Requested* flag. If there is more than one active task in the same time slice, each task will execute concurrently. Instead of using a loop for checking every task. We use timeout in every ClockState to wait till every Active task requesting their resources. After no more request from any task, the ClockState advances to the next state. With the timeout statement, it is guaranteed that there is no Active task that have requested the resources left. This statement will ensure that every task can request resource on their ticks precisely before ClockState to Schedule state.

```
1   proctype CreateTask(taskdef myTask){
2   int cycleCount = 0;          /* Cycle counter */
3   int durationCount = 0;       /* Duration counter for task */
4   bool isTaskActive = false;  /* Guarding for this task if not Active yet*/
5   mtype RES[MaxResource];
6   WAIT:   /* When finish every tick, task will wait here */
7   do
8   ::((!isTaskActive)&&(GlobalClockState==Open)
9     &&(isMyPhase))->ACTIVE:      isTaskActive=true;
10  ::((isTaskActive)&&(GlobalClockState==Open)
11    &&(!Tlist[myTask.my_type].is_Requested))->
12    TaskArray[myTask.my_type].is_Requested = true;
13  REQUEST:
14  ::((isTaskActive)&&(GlobalClockState == Schedule))->
15     /* Implement Scheduler
16  here*/  ::((isTaskActive)&&(GlobalClockState==Close)
17     &&(!isLessPriority(myTask.my_type))
18     &&(Selected == none)
19     &&(TaskArray[myTask.my_type].is_Requested))->
20   SELECTING: Selected = myTask.my_type;
21  :: ((isTaskActive)
22      &&(GlobalClockState == Execute)
23      &&(TaskArray[myTask.my_type].is_Requested)
24      &&(Selected == myTask.my_type))->
25  EXCUTION:  TaskArray[myTask.my_type].is_Requested = false;
26      durationCount++;     /* Increase task execution duration */
27       if :: (durationCount >= myTask.duration)->durationCount=0->
28            goto TaskIsFinished;
29        :: else->skip;
30        fi;
31        HighestGetResource = true;
32   ::((GlobalClockState == Excecute)&&(isTaskActive)
33    &&(TaskArray[myTask.my_type].is_Requested)
34    &&(HighestGetResource)&&(CheckMyResource))->
35  Tlist[myTask.my_type].is_Requested = false;
36  Tlist[myTask.my_type].is_Scheduled = false;
37   If ::(Resource[0].owner==_IDLE_ && myTask.isreqRES0)
38        ->Resource[0].owner=myTask.my_type;
39      :: else;
40   fi;
41   if ::(Resource[1].owner==_IDLE_ && myTask.isreqRES1)
42      ->Resource[1].owner=myTask.my_type;
43     :: else;
44  fi;
45  if  ::(Resource[2].owner==_IDLE_ && myTask.isreqRES2)
46      ->Resource[2].owner=myTask.my_type;
47      :: else;
48  fi;  durationCount++;
49  if  :: (durationCount >= myTask.duration)->durationCount=0
50       ->goto TaskIsFinished;
51      :: else->skip;
52  fi;
53  GlobalClockState == C_Open;
54   od;
55  TaskIsFinished:  isTaskFinished[myTask.my_type] = true;
56   durationCount = 0;
57   isTaskActive = false;
58   if ::(myTask.is_periodic)->cycleCount=0;
59      goto WAIT; /* Run infinitely eventually */
60      ::((cycleCount+1) < myTask.cycle)->
61      cycleCount++;printf("cycle count %d\n",cycleCount); goto WAIT;
62      ::else->cycleCount=0;
63   fi;
64   DESTROY:
65   }
```

Figure 4. Promela model of Task process

5.2.2 Schedule state of ClockState

The second state value of ClockState, where the scheduler collects all the resources request from every active task. In our implementation, we are using fixed rate priority preemptive scheduling topology. Each task has predefined priority. A higher priority task will always get the resources. Tasks that have been requesting the

resources, have to check if they have right to hold the resources by checking *isLessPriority* flag. Task with the same level of priority will have equal chance to capture resources non-deterministically. Task with the highest priority will be selected and granted the right to hold the resource for a tick. Another supported scheduling topology such as EDF (Earliest Deadline First), DM (Dead-line monotonic) for example (shown in fig.5).

```
1    inline Scheduler_EDF(TaskType){
2    if ::(!iMoreDeadline(TaskType))-> Selected = myTask.my_type;
3      ::else->skip;
4    fi; }
5    inline Scheduler_DM(TaskType){
6    if ::(!iMoreDuration(TaskType))-> Selected = myTask.my_type;
7      :: else->skip;
8    fi;}
```

Figure 5. Promela model of scheduling Topology macros

5.2.3 Close state of ClockState

The third state value of ClockState, which will select the task that have the right to hold the resource. The *Selected* task will get the resource. If no scheduling topology is presented, tasks have to check if they have right to hold the resources by checking *isLessPriority* flag. Task with the same level of priority will have equal chance to capture resources non-deterministically. Task with the highest priority will be selected (at line 17 of Fig.4) and granted the right to hold the resource for a tick.

5.2.4 Execute state of ClockState

The fourth state value of ClockState. The resource will be registered to each selected task (at line 38 ,42 ,46 of Fig.4) The task execution duration will be decreased. Then, all resources are cleared for the next tick. The finished task would be destroyed if it is not a periodic task.

5.3 Priority and rendezvous operation

In the real-time system, the priority scheduling is used to manage timing constraint. If two or more tasks have active and request to hold the resources for execution. A priority is used to justify a right to access the resource. We propose the implementation of priority concept as a positive integer variable. In the Close state, each task has *!isLessPriority* guard (at line 17 of Fig.4) which will block the task from its execution. In Fig.6, the *isLessPriority* guard used to compare its task priority with another task and result in false only if a top priority task is selected. Priority variable could be changed during execution that allows us to implement the most complicated schedule. Only the active task will be checked. All of the active tasks will request the resource from the Open state.

```
1    #define TASK1isLessPriority
2    ((TASK1.Priority<TASK2.Priority) && TASK1.is_Requested)
3    || ((TASK1.myPriority<TASK3.myPriority)&&TASK3.is_Requested)
     || ((TASK1.myPriority<TASK4.myPriority)&&TASK4.is_Requested))
```

Figure 6. Promela model of *isLessPriority* macro

5.4 Resources

Each task execution required a particular resource. We propose the three resource model that are most affect to the timing constraints. From our model, we define two different buses for each transmission of I2C and SPI bus. The task has to hold the resource and execution at the specific tick for a specific duration to complete execution (at line 17 of Fig.4).

5.5 Pre-emptive

We design the model to support preemption of a task. In a periodic task, preemption could help task to operate within the deadline. The task with higher priority could preempt the current task at any tick. To prevent the task from preempting, we can define the schedule to increase the priority of the currently executing task to highest. In fig.7, we define process P2 as a high priority process that can preempt another task during execution. The process P2 preempts process P1 during the execution. After process P2 ends, then the process P1 can proceed to hold the resource and resume its execution.

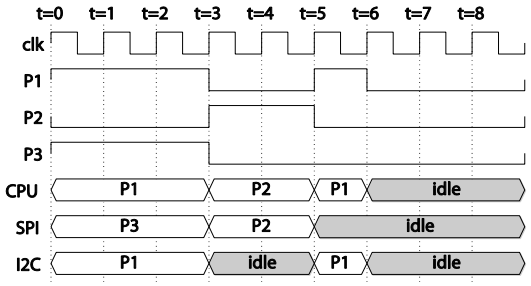

Figure 7. Timing diagram of preemptiving in model.

5.6 Dependent Task

To model the dependent task, we define a process to handle the control flow of the system shown in fig.8 Each feature may require a sequential execution of tasks, for example, sampling data from the sensor, process the data, writing a log file in volatile memory, and then record into non-volatile (flash) memory. The mentioned tasks have to be processed sequentially during the execution time that is required different timing requirements (process duration) and resource requirements (buses). After each process finishes its execution, the *isTaskFinished* flag become true. The scheduler waits for the *isTaskFinished* flag from any process. We allow developer to change the control flow without changing the clock automaton.

```
1    active proctype ControlFlow() {
2    do
3     :: isTaskFinished[Wear]-> isTaskFinished[Wear]=false;
4       CreateTask_HR;
5     :: isTaskFinished[HR]-> isTaskFinished[HR]=false;
6       CreateTask_HRcal;FlashCount++;
7     :: isTaskFinished[HRcal]&&FlashCount>=15)->
8       isTaskFinished[HRcal]=false;CreateTask_flash;  FlashCount =0;
9     :: isTaskFinished[Samp]-> isTaskFinished[Samp]=false;
10      CreateTask_pedo;
11    :: isTaskFinished[pedo]-> isTaskFinished[pedo]=false;
12      FlashCount++;CreateTask_Act;HRDisplay++;
13    :: (isTaskFinished[Act]&&HRDisplay>=20)->
14      isTaskFinished[Act]=false;CreateTask_Dsp;HRDisplay = 0;
15   od;
```

Figure 8. Promela model of ControlFlow process

6 Result

Using our formalization approach, we conducted 10 small experimental tasks consisting of five periodic tasks and five-parameter dependent tasks. These tasks were designed to infinitely run as reactive system.

Table 2. Resulting Verification Criteria

Topology	State	Depth	Time(sec)	Mem(MB)
Fixed	195,569	91,906	2.13	283.996
EDF	90,968	94,533	0.958	202.113
DM	439,682	93,178	4.44	465.552

Our verification experiments used SPIN model checker version 6.4.3, running on a typical personal computer equipped with Intel I7 2.8 GHz. 12GB of RAM. In Table 2, the resulting verification criteria were evaluated for three types of scheduling topologies, the fixed rate priority, and the earliest deadline first, and the deadline monotonic. The resulting figures were quite satisfied and showed that our formal verification modelling approach could cope with the problems regarding the timing correctness and the memory usages.

7 Conclusion and Future work

In our approach, we formalize the model of the timing behavior of the real-time embedded system into Promela. The timed automata are considered to formalize the infinite behavior of the system into finite state automata. Our formal model represents the four-valued clock states and ticking process which scope the timing requirements of the real-time system. Our model able to handle all required feature to verify the correctness of our case studies real-time embedded device. The SPIN model checker can verify the correctness of scheduler that contain seven periodic tasks and several dependent tasks with priority feature in a few minutes. In our next implementation, we are going to implement more scheduling topology into the model to increase ease of use for the developer to verify their task and task scheduler.

References

1. Sifakis, J., *Modeling Real-Time Systems-Challenges and Work Directions*, in *Proceedings of the First International Workshop on Embedded Software*. (2001), Springer-Verlag. p. 373-389.
2. Dill, D.L., *Timing Assumptions and Verification of Finite-State Concurrent Systems*, in *Proceedings of the International Workshop on Automatic Verification Methods for Finite State Systems*. 1990, Springer-Verlag. p. 197-212.
3. Ostroff, J.S., *Temporal logic for real time systems*. (1989): John Wiley & Sons, Inc. 209.
4. Henzinger, T.A., Z. Manna, and A. Pnueli, *Temporal proof methodologies for real-time systems*. (1991), Stanford University.
5. Baier, C. and J.-P. Katoen, *Principles of Model Checking (Representation and Mind Series)*. (2008): The MIT Press. 975.
6. Holzmann, G., *The SPIN Model Checker: Primer and Reference Manual*. (2011): Addison-Wesley Professional. 608.
7. Thomas, W., *Automata on infinite objects*, in *Handbook of theoretical computer science (vol. B)*, L. Jan van, Editor. (1990), MIT Press. p. 133-191.
8. Alur, R. and D.L. Dill, *Automata for modeling real-time systems*, in *Proceedings of the seventeenth international colloquium on Automata, languages and programming*. (1990), Springer-Verlag New York, Inc.: Warwick University, England. p. 322-335.
9. Alur, R. and D.L. Dill, *A theory of timed automata.* Theor. Comput. Sci., (1994). **126**(2): p. 183-235.
10. Bouyer, P., et al., *Timed Modal Logics for Real-Time Systems.* J. of Logic, Lang. and Inf., (2011). **20**(2): p. 169-203.
11. Wilhelm, R., et al., *The worst-case execution-time problem—overview of methods and survey of tools.* ACM Trans. Embed. Comput. Syst., (2008). **7**(3): p. 1-53.
12. Holzmann, G.J., *Design and validation of computer protocols*. (1991): Prentice-Hall, Inc. 500.
13. Holzmann, G.J., *Design and validation of protocols: a tutorial.* Comput. Netw. ISDN Syst., (1993). **25**(9): p. 981-1017.
14. Mihai Florian, E.G., Gerard Holzmann. *Logic Model Checking of Time-Periodic Real-Time System*. in *Aerospace 2012 Conference*. (2012).
15. Tripakis, S. and C. Courcoubetis, *Extending Promela and Spin for Real Time*, in *Proceedings of the Second International Workshop on Tools and Algorithms for Construction and Analysis of Systems*. (1996), Springer-Verlag. p. 329-348.
16. Bosnacki, D. and D. Dams, *Integrating Real Time into Spin: A Prototype Implementation*, in *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XI) and Protocol Specification, Testing and Verification (PSTV XVIII)*. (1998), Kluwer, B.V. p. 423-438.
17. Bosnacki, D., *Partial Order Reduction in Presence of Rendez-vous Communications with Unless Constructs and Weak Fairness*, in *Proceedings of the 5th and 6th International SPIN Workshops on Theoretical and Practical Aspects of SPIN Model Checking*. (1999), Springer-Verlag. p. 40-56.