Scientific Bulletin of the Petru Maior University of Tirgu Mures
Vol. 6 (XXIII), 2009
ISSN 1841-9267

# Formal Verification and Implementation of Real-Time Applications

Liviu Hațegan, Piroska Haller
"Petru Maior" University of Tîrgu-Mureş, Romania
liviu.hategan@yahoo.com, phaller@upm.ro

## Abstract

*This paper presents a method for the formal description, verification and automatic source code generation of embedded real-time multitasking applications, based on a model consisting of networks of timed automata. The model describes a real-time operating system kernel and application tasks, taking into consideration both non-preemptive and preemptive scheduling. The timing properties of the proposed model can be verified using a model-checking tool. We also provide a solution for C source code generation based on the application's model. For this purpose a unified resource access interface was implemented.*

## 1. Introduction

Real-time embedded systems have become widely used in a large number of fields, especially in the industrial environment, playing an increasing role in the modern society and are rapidly evolving, growing in complexity. Moreover they are often used not only by themselves, but in clusters and networks.

A real-time system is in close relationship with the physical environment it interacts with, fact that induces a set of constraints the system must obey. A real-time behavior of an embedded system implies that *when* an operation is performed is just as important as *what* is executed. Furthermore, the complexity of such systems and the strictly timing requirements emphasize the necessity of a real-time operating system.

Because of the strict nature of a real-time embedded system, a thorough verification of its properties, to ensure that it satisfies all the requirements is crucial.

The timed automata formalism is widely used and well-proven in the description and verification of real-time systems [3][4][5][7]. In this paper we propose a simplified model for single processor microcontroller based systems. The model consists of a network of timed automata that describes a basic real-time kernel, the physical system's resources and concurrent application tasks. The method presents the possibility for the verification of essential properties such as: task schedulability, state reachability, access to shared resources and temporal constraints regarding the response to specific events. This verification is done using the UPPAL model-checker [8]. This approach allows for the system's simulation and the observation of its behavior before the actual implementation.

Using the model of the embedded system, we can also automatically generate the application's source code, avoiding most of the error-prone human coding [7].

For the implementation we chose the SAM7-EX256 development board and FreeRTOS - a portable, open source, mini real-time kernel [9]. The FreeRTOS scheduler is capable of running in either preemptive or non-preemptive (cooperative) mode. We also implemented hardware-specific elements: drivers including mechanisms for interrupt handling for every device and a module for unified peripheral access.

## 2. The application tasks

A real-time embedded application can be constructed as a collection of independent tasks. Tasks are processes managed by the real-time operating system accordingly to the scheduling policy.

Because some functionalities of an application are more important or have shorter deadlines than others, the tasks are prioritized. The scheduler ensures that the most important task will be given processor time.

Each task instance is modeled by a timed automaton that is synchronized with the OS model via channels.

***Keywords:*** *embedded real-time multitasking applications, cooperative scheduler model, pre-emptive application tasks*

### 2.1. Task behavior and properties

In general, embedded tasks present the following behavior[5][7]:

INFINITE_LOOP
- Request access to a resource (blocking call)
- Perform a read/write operation
- Perform a computation
- Request access to another resource (blocking call)
- Perform a read/write operation

……
END_INFINITE_LOOP

Resources will be accessed by tasks through blocking request calls. The desired resource is explicitly specified through its RID (resource ID) [6]. After making a request call, the task will enter a blocked state, where it will wait for the resource to become available.

The computations performed by the tasks are characterized by a worst case execution time (WCET) and a best case execution time (BCET). This is a consequence of branching instructions in the computations. Also, the running task will be delayed by the occurrence of interrupts generated by the resources. The ISR execution time will be added to the current task's *WCET* and *BCET* [3].

## 3. The resources

In the proposed model we are only interested in the resource's effect on the task behavior, therefore they will be considered only as data sources that can unblock waiting tasks [7].

In constructing the general resource model, we considered the following:
- Resources are reusable and can be shared, but only one task can have access to a resource at any given time.
- A task can request a single resource at one time (via the *Request(RID)* function).
- In the request call, the resource is explicitly specified through its RID.

Every resource has a minimal inter-arrival time (the MIAT). A resource can unblock a waiting task and provide data at anytime after its *MIAT* expires, but not sooner.

### 3.1. The unified access interface

In our implementation on the SAM7-EX256 development board, resources provide interrupts that are managed by drivers. Depending on the peripheral, data is read from registers and stored in queues when the interrupt occurs for an input peripheral or data is sent when an output peripheral is ready to accept it. A task that is waiting on the resource's queue will immediately be unblocked.

The unified resource access interface consists of a *Request*, a *Read* and a *Write* function. The *Read* and *Write* are nonblocking calls, they will be performed after the access to the requested resource is granted. Each resource has a state variable assigned. The read and write operations will be performed using the state variables.

## 4. The cooperative model

### 4.1. The cooperative application tasks

All cooperative tasks have three states:

READY: the task is able to execute, but has not been scheduled yet.

RUN: the task is utilizing the processor. Only one task can be in this state at any given time.

BLOCKED: the task is waiting for an event. It is not available for scheduling.

Considering the general task form described in the previous section, we present a simple task model that requests access to a resource (RID), performs a read operation and executes a computation (Figure 1). The task automaton is synchronized with the resource and scheduler models via channels. Tasks can be identified through their unique ID (the constant *pid*).

```
Task1
        {
            Loop
            { Request(RID);
              Read(RID, var_rid, size);
              computation();
            }
         }
```

**Figure 1 a) Simple task in pseudocode**

The task automaton's states:

READY1,READY2: the task is available for scheduling, awaiting synchronization with the scheduler via the *schedule[pid]?* channel. The initial *READY1* location represents the task's state at system startup, when all tasks are ready to execute.

RUN1, RUN2, RUN3: represent the states where the task has control of the processor.

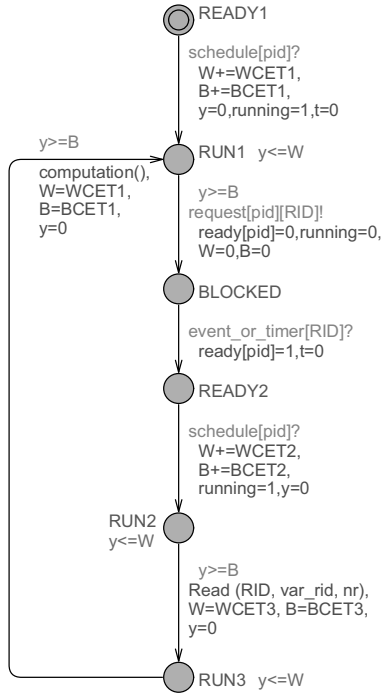


**Figure 1. b) Simple cooperative task's model**

These locations are characterized by a best case and worst case execution time (*WCET, BCET*). The *WCET* is modeled by the location's invariant (*y<=W*), allowing the system to remain in that state only as long as the *y* clock's value is smaller or equal to the *W* integer variable. The guard on the outgoing transition (*y>=B*) simulates the state's *BCET*, insuring that the transition will not occur until the clock *y* has a higher or equal value to that of the *B* variable. On the incoming transition, the *W* and *B* variables are initialized with the *WCET* and *BCET* of the current location and the clock *y* is reset.

While being in a running state, a task can be interrupted by a resource's ISR. In this case the duration of the interruption will be added to the *W* and *B* variables, increasing the time spent by the automaton in the current location.

BLOCKED: the state is entered as a result of the request for a particular resource through the *request[pid][RID]!* channel. The *ready[pid]=0* update on the incoming transition specifies that the task is unavailable for scheduling. Also, the variable *running* is reset, symbolizing that the task is not in a running state and the value 0 is assigned to *B* and *W*. The state will be left when the resource becomes available (the *event_or_timer[RID]?* channel is activated), the *ready[pid]* value being set to 1.

If there is no task running or ready, the FreeRTOS kernel schedules the *Idle Task*. It is automatically created when the scheduler is started and has the lowest priority. Following the general task form, the *Idle Task* requests the *NULL* resource, periodically yielding processor control. So that it is always available for scheduling, is has no blocked state, the *NULL* resource being considered as always available. The Idle Task automaton is presented in Figure 2.

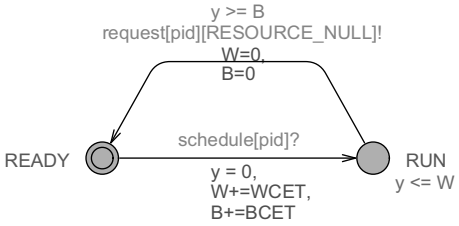


**Figure 2. The idle task's model**

## 4.2. The cooperative scheduler model

In the cooperative behavior the running task will have full control of the processor, regardless of its priority, until it makes a blocking call. The task can explicitly invoke the scheduler by calling the *taskYIELD()* macro or by requesting access to a resource using the *Request(RID)* function implemented in the unified resource access interface. The task will block until the resource becomes available. At this point the next task to be scheduled will be selected. The scheduler will determine the highest priority task that is ready to execute and it will be given control of the processor until it yields the processor by itself or requests a resource.

The cooperative scheduler is described by a timed automata that presents three states: INIT, SELECT and IDLE (Figure 3).

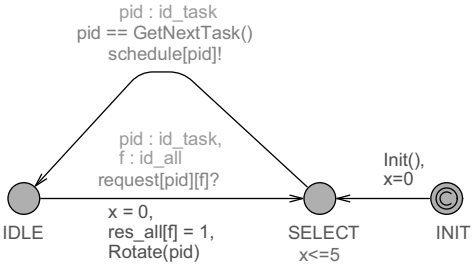


**Figure 3. The cooperative scheduler's model**

The scheduler automaton's states:

INIT: the necessary hardware settings and initialization of task priorities and data structures take place. Because the state is committed, the scheduler will leave this state immediately at startup, taking the transition towards the *SELECT* state.

SELECT: the ready task with the highest priority is chosen for scheduling (the *GetNextTask()* function). The invariant *x<=5* specifies the time needed to select the next task, the value can be changed to match the actual physical time, witch is hardware-dependant.

IDLE: the previously chosen task is running. The state is left when the task makes a blocking request call. In order for the tasks with the same priority to have an equal chance at scheduling, the *Rotate()* function is called when the scheduler exits the *IDLE* state.

### 4.3. The resource model

Figure 4 illustrates the resource model.

The *MIAT* values for all of the system's resources are stored in the array *all_period[NR_RESOURCES]*.

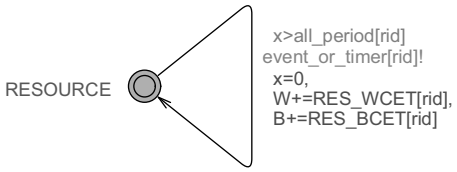


**Figure 4. The resource model**

The guard *x>all_period[rid]* ensures that the transition doesn't take place before the resource's *MIAT*. When the transition does occur, the *event_or_timer[rid]!* channel is triggered. The *RES_WCET[pid]* and *RES_BCET[pid]* constants are used to increase the values of the *W* and *B* variables, delaying the task interrupted by the resource ISR.

In case a task must execute an action periodically, at strict interval, it can utilize the timer resource. Figure 5 presents the timer automaton.

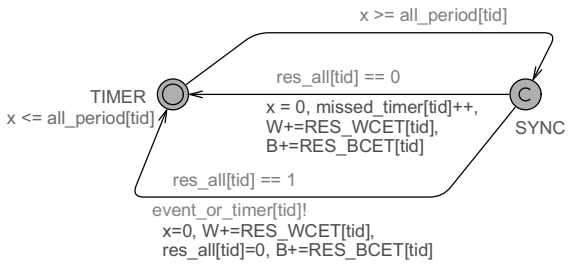


**Figure 5.Timer resource**

The timer will unblock a waiting task periodically by sending a signal through the *event_or_timer[tid]!* channel when the *x* clock has the same value as *all_period[tid]*. The constant *tid* represents the timer's RID.

The timer automaton's states:

TIMER: is the initial state, where it will remain until the predefined period expires. This state is left only when the clock *x* has the exact value as *all_period[tid]*.

SYNC: this state is reached when the timer expires. The state is committed so it will be immediately left, the automaton unblocking any waiting task via the *event_or_timer[tid]!* channel. In order to avoid system deadlock, the timer is allowed to expire even if none of the tasks are blocked waiting for it. This is ensured by the fact that there are two edges from the *SYNC* to the *TIMER*. state.

The FreeRTOS tick hook function is used to implement the timer's functionality. We have also developed drivers for the physical system's hardware timers. The resource model is identical for both preemptive and cooperative systems.

## 5. The preemptive model

### 5.1. The preemptive application tasks

In addition to the READY, RUN and BLOCKED states presented by the cooperative tasks, the preemptive versions also have a SUSPENDED state. This state is entered when the task is preempted and it will remain in this location until rescheduling.

Figure 6 presents the preemptive version of the simple read-request-computation task model described in section 4.1.

The task automaton's states:

READY1, READY2: the ready states have the same role as in the cooperative task's case.

RUN1, RUN2, RUN3: in addition to the characteristics presented in section 4.1, from a running state the task can enter a suspended state if preempted by the scheduler.

SUSPENDED1, SUSPENDED2, SUSPENDED3: the running task can be suspended via the *suspend[pid]?* channel. The state's invariant *y'==0* will stop the *y* clock from advancing while the automaton is in a suspended location. The task will re-enter the running state after being rescheduled (via the *schedule[pid]?* channel) and the clock will be restarted by the *y'==1* expression in conjunction with the state's invariant.

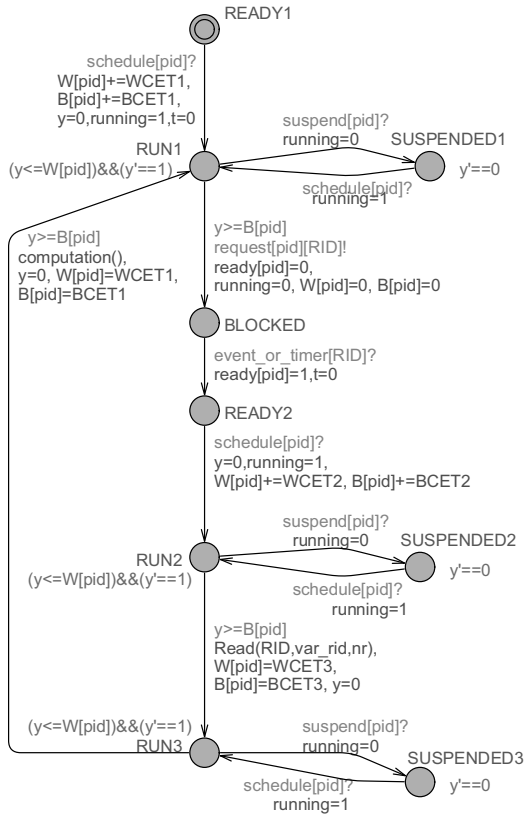


**Figure 6. Preemptive task's model**

The *Idle Task* is the same as in the case of the cooperative model, except the fact that it doesn't suspend itself by requesting the *NULL* resource, but it is preempted by the scheduler.

## 5.2. The preemptive scheduler model

The preemptive scheduler periodically performs a context switch, temporarily suspending the running task in favor of an equal or higher priority one. It does so by using the tick interrupt. The FreeRTOS tick interrupt increments a tick count variable with strict temporal accuracy, providing a time base to the scheduler. Each time the interrupt occurs, the kernel determines if a task must be unblocked or awaken. If a higher (or equal) priority task is ready to execute, the scheduler will preempt the current task and schedule the new one. As a consequence, tasks of the same priority will have a time-slice equal to the kernel tick interval.

The preemptive scheduler's model differs from the cooperative one by the fact that it can suspend a task that is in a running state via a synchronization channel. The model is illustrated in Figure 7.

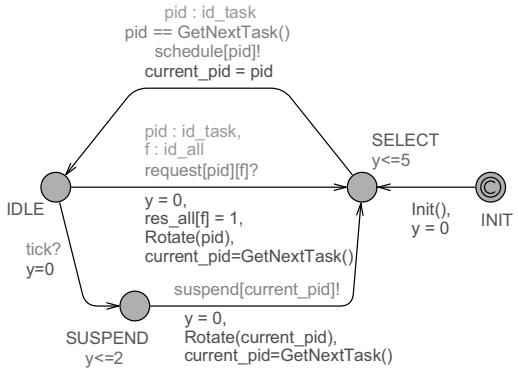


**Figure 7. The preemptive scheduler's model**

The scheduler's INIT,SELECT and IDLE states have the same role as in the cooperative scheduler's case.

The automaton reaches the *SUSPEND* state when it is in the *IDLE* location and receives a signal through the *tick?* channel. Here, the current task is suspended via the *suspend[current_pid]!* channel and another one will be selected in the next state (*SELECT*).

The scheduler is synchronized with the Tick_Interrupt automaton (Figure 8).

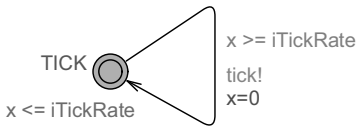


**Figure 8. Tick_Interrupt automaton**

The *x<=iTickRate* invariant and the *x>=iTickRate* guard ensure that the transition occurs exactly when the *x* clock has reached the *iTickRate* constant's value, a signal being sent through the *tick!* channel periodically.

# 6. Simulation and formal verification

Before simulation and verification, the time-related constants (*WCET*, *BCET*, *MIAT*, timer period, kernel tick rate, etc.) must be adjusted to fit the actual physical system's characteristics. The simulation can be performed using the UPPAAL integrated simulation tool. The requirements specifications must be expressed in UPPAAL's CTL subset [10].

The behavioral properties verifiable are: reachability, safety and (bounded) liveness.

Reachability properties: verify if it is possible to reach a state in witch a formula *p* is satisfied. Such properties are expressed using the path formula *E<>* *p* ("p – is reachable"):

- E<> Task1.RUN1 – checks if *Task1* will ever reach the RUN1 state (therefore executing the corresponding computation).
- E<> Idle_Task.RUN – checks if the *Idle*_Task will ever be scheduled (all user tasks are blocked).
- E<> Task1.SUSPENDED1 – verifies if *Task1* will ever be suspended, a context switch taking place (for the preemptive version of the model).

Safety properties: state that a formula *p* is satisfied in all reachable states or that there is a path in witch *p* is always true. This can be expressed by the path formulae *A[] p* ("invariantly p") and *E[] p* ("p – potentially always true"):

- A[] not (Task1.running and Task2.running) – two different tasks can not be running at the same time.
- E[] Task1.READY2 imply Task1.t<=500 – there is a path in witch *Task1* will not spend more than 500 time units in the READY2 state.

Liveness properties: in all cases, the automaton will eventually reach a state where a property *p* is true. Another form is that if a property *p* is true, another property *q* will become true eventually. This can be formulated with the *A<>* *p* formula ("inevitably p") or *p* --> *q* ("p leads to q"):

- A<> Task1.RUN1 – *Task1* will inevitably be in the RUN1 state at some point.
- Timer(6).SYNC --> Task5.RUN2 – considering a blocked task waiting for a timer, if *Timer(6)* expires *Task5* will inevitably be scheduled.
- Task2.SUSPENDED1 --> Task2.RUN1 – if a task is suspended, it will eventually be rescheduled (preemptive model).

Bounded liveness properties: whenever *p* becomes true, *q* becomes true within the time limit *t*:

- Timer(6).SYNC --> (Task5.RUN2 and Task5.t<=200) – when *Timer(6)* expires *Task5* will be scheduled within 200 time units.

Deadlock freeness: UPPAAL features a special formula for verification of deadlock freenes: *A[] not deadlock* [10].

## 7. Source code generation

Once assured the application meets all the requirements, we can proceed with the implementation. We have developed a code generator application that, based on the model presented as input (XML file), will recognize the states and function calls of each task and output the corresponding source code skeleton.

In the case of the simple request-read task presented in the previous sections the following code will be outputted:

```
void Task1( void *pvParameters )
 {
    for( ;; )
    {
     //RUN1
     Request(RID);//BLOCKED
     //RUN2
     Read(RID, rid_var, nr);
     //RUN3
     //!!computation();
     }
 }
```

The generator creates a header file containing the declarations for all the tasks and a C source code file with their implementation. These resulting files can be compiled in a project along with the FreeRTOS source code. Before compilation, all that remains is to add the computational blocks containing the algorithms that manipulate the data. Because the task code is identical in both cases, the same code generator can be used for both the cooperative and preemptive models.

## 8. Conclusions

This paper presents a framework that can be used to model and verify real-time multitasking applications. The operating system, resources and application tasks are modeled by timed automata. This approach allows for the system's simulation and verification before the actual implementation, permitting the early detections of any undesirable behavior. Because the method is susceptible to state space explosion, the model must be abstract as much as possible, making a compromise between model complexity and its state space size.

The unified resource access interface and the code generator make possible the automatic generation of the modeled (and verified) application's source code, avoiding most of the error-prone human coding.

## 9. References

[1] Culler, D. E., Hill, J., Buonadonna, P., Szewczyk, R., Woo, A: "A Network-Centric Approach to Embedded Software for Tiny Devices", *EMSOFT 2001:First International Workshop on Embedded Software*, Oct. 2001. p. 114-130.

[2] Liu, J.W.S.: *Real-time systems*, Prentice-Hall, Inc., Upper Saddle River, New Jersey 2000.

[3] Waszniowski, L., Hanzálek, Z.: "Formal Verification of Multitasking Applications Based on Timed Automata Model", *Real-Time Systems*, Volume 38, Number 1, Springer-Verlag, 2008, p. 39-65.

[4] Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P., Skou, A.: "Testing Real-Time Systems Using UPPAAL", *Formal Methods and Testing,* Springer-Verlag, 2008, p. 77-117.

[5] Fersman, E.: *A Generic Approach to Schedulability Analysis of Real-Time Systems*, Ph.D. Thesis, Faculty of Science and Technology, Uppsala University, November 2003.

[6] Li, P., Ravindran, B., Suhaib, S., Feizabadi, S.: "A Formally Verified Application-Level Framework for Real-Time Scheduling on POSIX Real-Time Operating Systems", *IEEE Trans. Software Eng*. 30(9), 2004, p. 613-629.

[7] Zaharia T., Haller P.: *Formal Verification and Implementation of Real Time Operating System Based Applications*, 4th IEEE International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania, Aug. 2008, p. 299-302

[8] UPPAAL – tool box for modeling, validation and verification of real-time systems modeled as networks of timed automata; http://www.uppaal.com

[9] FreeRTOS – a portable, open source, mini Real Time Kernel; http://www.freertos.org

[10] Gerd Behrmann, Alexandre David, and Kim G. Larsen, *A Tutorial on Uppaal*, In proceedings of the 4th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM-RT'04). LNCS 3185, Springer-Verlag, 2004