Acta Universitatis Sapientiae
Electrical and Mechanical Engineering, 2 (2010) 51-62

Framework for Modeling, Verification and Implementation of Real-Time Applications

Liviu HAȚEGAN [1], Piroska HALLER [2]

[1]Technical University of Cluj-Napoca, Cluj-Napoca, România,
e-mail: liviu.hategan@yahoo.com,
[2]"Petru Maior" University of Tîrgu Mureş, Tîrgu Mureş, România,
e-mail: phaller@upm.ro

Manuscript received October 01, 2010; revised October 30, 2010.

Abstract: This paper proposes a framework for modeling, simulation and formal verification of embedded real-time applications running over a real-time multitasking kernel. We extend a simple real time kernel (RTOS) with synchronous and asynchronous message passing interface to communicate between tasks and drivers. In the same time some embedded system's specific drivers have been added, allowing unified resource access through these interfaces. The process engineer defines the control system as a set of tasks interacting with events occurring irregularly in time (alarms, user commands, communication) and regularly in time (sampled sensor data and actuator control signals). Taking into consideration both non-preemptive and preemptive scheduling, we propose two models consisting of networks of timed automata. Using a model-checker tool (UPPAAL), one can verify the timing and logical properties of an application, changing the time constrains and priorities. In a priority-based scheduling scheme, tasks interact both through the scheduler and through the mutual exclusion mechanism, but there are hidden from the engineer by the framework. The framework also offers a solution for generating the source code skeleton of the modeled application. This reduces the risk of errors do to error-prone human coding and most importantly ensures that the task will have the same behavior as described in the model.

Keywords: Formal verification, timed automata, real-time applications.

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 modern society and are rapidly evolving, growing in complexity. Moreover they are often used not only by themselves, but in clusters and networks. An

embedded real-time system is in close relationship with the physical environment it interacts with, it is involved in monitoring and control of complex physical processes. The applications running on such systems face a set of constrains like memory, processing power, energy consumption, but mainly timing constraints. Consequently, a real-time system must exhibit a predictable behavior, under a given set of conditions the designer must be able to know if the system will meet its requirements. Developing and running embedded applications with predictable and controllable behavior requires a real-time operating system (RTOS). This allows for an application to be constructed as a collection of tasks managed by the RTOS according to the scheduling policy, the timing requirements being mapped as task deadlines.

But the engineers that write embedded software are rarely computer scientists or experts in operating systems. It should be necessary to create an integrated framework for modeling, simulation, verification and code generation. On the other hand timeliness, concurrency, bounded response time, and heterogeneity need to be an integral part of the programming abstractions.

The timed automata formalism is widely used and well-proven in the description and verification of real-time systems. In [1] timed automata are proposed for the description of task arrival patterns. The authors present a unified model for finite control structures, concurrency, synchronization, and tasks with combinations of timing, precedence and resource constraints.

Another work [2] approaches the problem of formal modeling based on timed automata of a multitasking application running under a RTOS. The described model considers an operating system, the application tasks and the behavior of the controlled environment. In this approach the authors also model the internal structure, allowing for the verification of not only task schedulability, but other complex properties like safety and bounded liveness.

In [3] the authors present a framework for modeling and verification of real-time embedded applications running under a multitasking RTOS kernel. The authors propose a model of a minimal operating system defined as a network of timed-automata developed in order to use it as a framework for simulation and verification of mini real-time applications. The authors chose and analyzed the FreeRTOS [4] mini real-time kernel either in a non-preemptive (cooperative) or a preemptive configuration. Access to resources is done using a unified resource access interface.

In this paper we propose a framework for modeling, simulation and formal verification of embedded real-time applications and a solution for automatic source code generation of the modeled application. As in [3], the simulation and verification can be done using the UPPAAL [5] model checker.

We extended the model described in [3] taking into consideration in more detail the internal structure of the tasks [6], adding synchronous and

asynchronous message passing interface to communicate between tasks and drivers, allowing the model to be more expressive. Access to resources is done using a unified resource access interface, fact that simplifies the task structure and facilitates automatic source code generation based on the model. The existing drivers were extended with these interfaces and new drivers were added for the SAM7-EX256 development board specific devices. For the implementation we chose the SAM7-EX256 development board. We improved the unified resource access interface, now consisting of three functions: *Request()*, *Read()* and *Write()*. This allows for a simplified structure of the modeled tasks and is necessary for the purpose of translating the tasks from the model into the corresponding source code.

This paper is organized as follows: section 2 will describe the general properties and behavior for the task templates; in section 3 we describe the properties of the resources and the unified resource access interface; section 4 presents the cooperative model (tasks, cooperative scheduler, resources); the model constructed for the preemptive version of the FreeRTOS kernel is detailed in section 5; section 6 addresses the issue of simulation and formal verification of the applications; in section 7 we describe the process of automatically generating the source code of the application from the model and finally, in section 8 we state our conclusions.

2. The application tasks

Each task instance is modeled by a timed automaton that is synchronized using channels. In general, embedded tasks present the following behavior [1], [3]:

```
   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 are accessed by tasks through blocking request calls. The desired resource is explicitly specified through its RID (resource ID) [7]. After making a request call, the task enters in blocking state, where it waits for the resource to become available.

The FreeRTOS tasks are prioritized.

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 con-

sequence 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 [2].

3. The resources

In constructing the general resource model, we considered that 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 and in the request call the resource is explicitly specified through its resource ID. Every resource has a minimal inter-arrival time (the MIAT). A resource can unblock a waiting task and provide data at any time after its MIAT expires.

3.1 The unified resource 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.

As we mentioned in the previous sections, access to a particular resource is performed trough the unified access interface. Each resource has a state variable associated. The read and write operations are performed using the state variables (or any other user-defined data structure that corresponds). The interface is composed of the following functions:

- Request (RID)
- Read (RID, var_rid, nr)
- Write (RID, var_rid, nr)

When requesting a resource, it must be explicitly specified through its RID. If the resource is not available (is being held by another task) the calling task will block until the requested resource becomes available. The *Read()* and *Write()* are nonblocking calls, they will be performed after the access to the requested resource is granted and provide the means for read/write operations. Besides these resource RIDs, the function calls must also include the state variables associated with the resource (*var_rid*) and the number of bytes to be read or written. The functions implemented in the unified access module are also present in the model and permit a simple description of concurrent access to the desired resources and read/write operations.

4. The cooperative model

4.1 The cooperative application tasks

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. The task in pseudocode is:

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

The task automaton (*Fig. 1*) is synchronized with the models of the scheduler and resources via channels.

The RUN locations are characterized by a best-case and worst-case execution time (WCET, BCET). This is modeled by the location's invariant (*y<=W*), and the guard on the outgoing transition (*y>=B*). *REQ_BCET[RID]*, *REQ_WCET[RID]*, *READ_BCET[RID]* and *READ_WCET[RID]* are constants defined in the model and they are the BCET and WCET for requesting and reading data in the case of the resource represented by *RID*. While being in a running state, a task can be interrupted by a resource's interrupt service routine (ISR). The execution time of the ISR is added to the *W* and *B* variables.

BLOCKED: the state is entered when the task requests a particular resource through the *request[pid][RID]!* channel. The state is left when the resource becomes available (the *event_or_timer[RID]?* channel is activated).

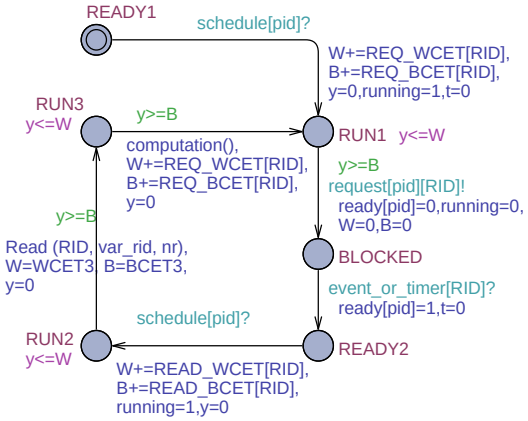

Figure 1: Simple cooperative task model.

If there is no task running or ready, the FreeRTOS kernel schedules the Idle Task (*Fig. 2*), which is always available for scheduling. Following the general task form, the Idle Task periodically requests the *NULL* resource, yielding processor control.

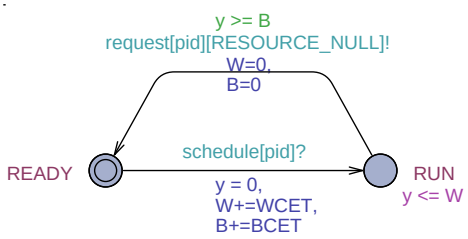

Figure 2: The Idle Task model.

4.2 The cooperative scheduler

In the cooperative behavior the running task has 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 (the *Request(RID)* function).

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

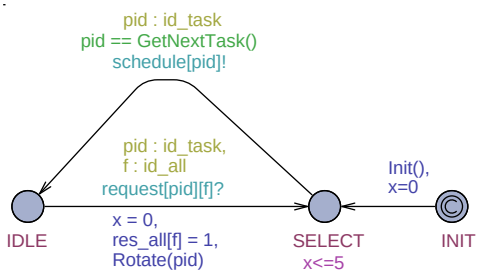

Figure 3: The cooperative scheduler model.

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.

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, which is hardware-dependant.

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

4.3 The system resources

Fig. 4 illustrates the general resource model.

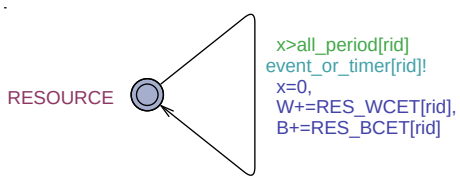

Figure 4: Resource model.

The MIAT values for all of the system's resources are stored in the array *all_period[NR_RESOURCES]*. The MIAT is modeled by the guard *x>all_period[rid]*. The waiting task is unblocked via the *event_or_timer[rid]!* channel. The *RES_WCET[pid]* and *RES_BCET[pid]* constants are used to delay 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 (*Fig. 5*). The timer unblocks a waiting task when the *x* clock has the same value as *all_period[tid]*. The constant *tid* represents the timer's RID.

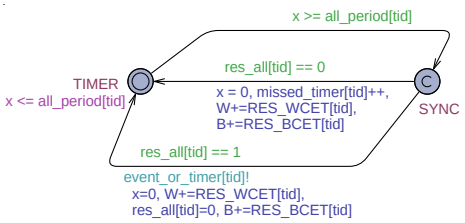

Figure 5: Timer model.

The initial state TIMER is left when the predefined period expires. The SYNC state is committed so it is left immediately, the automaton unblocking any waiting task. In order to avoid system deadlock, the timer is allowed to expire even if none of the tasks are blocked waiting for it.

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 (via the *suspend[pid]?* channel). The invariant *y'==0* will stop the *y* clock while the automaton is in a suspended location. Upon rescheduling, the clock is restarted (the *y'==1* expression). *Fig. 6* presents the preemptive version of the simple read-request-computation task model described in section 4.1.

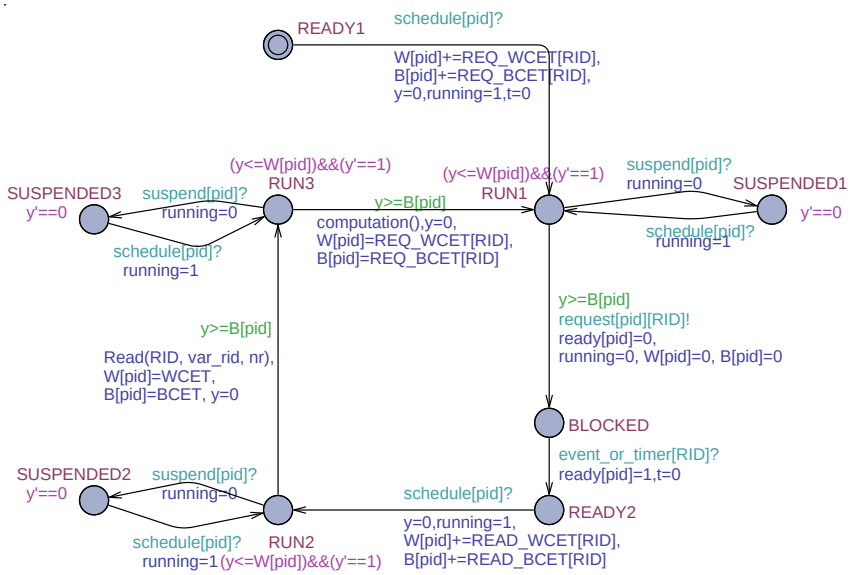

Figure 6: Simple preemptive task model.

The Idle Task is the same as in the case of the cooperative model, except for 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

The preemptive scheduler model is illustrated in *Fig. 7*.

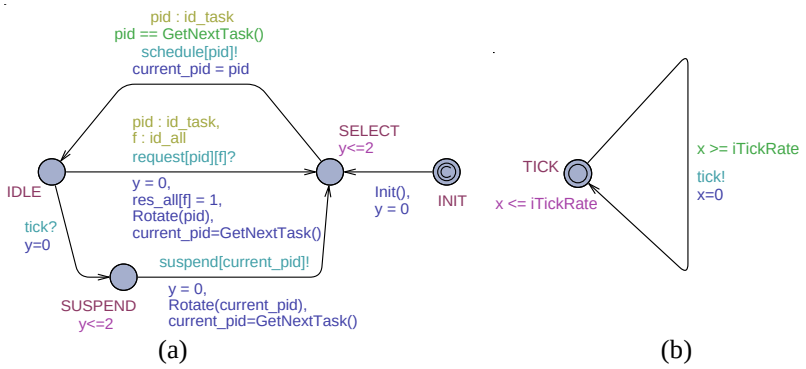

Figure 7: The preemptive scheduler model (a) and the tick interrupt model (b).

The preemptive scheduler can periodically perform 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. Each time the interrupt occurs, the kernel determines if a context switch must take place. This action is performed in the SUSPEND state.

6. Simulation and formal verification

Simulation and formal verification can be performed using the UPPAAL [8] integrated simulation and verification tools. For formal verification, the properties required for an application to function according to requirements must be expressed in UPPAAL's CTL subset [9]. The verifiable properties are: reachability, safety, liveness, bounded liveness and deadlock-freeness.

Reachability properties are checked to see if it is possible to reach a state where a formula *p* is satisfied, for example:

- E<> Task1.RUN1 - checks if *Task1* can ever reach the *RUN1* state;
- E<> Task1.SUSPENDED1 - verifies if there is a possibility that *Task1* will ever be suspended, a context switch taking place.

Safety properties require that a formula *p* is satisfied in all reachable states or, that there is a path in which *p* is 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 where *Task1* will not spend more than 500 time units in the *READY2* state.

Liveness properties require that, in all cases, the system will eventually reach a state where a formula *p* is true. Another form is that if a formula *p* is true, another formula *q* will become true eventually:

- A<> Task1.RUN1 - *Task1* will inevitably be in the *RUN1* state at some point;
- Timer(1).SYNC --> Task5.RUN2 - considering a blocked task waiting for a timer, if *Timer(1)* expires then *Task5* will inevitably be scheduled.

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

- Timer(1).SYNC --> (Task5.RUN2 and Task5.t<=200) - when *Timer(1)* expires *Task5* will be scheduled within 200 time units.

Deadlock-freeness [7]: A[] not deadlock.

To verify the time constrains, the execution time of the elements regarding our implementation for the SAM7-EX256 development board (task scheduling time, resource ISR execution time, resource access and read/write operations time, etc.) were measured using the system's physical timers and were introduced in the models.

The interrupt service routine execution time:

- Analog/digital converter: RES_BCET=RES_WCET=4,8 μs;
- Key pressed: RES_BCET=5,2 μs; RES_WCET=10,2 μs;
- Joystick: RES_BCET=5,5 μs; RES_WCET=10,4 μs;
- Timer0 ÷ Timer2: RES_BCET=RES_WCET=2.5 μs.

Driver unified resource access:

- Analog/digital converter: RES_BCET=RES_WCET=26,5 μs;
- Display: RES_BCET=RES_WCET=30 μs;
- External storage: RES_BCET=RES_WCET=30 μs;
- Keys: RES_BCET=RES_WCET=15,4 μs;
- Joystick: RES_BCET=RES_WCET=15,4 μs.

Write operation:

- Display: RES_BCET=390 μs (1 char.); RES_WCET≈7 ms (20 char.);
- External storage: RES_BCET=54 μs (1 char.) ;RES_WCET ≈100 μs(128 char.).

Read operation:

- Analog/digital converter: RES_BCET=RES_WCET=26,5 μs;
- Keys: RES_BCET=27,2 μs; RES_WCET=28 μs;
- Joystick: RES_BCET=27,2 μs; RES_WCET=28 μs.

7. Source code generation

We have developed a code generator application that, based on the model presented as input (XML file), recognize the states and functions calls of each task and output the corresponding source code skeleton. In the case of the simple request-read-computation task presented in the previous sections the following code is generated:

```
void Task1( void *pvParameters )
 {
        for( ;; )
        {
            Request(RID);
            Read(RID, var_rid, nr);
            //!!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. The task code is identical in both cases, cooperative and preemptive.

Alongside the models and source code generator, we constructed a source code project that contains the FreeRTOS source, drivers including interrupt mechanisms for the system's resources and the module for unified peripheral access.

The source code project also includes a special task that can be used to directly measure the time necessary for a sequence of code to execute on this physical system (for use in the model). Also, we included functions to facilitate the conversion of data from the type specific to a particular resource to another resource's type. For example, to convert and copy the data from the state variable associated with the system's analog-to-digital converter to the state variable of the LCD display, one can use the function *ADCtoLCD(res_adc, res_lcd)*. The available functions and they execution time is:

- INTtoLCD(Integer, res_lcd); RES_BCET=7,8 μs; RES_WCET=44,1 μs;
- INTtoSD(Integer, res_sd); RES_BCET=66,5 μs; RES_WCET=305 μs;
- ADCtoSD(res_adc, res_sd); RES_BCET=45,3 μs; RES_WCET=333 μs;
- ADCgetINT(res_adc); RES_BCET=RES_WCET=31,4 μs;
- ADCgetSTR(res_adc); RES_BCET=17,5 μs; RES_WCET=350 μs;
- LCDgetSTR(res_lcd); RES_BCET=RES_WCET=18 μs;
- SDgetSTR(res_sd); RES_BCET=RES_WCET=47,2 μs.

These functions are also present in the model, allowing it to be more expressive and further simplifying the code of application tasks.

8. Conclusions

This paper presents a framework that can be used to model, verify and implement 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. 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. 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.

References

[1] Fersman, E., "A generic approach to schedulability analysis of real-time systems", *Ph.D. Thesis*, Faculty of Science and Technology, Uppsala University, November 2003.

[2] Waszniowski, L., and Hanzalek, Z., "Formal verification of multitasking applications based on timed automata model", *Real-Time Systems*, vol. 38, no. 1, Springer-Verlag, pp. 39-65, 2008.

[3] Zaharia, T., and Haller, P., "Formal verification and implementation of real time operating system based applications", *in Proc. of the 4th IEEE International Conference on Intelligent Computer Communication and Processing, Cluj-Napoca, Romania*, pp. 299-302, 2008.

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

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

[6] Liu, J.W., "Real-time systems", Prentice-Hall, Inc., Upper Saddle River, New Jersey 2000.

[7] Li, P., Ravindran, B., Suhaib, S., and Feizabadi, S., "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems", *IEEE Trans. Software Eng.* vol. 9, no. 30, pp. 613-629, 2004.

[8] Hessel, A., Larsen, K. G., Mikucionis, M., Nielsen, B., Pettersson, P., and Skou, A., "Testing real-time systems using UPPAAL", *Formal Methods and Testing*, Springer-Verlag, pp. 77-117, 2008.

[9] Behrmann, G., David, A., and Larsen, K. G., "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.