Timing Analysis of Real-Time Embedded Systems using Model Checking

Vallabh R. Anwikar, Purandar Bhaduri

Timing Analysis of Real-Time Embedded Systems using Model Checking

Vallabh R. Anwikar and Purandar Bhaduri
Indian Institute of Technology Guwahati
Guwahati, India 781039
{anwikar,pbhaduri}@iitg.ernet.in

Abstract

Modern real-time embedded systems are highly complex and distributed. Timing analysis of these systems is a challenging task. Model checking is increasingly being used for analyzing such systems. In this paper, we use timed automata based model checking for the timing analysis of distributed embedded systems with fixed priority preemptive tasks which exchange messages via communication buses with specific access protocols. We have constructed a general task model in UPPAAL *for preemptable tasks based on the preemption handling method proposed by Waszniowski* et al. *We present two case studies, one involving an advanced automotive control application using the FlexRay bus, and the other using a Controller Area Network (CAN) bus. We also present a case study showing how the explicit-time model checker* SPIN *can be used for computing the end-to-end latency between tasks and how it compares with the implicit-time handling methods used by timed automata based model checkers.*

1. Introduction

Real-time embedded systems are increasingly permeating all aspects of our life. Even a small car has dozens of processors communicating with each other to control and monitor the functionality of the vehicle. The constraints imposed by application requirements in the automotive and avionics domains very often lead to complex and distributed real-time systems. These systems are heterogeneous in terms of processor architectures and scheduling schemes, and involve resource sharing and complex dependencies between tasks. This makes it very difficult to analyze such systems for timing and other performance properties. Existing analytic methods used for analyzing such systems can be overly pessimistic in terms of results, leading to higher costs.

An approach to multiprocessor scheduling using fixed priority preemptive tasks and TDMA (or fixed-priority non-preemptive) messages, called *holistic scheduling* was pioneered by Tindel *et al* [26, 25]. These techniques extended the classical theory of uniprocessor scheduling [16] by combining processor and bus scheduling. They use an iterative fixed point computation to determine worst case latencies, with static offsets for tasks for modelling communication delays. This holistic scheduling analysis was extended by Palencia and Harbour [21] allowing both static as well as dynamic offsets to give more accurate results.

Another analytical approach to the timing analysis of distributed embedded systems is offered by the framework of *Real-Time Calculus* [24, 5]. It models the arrival pattern of tasks and the service offered by the resources to the tasks, along with the scheduling policy. It can be used to derive hard upper and lower bounds of various performance criteria such as maximum end-to-end delay experienced by an event stream or buffer requirements.

Perathoner *et al* [22] compare the influence of different system abstractions on the performance analysis of distributed real-time embedded systems. They have considered various activation patterns and dependencies as benchmarks. According to their study, timed automata [2] based model checking is the only technique which gives accurate results across all benchmarks. This is because the other techniques involve abstractions that ignore the presence of correlations between task activations and data dependencies, in the process computing the end-to-end delay of a task chain as the sum of the local worst case response times. Model checking on the other hand explores actual execution paths and is able to come up with precise figures for end-to-end timing. This has motivated us to perform analysis using timed automata on large heterogeneous and multiprocessor embedded systems.

Although timed automata perform well across all benchmarks there are serious issues about their scalability. The size of the state space is exponential in the number of clocks, the largest constant in the system and the number of concurrent timed automata. The complexity caused by the largest constant can be tackled by using symbolic methods but the number of concurrent tasks and clocks depends entirely on the application in hand. This makes timed automata difficult to apply for real-world scheduling problems.

The clock variables in a timed automaton model cannot be stopped; this makes handling of preemptable tasks using timed automata impossible. Modeling such tasks correctly can only be done by using *stopwatch automata*, au-

tomata with clocks that can be stopped and restarted. Unfortunately, reachability analysis of stopwatch automata is an undecidable problem [13]. However, there are certain methods [28, 18] by which we can model preemptable tasks in timed automata with over-approximation. More recently, UPPAAL 4.1 has made modelling with stop-watches possible with the the support of an efficient, zone-based over-approximate state-space exploration [7].

Related Work The DREAM tool [8] implements a conservative approximation method for the verification of distributed real-time embedded systems with preemptive tasks by timed automata [18]. To model preemptions it uses discrete checkpoints at which task interruptions are allowed. This is a discretized approximation of stopwatches, which gives an overapproximate result.

In an alternative method proposed by Waszniowski [28], the clock value at the time of preemption is abstracted by the nearest lower and upper integer value to provide the over-approximation. The DREAM tool has a generalized task model for preemptable task which makes it very easy to use. The lack of such a generalized model makes the use of Waszniowski's method difficult. Although the method in the DREAM tool was tried on quite a few large systems, to the best of our knowledge, the method proposed by Waszniowski *et al* was never used in such a generic way and on such large examples.

In this paper we propose a generalized model for preemptable tasks based on the preemption handling method of Waszniowski *et al*, so that it can be easily used for modeling distributed asynchronous systems. We provide case studies of two automotive control systems - one using a FlexRay bus [12] and the other, a Controller Area Network bus [20]; both involve preemptive scheduling. We also compare how the above preemption handling methods perform in terms of time required for verification.

We also experiment with a general purpose (*i.e.*, untimed) model checker using an *explicit-time method* for modeling timing properties and compare its results with UPPAAL. This is done by adapting an example from Mohalik *et al* [19]. In the explicit-time method, we model time by using an integer variable which is incremented or decremented to signify the passage of time. This explicit-time handling method was first proposed by Lamport [14].

While preparing the final draft of this paper we came to know of the work by Rajeev *et al* [23] which has goals similar to ours. This work uses a formalism called Calendar Automata [9] which represents time by maintaining timestamps of events, and increments time by the maximum possible value so that no event in the calendar is missed. This potentially reduces the state space, as intermediate time points are not considered. Since time is considered to be discrete, and tasks have finite periods, offsets and execution times, the authors show that the state space is actually finite, allowing the use of a general purpose model checker. Based on several case studies, the authors claim that the technique is more scalable than existing formal methods based timing analysis techniques, while allowing more accurate results. It would be interesting to compare their method with ours on actual case studies.

To summarize, the contribution of this paper is to propose a generalized timed automata based representation of preemptable tasks based on Waszniowski's method. By using the generalized task model, we can very easily express larger examples that were difficult to express earlier. Moreover, using this generalized model we have successfully analyzed the timing properties of systems which were earlier found to be too big for model checking. We have also shown how the explicit time approach can be used for timing analysis, and presented some data about its time and memory requirements on case studies.

The rest of the paper is organized as follows. In Section 2 we explain the basics of timed automata and the UPPAAL model checker. Section 3 explains the preemptable task model we have proposed based on the preemption handling method of Waszniowski *et al*. In Section 4, three case studies are presented using the proposed task model, along with experimental results. In Section 5, we describe the explicit-time method. This is followed by case studies along with experimental results in Section 6. Finally, in Section 7, we present concluding remarks.

2. Preliminaries

2.1 Timed Automaton

A timed automaton is a finite automaton which is extended with clock variables. Clocks are assumed to proceed with uniform speed. Clock variable can be tested and reset to zero.

For the set C of real valued clocks with $x, y \in C$, the set $\psi(C)$ of *clock constraints* over C is given by:

$$\alpha ::= x \sim c \mid x - y \sim c \mid \neg\alpha \mid \alpha \wedge \alpha$$

where $c \in \mathbb{N}$ and $\sim \in (\leq, \geq, <, >, =)$.

A *timed automaton* A over the set of actions Act, the set of atomic propositions AP and the set of clocks C is a 5-tuple (L, l_0, E, I, V) where:

- L is a finite set of *locations*;
- l_0 is the *initial location*;
- $E \subseteq L \times \psi(C) \times Act \times 2^C \times L$ is a set of *edges*. If $(l, g, a, r, l') \in E$ we write $l \xrightarrow{g,a,r} l'$, which represents a transition from the location l to the location l' with clock constraint g (also called the enabling condition of the transition or *guard*), action a to be performed (providing synchronization of concurrent automata) and the set of clocks r to be *reset*;
- $I : L \longrightarrow \psi(C)$ is a function, which for each location assigns a clock constraint (also called the *invariant* condition of the location); and

- $V : L \longrightarrow 2^{AP}$ is an *observation* function, which for each location assigns a set of atomic propositions that must hold in the location.

UPPAAL [3, 15] is a symbolic tool for simulation and automatic verification of real-time systems modeled as networks of timed automata extended with integer variables. Models in the system consists of non-deterministic processes with finite control structures and real valued clock variables communicating through synchronization and shared variables. Every component in the system is represented by a timed automaton which describes its behavior.

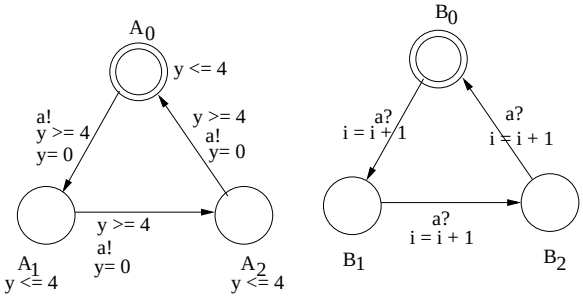

Figure 1. Example of Timed Automaton in UPPAAL

Consider the UPPAAL model shown in Figure 1. The model consists of two components with three control locations each. The component *A* has one clock variable *y* and a synchronization channel *a*, which is shared with the component *B*. The component *B* has an integer variable *i*. Generally a transition has three labels: a *guard* involving integer variables and clock values; a *reset* action which assigns new values to variables and resets clock variables to zero, and a *synchronization action* with another component. The notation *a!* denotes the sending of an event and *a?* the receiving of the event. Every location can also have an *invariant*. The invariant represents the condition which has to be satisfied within that particular location. If the invariant does not hold, control cannot stay in that location and a transition must be taken.

In the model in Figure 1, control can stay in location A_0 until the invariant $y \leq 4$ is true, and the transition to A_1 is enabled as soon as the guard $y \geq 4$ becomes true. When the transition is taken, the clock variable y is reset and the synchronization action $a!$ is taken, which forces the component *B* to change its location from B_0 to B_1 performing an update on the value of i.

3. Preemptable Task Model

Modeling preemption accurately using timed automata is not possible as a clock variable measuring the execution time of a task cannot be stopped when the task is preempted. However, there are some methods [28, 18] by which we can model preemption with slight over-approximation. In this section we explain the proposed generalized task model, which is based on the method of Waszniowski *et al* [28] for modeling preemption.

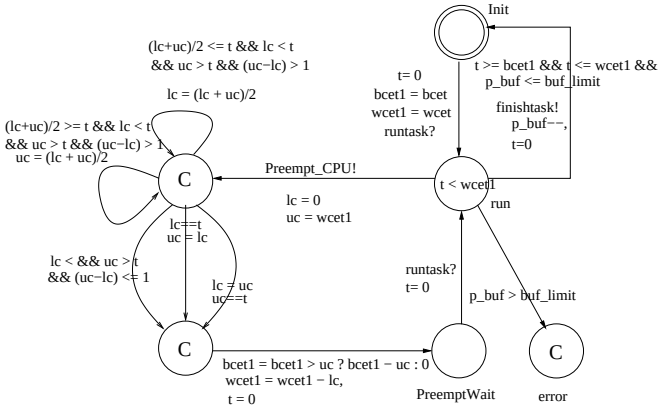

Figure 2. Preemptable Task Model

The basic idea about preemption handling is that only one task can run on a processor at a time, so whenever a low priority task is preempted by a higher priority task the remaining execution time of the preempted task is stored in a variable. Whenever the task is rescheduled it will only execute for the remaining amount of time.

The only problem in doing this is, a clock variable is a real variable and if we want to store the remaining execution time of the preempted task, we have to store it in an integer. To overcome this problem, the value of a clock variable *c* measuring the time for which the task has executed is bounded by the nearest lower and upper integers. The lower and upper bounds are computed by a simple bisection algorithm shown in Figure 2. Consider for example a task whose execution time is bounded by the range *(BCET,WCET)*. Let *uc* and *lc* be the nearest upper and lower integers of the value of clock *c* at the time of preemption; then the remaining execution time of the task is the interval bounded by $BCET_{new} = max(0, BCET - uc)$ and $WCET_{new} = WCET - lc$.

Every preemptable task is represented by a timed automaton shown in Figure 2. A task starts in the *Init* location. The task starts execution whenever it receives the *runtask* signal from the scheduler which is modeled by another automaton. The clock variable *t* is reset and it start measuring the execution time of the task. The task can stay in *run* location for its *wcet* duration. Whenever the task finishes its execution it goes back to *Init* location, which is modeled by transition from *run* to *Init*. The guard on this transition represents the condition for the task finishing its execution. If any higher priority task is enabled when a lower priority task is executing, the scheduler will schedule the higher priority task. The lower priority task is preempted and it moves to the *PreemptWait* location, via two intermediate *committed* locations. The committed locations, represented by a mark *C* in the model, ensure that the calculation of the remaining execution time of the task is done atomically and in zero time.

The bisection algorithm is implemented and new values of *BCET* and *WCET* are calculated in the sequence of transitions from the *run* to the *PreemptWait* location. Whenever the higher priority task finishes its execution, the lower priority task again starts execution by synchronizing with the *runtask* channel. Till then it waits in the *PreemptWait* location.

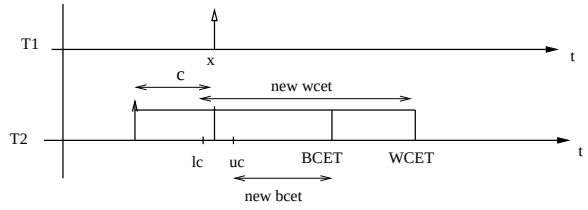

Figure 3. Example: Preemption and Updating of *BCET* and *WCET*

As shown in Figure 3 , the task T_2 is preempted by the higher priority task T_1 at x. The task T_2 will update its new BCET and WCET as shown in Figure 3 and wait in the *PreemptWait* location waiting to be scheduled again by the scheduler.

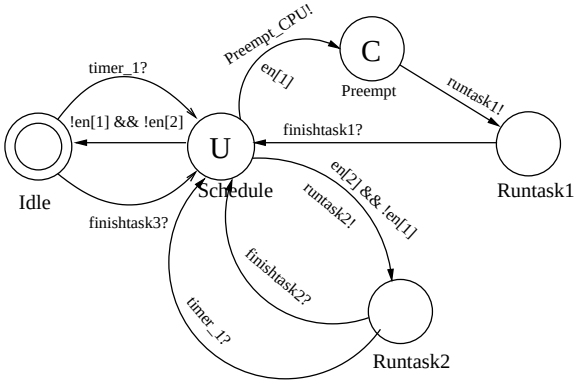

Figure 4. Scheduler Automaton

The scheduler automaton for two tasks *Task1* and *Task2* is shown in Figure 4. Control starts in the *Idle* location, and whenever a task is released, it moves to the *Schedule* location. A task can be released either by the expiry of a timer or by the completion of some previous task in the task chain. In this example *Task1* is released by the expiry of the timer *timer_1* while *Task2* is released asynchronously by the completion of *Task3*.

The decision about which task to execute is made in the *Schedule* location. In the above figure, the guard *en[1]* indicates that *Task1* is enabled and it will start its execution irrespective of the status of *Task2*. But *Task2* will start its execution only when Task1 is not enabled, thus according a higher priority to *Task1*.

The scheduler sends a *Preempt_CPU* signal before scheduling a higher priority task so that the lower priority task currently executing will be able to store the bounds for its remaining execution time before yielding the processor to the higher priority task (see Figure 2). While any task is executing, the scheduler automaton remains in its corresponding *Runtask* location. If a higher priority task is enabled for execution when a lower priority task is executing, the control will move from corresponding *Runtask* location to *Schedule* location so that we can schedule the higher priority task ready at that moment. In Figure 4, this is shown by the transition from the *Runtask2* to the *Schedule* location on receiving the signal *timer_1*. A task announces its completion by sending a *finishtask* signal. The scheduler automaton moves back to the *Schedule* location after receiving this signal. If no other task is enabled, the scheduler will go back to the *Idle* location. The *Schedule* location is shown to be *urgent* representing the fact that all the scheduling decisions are made instantaneously. Note that all the signal used for communication are declared as broadcast signals in UPPAAL.

4. Case Study Using UPPAAL

4.1. Automobile Control Application

We present a case study of a set of advanced automotive control applications presented in [12]. The system contains three threads of control: adaptive cruise control (ACC), electric power steering (EPS) and traction control (TC). Data is collected from sensors and processed in one of the ECUs. The ECUs then respond to the sensed environment through actuators. ACC is responsible for maintaining safe distance between two cars while TC helps during slippery road condition by providing efficient traction. EPS is responsible for assistance in steering. There are a total of ten sensors, three ECUs and four actuators. Figure 5 presents the system as a set of task graphs. Table 1 shows the worst case transmission time (WCTT) of messages. The schedule of every thread is shown in the Tables 2, 3 and 4.

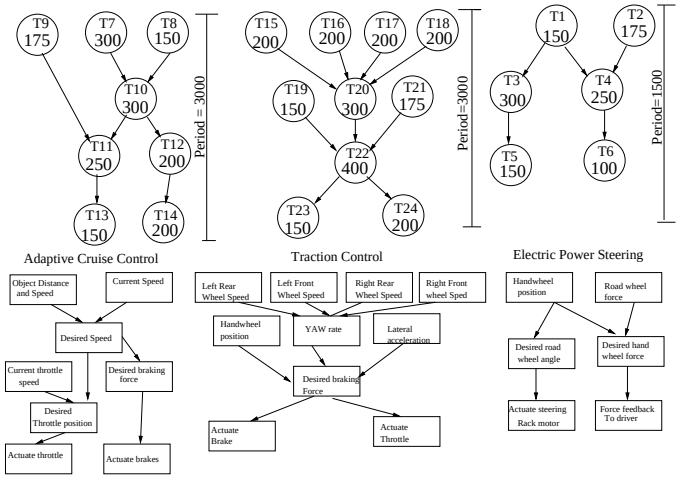

Figure 5. Task graph of system

Message	WCTT	Message	WCTT
T_1 T_3	48	T_{10} T_{11}	48
T_1 T_4	48	T_{10} T_{12}	48
T_2 T_4	48	T_{11} T_{13}	40
T_3 T_5	80	T_{12} T_{14}	40
T_4 T_6	48	T_{15} T_{20}	48
T_7 T_{10}	48	T_{16} T_{20}	48
T_8 T_{10}	48	T_{17} T_{20}	48
T_9 T_{11}	40	T_{18} T_{20}	48
T_{19} T_{22}	40	T_{20} T_{22}	88
T_{21} T_{22}	80	T_{22} T_{23}	48
T_{22} T_{24}	48		

Table 1. Worst case transmission time of messages

Tasks T_4,T_{11} are mapped onto ECU1, tasks T_3,T_{12},T_{22} are mapped on ECU2 and T_{20},T_{10} are mapped on ECU3. Fixed priority preemptive scheduling is used on these ECUs and priorities are assigned according to period. We assume that time triggered scheduling is used for messages, which are triggered at the end of task execution. Further, four FlexRay buses with static slot scheduling are used for the communication. The detailed bus description is shown in Figure 6.

Task	Release	Deadline
T_7	0	800
T_8	0	800
T_9	0	800
T_{10}	800	1600
T_{11}	1600	2350
T_{12}	1600	2300
T_{13}	2350	3000
T_{14}	2300	3000

Table 2. Schedule of Adaptive Cruise Control

Task	Release	Deadline
T_{15}	0	675
T_{16}	0	675
T_{17}	0	675
T_{18}	0	675
T_{19}	0	1450
T_{20}	675	2300
T_{21}	0	1450
T_{22}	1450	2450
T_{23}	2450	3000
T_{24}	2450	3000

Table 3. Schedule for Traction Control

Preemptable tasks in the system are represented by the automata which we have presented in the Section 3. Static slot scheduling is modeled by a slot monitor automaton

Task	Release	Deadline
T_1	0	450
T_2	0	533
T_3	450	1050
T_4	533	1200
T_5	1050	1500
T_6	1200	2500

Table 4. Schedule for Electronic Power Steering

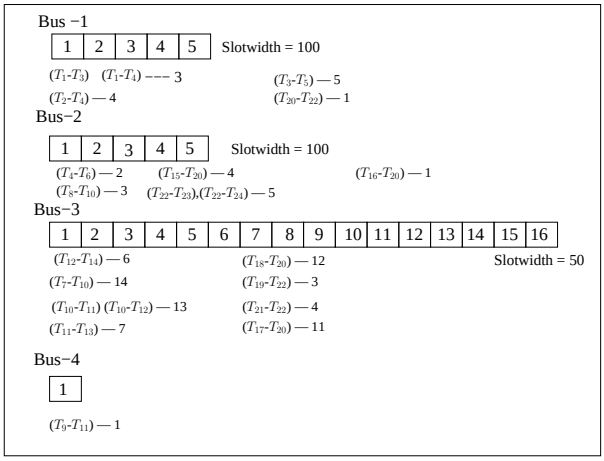

Figure 6. Static slot allocation on buses

which increments the slot number after the corresponding slot width delay. When scheduling a message an extra condition is added to check that every message is sent in its fixed slot. Time triggering of the tasks can be easily modeled by a timer automaton which will take transition at that particular time instant. In every task (or message) automaton two clock variables are used, one modeling its execution time and the other modeling the time since its activation. The time after activation can be used for checking the deadline of the task. Whenever a task or message crosses its deadline, the automaton goes into a committed location with no outgoing transition, modeling a deadlock.

The worst case response times of the tasks and messages are checked by checking the reachability of the *error* location of the automaton. This property can be specified by CTL as AG *!(Task.error)*. Table 5 and Table 6 shows the worst case response time of messages and tasks which are scheduled on three ECUs respectively.

4.2. Controller Area Network Bus

The end-to-end latency of messages is an important design parameter that needs to be within certain specified bounds for correct functioning of real-time systems. The time taken by a message through a task chain, starting from a sensor and ending in an actuator, is called its *end-to-end latency*. We have used timed automata model checking for verifying end-to-end latency in this example.

The system we analyze is shown in Figure 7. It is adapted from [20] and consists of three ECUs, one CAN

Message	WCRT	Message	WCRT
$T_1\ T_3$	98	$T_{10}\ T_{11}$	373
$T_1\ T_4$	98	$T_{10}\ T_{12}$	373
$T_2\ T_4$	173	$T_{11}\ T_{13}$	90
$T_3\ T_5$	430	$T_{12}\ T_{14}$	40
$T_4\ T_6$	365	$T_{15}\ T_{20}$	148
$T_7\ T_{10}$	448	$T_{16}\ T_{20}$	348
$T_8\ T_{10}$	98	$T_{17}\ T_{20}$	398
$T_9\ T_{11}$	255	$T_{18}\ T_{20}$	448
$T_{19}\ T_{22}$	40	$T_{20}\ T_{22}$	313
$T_{21}\ T_{22}$	105	$T_{22}\ T23$	98
$T_{22}\ T_{24}$	98		

Table 5. Worst case response time of messages

Task	Deadline
T_3	300
T_4	250
T_{10}	600
T_{11}	300
T_{12}	200
T_{20}	300
T_{22}	900

Table 6. Worst case response time of tasks scheduled on ECUs

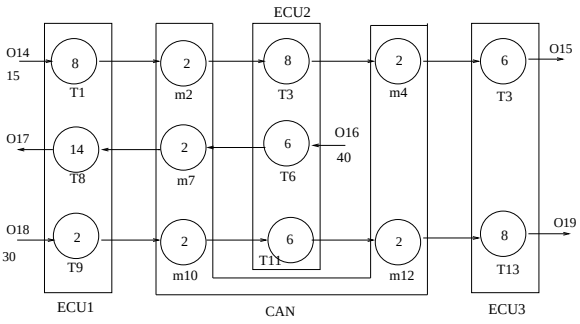

Figure 7. Example network of ECUs communicating through a CAN bus.

bus, eight tasks (T_i) mapped on different ECUs and five messages (m_i) mapped on a CAN bus as shown. Tasks mapped on ECUs execute according to fixed priority preemptive scheduling, whereas messages are scheduled on the CAN bus according to fixed priority non-preemptive scheduling. If task T_i and T_j are scheduled on the same ECU then T_i has higher priority than T_j if $i < j$. Similarly for the messages, a lower index shows higher priority.

Three computation paths are defined, $O_{14}-O_{15}$, $O_{16}-O_{17}$ and $O_{18}-O_{19}$. The objects follow event based activation, *i.e.*, T_1 is activated periodically every 15 time units, its completion will trigger message m_2 and so on. We have analyzed the worst case latency of these chains. Previous analysis of this system was carried out using holistic scheduling [20] and real time calculus [6]. But as mentioned in [22] these methods gives pessimistic results when it comes to complex activation pattern of tasks and dependencies between tasks.

There are multiple active chains present in the system. To model this, an array of clocks is used. Every time a new chain is activated, a new clock is activated and it measures the latency of that particular chain. Whenever that particular chain finishes its execution, the value of clock is checked against its deadline.

Table 7 shows a comparison of our results with those obtained by using Real-Time Calculus. We can see that the timed automata method shows more accurate results. As noted earlier, this is due to the fact that model checking explores each and every computation by searching the entire state space.

Chain	UPPAAL	Real-Time Calculus
$O_{14}-O_{15}$	28	32
$O_{16}-O_{17}$	50	60
$O_{18}-O_{19}$	110	210

Table 7. Worst case latencies of three task chains

We attempted to perform the timing analysis of this case study using the approach proposed in the DREAM tool with some customizations. But the verification encountered the state explosion problem. In order to reduce the complexity of the system, we modeled the buffers with variables instead of *buffer automaton* proposed in the DREAM tool. After all these changes, we were able to verify the system.

4.3. Real-Time CORBA Application

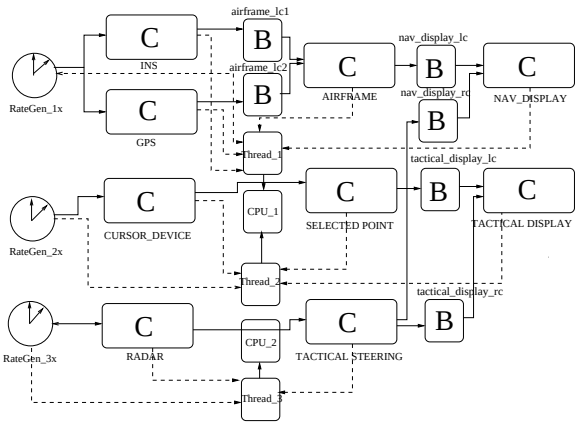

Figure 8. Real-Time CORBA Application.

The case study shown in Figure 8 is adapted from [17]. The input data is shown in Tables 8 and 9. *BCET* and *WCET* stand for best and worst case execution time of a task, *DL* is its deadline while *SP* is its sub-priority.

BCDelay and *WCDelay* are best and worst case channel latencies. The tasks in the systems are represented with rectangular boxes and marked with a *C* while the buffers are marked with a *B*. In Figure 8, the dependencies between tasks are shown by solid lines while the mapping of tasks on threads is shown by dotted lines. Threads are in turn scheduled on processors. The scheduling strategy between threads is preemptive. The system has nine tasks, six buffers, three timers and two processors.

To carry out a comparison between the DREAM tool [8] and our method, we replaced the preemptable task model used in DREAM with the model discussed in Section 3. We kept all other components as specified by the DREAM tool and checked whether the system is schedulable by checking the property *A[] not deadlock*. We were able to verify the schedulability of the system in 1.31 seconds while the DREAM tool model required 5.11 seconds. We are looking into bigger case studies to analyze in more detail the relative speeds of these two methods.

Task	BCET	WCET	DL	SP
Ins	32	32	170	1
Gps	29	29	170	1
Airframe	80	80	246	2
Nav_Display	19	19	532	3
Cursor_Device	18	18	65	1
Selected_Point	24	24	25	2
Tactical_Display	21	21	80	3
Radar	12	12	13	1
Tactical_Steering	16	16	17	2

Table 8. System Description

Channel	WCDelay	BCDelay	BufferSize
airframe_lc1	0	0	2
airframe_lc2	0	0	2
tactical_display lc	0	0	2
nav_display_lc	0	0	3
tactical_display rc	2	2	2
nav_display_rc	2	2	6

Table 9. System Description

5. Explicit-Time Description Method

Timed systems can be modeled in two ways. The first, known as the *implicit-time* approach, uses special logics and languages developed for modeling time. UPPAAL is an example model checker which models time implicitly using clock variables. There is an alternative method for handling time, known as the *explicit-time* handling method. In this method, an ordinary variable, designated as a *timer*, is incremented (or sometimes decremented) to model the elapsing of time. With the explicit-time method, we can use any model checker (*i.e.*, without clock variable) for verifying timed systems. Lamport has proposed an explicit-time description method [14] using a clock-ticking process *tick* to simulate the passage of time. In an earlier work, Abadi and Lamport [1] showed that the explicit-time approach works fine for specifying and verifying properties of many real-time algorithms. Recently, the explicit-time method was successfully applied by Van den Berg *et al* [27] to verify the safety of railway interlockings for one of Australia's largest railway companies.

In the explicit-time method, the timing bounds on actions can be specified in three ways, via the use of a *countdown timer*, *countup timer* or *expiration timer*. A countdown timer decrements the value of a timer and an action is triggered when its value reaches zero. A countup timer increases the timer value and an action is triggered when it reaches a particular value. An expiration counter doesn't change its value; an action will be triggered when the difference between the expiration counter and the current time instance reaches a certain value.

As stated earlier, the passage of time is modeled by a *variable* which is either incremented or decremented. Since model checkers can handle only integer variables, we have to model the system with discrete-time semantics. Although we lose the continuous semantics of time, it has been proved that integer clocks are sound for common real-time systems and their properties [10].

The main advantage of using the explicit-time over the implicit-time method its its ability to remember the the current time instant, which helps to model systems with preemptive scheduling. Moreover, it allows us to use general purpose model checkers which have easier learning curves. The explicit-time approach also suffers from the state explosion problem. As the *tick* process decrements the time variable by one unit, the state space increases rapidly as the timing parameters in the system increase. The conceptual simplicity of the explicit-time method has motivated us to carry out a comparison with the timed automata based implicit-time methods on a few case studies.

6. Timing Analysis using SPIN

6.1. Task Chain

We have adapted this example from [19]. We are given a chain of tasks, each of which executes periodically. Each task processes a message, if there is any, in its input buffer and then writes the processed message to output buffer, if it is not full. We have to find out the worst case latency of messages. A task can start processing a given message at periodic intervals and the processed message is written to the output buffer after the worst case execution time of the task. We have used the SPIN [11, 4] model checker for analyzing the system. The basic idea of the explicit-time method is to execute the system one time tick at a time. The problem in doing this is ensuring that actions enabled in the current time tick are executed before a new time tick. This can be done by introducing a new process called *Tick* and forcing each process to stop after executing all

statements enabled in the current clock tick, and waiting for a signal from the *Tick* process before proceeding.

We have made use of the special PROMELA instruction *timeout* for implementing the *Tick* process. The *timeout* action is enabled only when no other process in the system is enabled. This ensures that all the actions enabled in the current instant are executed before executing actions from the next time instant. The *Tick* process decrements the value of all the timers and this is carried out atomically, guaranteeing uniformity of time advancement.

Tasks	Period	WCET
T_1	100	50
T_2	100	12
T_3	50	28
T_4	50	17
T_5	10	4
T_6	10	3
T_7	10	2
T_8	7	1
T_9	15	2
T_{10}	10	5
T_{11}	12	3

Table 10. System description .

	Latency		Time taken	
No. of Tasks	UPPAAL	SPIN	UPPAAL	SPIN
6	233	234	0.4	0.8
7	242	244	0.6	2.29
8	250	251	3.0	10.2
9	267	269	30.14	58.1
10	277	279	63.18	144
11	283	287	284.15	733

Table 11. Bounds on the latencies.

Figure 9 shows the model of tasks in the chain. We have translated the model to PROMELA and tried the timing analysis using SPIN. Table 11 compares the latencies and time taken for verification using the SPIN and UPPAAL model checkers. The verification time is in seconds. The results shows that although the implicit-time approach is more accurate than the explicit-time method, results in case of the later approach are quite comparable to results of former approach. The only problem which was observed is that the memory requirement of the explicit-time method becomes very high as the time parameters increase. The verification was carried out on Intel(R) Pentium(R) Dual CPU T2330 1.60 GHz with 2 GB RAM, running Linux Kernel 2.6.

6.2. Controller Area Network Bus

Using the explicit-time method, we have also tried to perform an end-to-end timing analysis of the controller area network (CAN) bus used in the case study presented in Section 4. The periods and execution times of the tasks are modeled with countdown timers. Whenever a timer expires, *i.e.*, reaches zero, a new instance of a task is released. The asynchronous triggering of the tasks is modeled by using the *buffered channel* construct of SPIN.

Channels in SPIN have the capability of *sorted send*, which allows the task with minimum *id* to acquire the first place in the buffer. This feature of SPIN is used for modeling task preemption. Whenever a higher priority task, *i.e.*, a task with lower *id*, is enabled for execution, it is put at the head of the buffered channel. This stops execution of the current task and starts executing the higher priority task.

Figure 10 shows the PROMELA fragment of the process modeling a task. *Proc_i* is a buffered channel, which models a processor while *exe_i* represents the execution time of the *Task_i*. The variable *rem_i* shows the remaining execution time of the task at any moment. The *expire(exe_i)* is a defined macro that becomes true whenever the variable *exe_i* reaches zero, which represents the fact that *Task_i* has finished its execution.

The PROMELA construct *((Proc ? [eval(id)]))* in line 6 of Figure 10 makes sure that the task will start executing only if it is the highest priority task present in the system, *i.e.*, it is present at the head of the channel. Here we are using the polling facility provided with PROMELA buffered channels. The question mark in the instruction checks if the argument can match at the head of the buffer. The square brackets represent side-effect-free polling, and *eval* computes the current value of the *id*. The instruction at line 7 becomes true whenever the task finishes its execution, and so its *id* must be removed from the buffer. The instruction at line 8 does this. Now we initialize the remaining execution time of the task to its original execution time as shown by a dummy variable *n* in the code.

The system we are analyzing is asynchronous, so the completion of one task may release another task. When a task finishes its execution, the *id* of the newly enabled task should be placed in the corresponding buffered channel. The instruction in line 10 does this through a sorted send. The double exclamation mark inserts a message into the buffer queue ahead of the messages with a larger value.

The currently running task is preempted whenever a higher priority task is enabled for execution. In Figure 10 this condition is indicated by the currently executing task not being at the head of the buffer, thus causing the condition at line 11 to hold true. Whenever a task is preempted, we store its remaining execution time in the corresponding *rem_i* variable so that we can execute the task for that much duration whenever it is rescheduled.

To check the end-to-end latency of messages, we set a deadline variable, a countdown variable, to a *limit* whenever a new message enters in the chain. Whenever that particular message exits the chain, we check that its value is non-zero, *i.e.*, the message latency is less than limit we are using.

Table 12 shows the latencies we have obtained by using

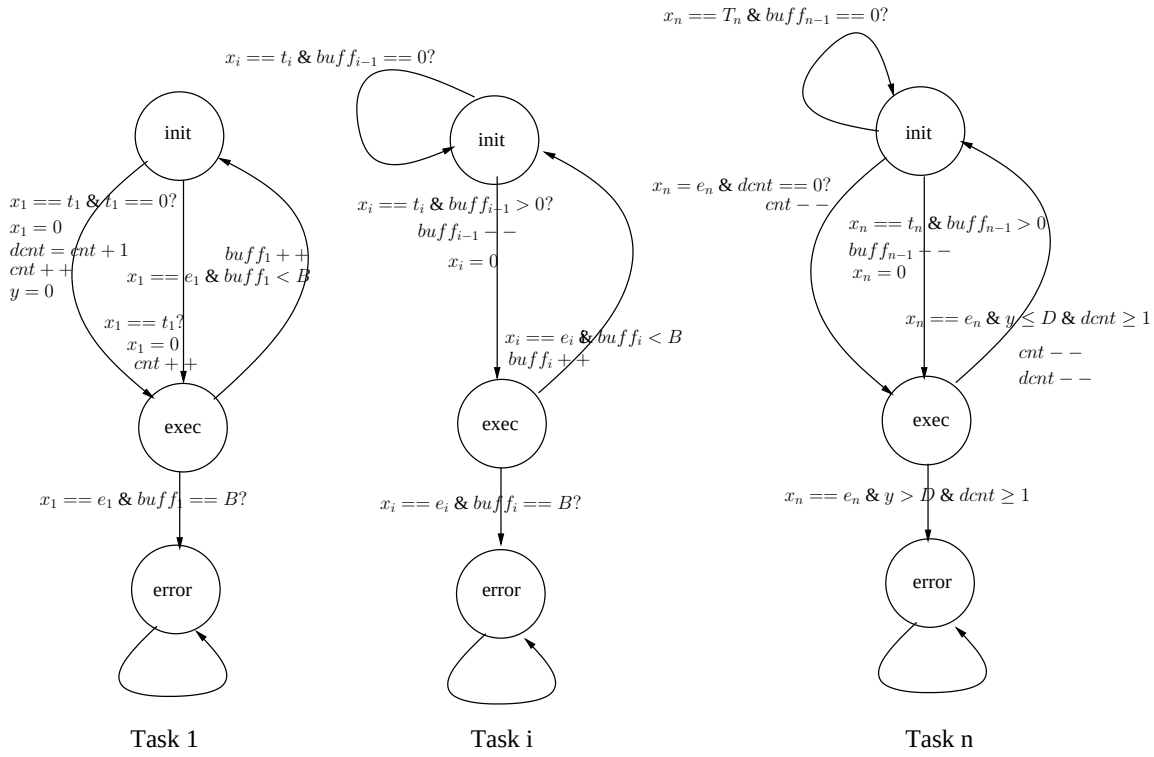

Figure 9. Example: Task Chains

```
1 active proctype()
2 {
3 start: do
4::atomic
5 {
6   ((Proc_i ? [eval(id)])) −−> exe_i = rem_i
7    if  :: expire(exe_i);
8          Proc  ?  eval(id);
9          rem_i = n;
10         (runid == −1) −−> Proc_j !! id;
11       :: !((Proc_i ? [eval[id]))
12            −−> rem_i = exe_i; goto start;
13  fi;
14}
15  od;
16}
```

Figure 10. Promela Fragment for Preemptable Task

Chain	UPPAAL	SPIN
$O_{14} - O_{15}$	28	28
$O_{16} - O_{17}$	50	55
$O_{18} - O_{19}$	110	120

Table 12. Worst case latencies of three task chains

SPIN. As in the above case, we observe that the results obtained are comparable with the results from UPPAAL.

7. Conclusion

In this paper, we have performed timing analysis of complex, distributed real-time systems with preemptive scheduling using timed automata. For handling preemption in timed automata, we have designed our general model of preemptable tasks based on the preemption handling method proposed in Waszniowski *et al* [28]. We have shown that results obtained with timed automata are more accurate than using holistic scheduling and Real-Time Calculus. We have also shown that the proposed task model performs much faster than the existing DREAM tool on the example we have considered.

We have also tried analyzing real-time systems with the explicit-time method by using the model checker SPIN. Here we observed that although the implicit-time methods gives accurate results, explicit-time methods do not perform much worse and the results obtained by using SPIN are only slightly over-approximate.

As future work, we would like to try the timing analysis experiments on bigger and more realistic case studies. A comparison with the Calendar Automata based model checking of Rajeev *et al* [23] for computing end-to-end latencies would be an interesting exercise.

References

[1] M. Abadi and L. Lamport. An Old-Fashioned Recipe for Real Time. *ACMTOPLAS: ACM Transactions on Programming Languages and Systems*, 16, 1994.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2):183–235, 1994.

[3] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems, SFM-RT 2004*, volume 3185 of *LNCS*, pages 200–236. Springer, 2004.

[4] M. Ben-Ari. *Principles of the Spin Model Checker*. Springer, 2008.

[5] S. Chakraborty, S. Künzli, and L. Thiele. A general framework for analysing system properties in platform-based embedded system designs. In *Proc. 6th Design Automation and Test in Europe (DATE)*, pages 190–195. IEEE Press, 2003.

[6] D. B. Chokshi and P. Bhaduri. Modeling Fixed Priority Non-Preemptive Scheduling with Real-Time Calculus. In *RTCSA*, pages 387–392, 2008.

[7] A. David, J. Illum, K. G. Larsen, and A. Skou. Model-based framework for schedulability analysis using UPPAAL 4.1. In G. Nicolescu and P. J. Mosterman, editors, *Model-Based Design for Embedded Systems*, pages 93–119. CRC Press, 2010.

[8] DREAM. `http://dre.sourceforge.net/links.html`.

[9] B. Dutertre and M. Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In *FORMATS/FTRTFTS*, volume 3253 of *LNCS*. Springer, 2004.

[10] T. A. Henzinger, Z. Manna, and A. Pnueli. What Good Are Digital Clocks? In *ICALP '92: Proceedings of the 19th International Colloquium on Automata, Languages and Programming*, volume 623 of *LNCS*, pages 545–558. Springer, 1992.

[11] G. J. Holzmann. The Model Checker SPIN. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[12] N. Kandasamy, J. P. Hayes, and B. T. Murray. Dependable Communication Synthesis for Distributed Embedded Systems. In *Computer Safety, Reliability, and Security, 22nd International Conference, SAFECOMP 2003*, volume 2788 of *LNCS*, pages 275–288. Springer, 2003.

[13] P. Krcal and W. Yi. Decidable and Undecidable Problems in Schedulability Analysis Using Timed Automata. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*. Springer, 2004.

[14] L. Lamport. Real-Time Model Checking Is Really Simple. In *CHARME*, volume 3275 of *LNCS*, pages 162–175. Springer, 2005.

[15] K. G. Larsen, P. Pettersson, and W. Yi. UPPAAL in a Nutshell. *Int. Journal on Software Tools for Technology Transfer*, 1(1–2):134–152, Oct. 1997.

[16] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[17] G. Madl and N. Dutt. Tutorial for the Open-source DREAM Tool. Technical report, CECS, UCI, 2006.

[18] G. Madl, N. Dutt, and S. Abdelwahed. A Conservative Approximation Method for the Verification of Preemptive Scheduling Using Timed Automata. In *IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 255–264. IEEE Computer Society, 2009.

[19] S. Mohalik, A. C. Rajeev, M. G. Dixit, S. Ramesh, P. V. Suman, P. K. Pandya, and S. Jiang. Model checking based analysis of end-to-end latency in embedded, real-time systems with clock drifts. In *Proceedings of the 45th Design Automation Conference, DAC 2008*, pages 296–299. ACM, 2008.

[20] M. D. Natale, C. Pinello, P. Giusto, and A. S. Vincentelli. Optimizing End-to-End Latencies by Adaptation of the Activation Events in Distributed Automotive Systems. In *Real-Time and Embedded Technology and Applications Symposium, IEEE*, pages 293–302. IEEE Computer Society, 2007.

[21] J. C. Palencia and M. G. Harbour. Schedulability Analysis for Tasks with Static and Dynamic Offsets. In *In Proceedings of the 19th IEEE Real-Time Systems Symposium*, pages 26–37, 1998.

[22] S. Perathoner, E. Wandeler, L. Thiele, A. Hamann, S. Schliecker, R. Henia, R. Racu, R. Ernst, and M. G. Harbour. Influence of different system abstractions on the performance analysis of distributed real-time systems. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, pages 193–202. ACM, 2007.

[23] A. C. Rajeev, S. Mohalik, M. G. Dixit, D. B. Chokshi, and S. Ramesh. Schedulability and End-to-end Latency in Distributed ECU Networks: Formal Modeling and Precise Estimation. In *Proceedings of the 10th International Conference on Embedded Software, EMSOFT 2010*. ACM, 2010.

[24] L. Thiele, S. Chakraborty, and M. Naedele. Real-time calculus for scheduling hard real-time systems. In *International Symposium on Circuits and Systems (ISCAS 2000)*, pages 101–104, 2000.

[25] K. Tindell, A. Burns, and A. Wellings. Calculating Controller Area Network (CAN) Message Response Times. *Control Engineering Practice*, 3:1163–1169, 1995.

[26] K. Tindell and J. Clark. Holistic schedulability analysis for distributed hard real-time systems. *Microprocessing and Microprogramming*, 40(2-3):117–134, 1994.

[27] L. van den Berg, P. A. Strooper, and K. Winter. Introducing Time in an Industrial Application of Model-Checking. In *FMICS*, volume 4916 of *LNCS*, pages 56–67. Springer, 2007.

[28] L. Waszniowski and Z. Hanzalek. Over-Approximate Model of Multitasking Application Based on Timed Automata Using Only One Clock. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 2*, page 128.1. IEEE Computer Society, 2005.