Timing Modeling and Analysis for AUTOSAR OS Schedule Tables*

ABSTRACT

Schedule table mechanism is an important characteristic in AUTOSAR OS during addressing its real-time property and flexibility. Schedule table provides an encapsulation of a statical configuration with one or more actions (i.e., events set or tasks activation). It is challenging to manually analyze schedulability of tasks in schedule tables. In this paper, we take into consideration the AUTOSAR OS scheduling with preemptive periodic schedule tables in uniprocessor. A schedule table is formally modelled by a transition system. An algorithm based on the models is presented to analyze schedulability by checking all of the possible offsets between schedule tables. Furthermore, checking schedulability tool is implemented. Testbenches of schedule tables are executed in our tool and an industrial AUTOSAR OS on microcontroller respectively. The result demonstrates our method is effective.

ACM Reference Format:

. 2018. Timing Modeling and Analysis for AUTOSAR OS Schedule Tables. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, Article 4, 10 pages. https://doi.org/10.475/123_4

1 INTRODUCTION

Formal methods have been widely and successfully applied to functional properties verification. Unfortunately, it is difficult to verify some non-functional characteristics, which are the key to maintain validity in highly safety-critical real-time systems. Formal analysis on schedulability of task models is researched. Liu et. al [14] first build a task model for periodic tasks with a period and execution time, and discuss two scheduling algorithms over the model. Stigge et al. [17] describe a task with a Diagraph Real-Time (DRT) task model, and study the feasibility problem on preemptive periodic tasks in a uniprocessor. For analyzing the temporal validity of real-time data objects, Wang et al. [19] give transaction model with three parameters, a fixed priority, its worst-case execution time (WCET) and a relative deadline. The temporal validity of real-time data object is analyzed by periodic update transactions. S.Baruah [5] analyze schedulability in a general 3-parameter sporadic task model [7] for mixed-criticality concurrent real-time tasks. Moreover, S.Baruah et al. [6] propose a scheduling model for periodic preemptive tasks inspired by control theory. Despite the rapid advancements in real-time scheduling theory, industry is willing to employ the very simple cyclic executive approach [4] for corrective scheduling to meet stringent certification requirement,. C.Deutschbein et al. [12] present the problem of constructing cyclic executive upon multiprocessors.

AUTOSAR (AUTomotive Open System ARchitecture) [3] is an open and standardized layered software architecture and interfaces. The integration of software modules from different vendors forms the complete architecture. As a consequence, more and more automotive companies tend to take AUTOSAR as their primary standard for developing their base software project, such as operating system. Researchers working on AUTOSAR OS are rich in progressive and proactive. There have been many studies on analyzing AUTOSAR OS specification [1]. Peng et al. adopt timed CSP method to model OS and the engine management system (EMS), some safety properties based on CSP models are verified through process analysis toolkit in [16]. Huang et al. [10] apply formal semantics to describing tasks in AUTOSAR OS with time protection mechanism. The model predicts whether a task would violate its time constraint, and is implemented in Mathematica tool with a case study. Zhu et al. [23] focus on the timing properties of AUTOSAR OS and propose an automatic verification framework based on rewriting logic to analyze the timing behaviors. Therefore, many researches of AUTOSAR OS focus on its specification, implementation and verification.

For schedulability of the AUTOSAR OS, both Zhao et al. [22] and Hatvani et al. [9] adopt preemptive threshold to improve the schedulability. Tasks are set with a higher priority and reduced execution time by the stack usage. In order to assist developers to assign the priorities of tasks, Yoon et al. [21] present a real-time task chain model, which helps to obtain a near-optimal priority assignment from the model. They concentrate on the scheduling of tasks, but few researches deal with schedule tables. Wang et al. propose a method for generating schedule tables containing periodic tasks with dependence in [20] for the multi-core architectures. However the method does not give the correctness proof of generating schedule tables.

Among those aforementioned researchers, most of them analyze schedulability by modeling tasks, few attentions were given to schedule table mechanism [3], which is an important mechanism in AUTOSAR OS. For automobile being a high safety-critical real time system, a practical approach [4] applies to schedule tables in order to satisfy all tasks deadline. Inspired by the method, we present a formal method to build deterministic schedule tables. The correctness of these deterministic schedule tables is proved, which means to verify every task in schedule tables meets its deadline.

A formal model is constructed for each schedule table. A set of schedule tables are composed by concurrent of all models. Tasks in schedule tables are static analyzed to prove that whether tasks run in the worst case response time. We propose an algorithm to travel all of the possible offsets between schedule tables so that each situation can exactly complete task execution. A time interval between a task releasing and its deadline is checked to confirm

*Produces the permission block, and copyright information

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 123-4567-24-567/08/06…$15.00
https://doi.org/10.475/123_4

whether enough execution time for itself and other higher priority tasks. Because all schedule tables in this method are periodic, analysis of infinite running of tasks is transformed to finite running of tasks by introducing hyperperiod of schedule tables. In addition, after analyzing the schedulability of a set of schedule tables by our approach, we run testbenches of schedule tables on an implemented AUTOSAR OS based on TC1782 32-Bit Single-Chip Micro-controller. The result shows the consistency between them.

The rest of this paper is organized as follows. Section 2 introduces some concepts of AUTOSAR and schedulability analysis. Section 3 discusses the constraints of AUTOSAR OS in this paper and describes our formal models of schedule tables. Section 4 presents some definitions and notations of our algorithm. In Section 5, we propose a method by integrating concepts in section 4. Then, we compare the result of our method with the execution on hardware in Section 6. Finally, in Section 7, we give the conclusion and future work.

2 BACKGROUND

For a better understanding of our work, we give a brief overview of AUTOSAR operating system and methods of schedulability analysis.

2.1 AUTOSAR Operate System

This part is given as prerequisites of this paper. We will state some concepts of AUTOSAR which will be used later. Those concepts are referred to OSEK/VDX operating system specification [15] and specification of AUTOSAR operating system [3].

2.1.1 Tasks and Scheduling Policy. AUTOSAR considers a task as the minimum unit to schedule. Even a simple application also needs to map to a task. Each task has to accomplish a specific function. A complex application is constituted by various tasks. The implementation of a task consists of a block of sequential code. AUTOSAR provides two types of tasks: extended and basic tasks.

Each basic task has three states: *running*, *suspended* and *ready*. A task in the *running* state is executed by processor. At most one task can be in *running* state at the same time. A task in *ready* state means the task has been put into ready queue and is waiting for allocation of the processor. A suspended task can be put into ready queue by a system service. A *suspended* task cannot be transferred into *running* state or be transferred from *ready* state directly. Once a basic task in *running* state is preempted by a task with a higher priority, it will be transferred into *ready* state. A running task is transferred into *suspended* state by a system service to terminate a task. An extended task has a strong resemblance to a basic task with except that 1) the former one could wait for a system event in an extra state named *waiting*, 2) the basic task is allowed to be activated once or multiple times while an extended task can only be activated once.

AUTOSAR uses the highest priority first and first come first served as its scheduling policy. AUTOSAR reserves a FIFO queue for each priority. When a task is activated, it will be put into the corresponding queue. Then scheduler will detect those queues in order of the highest priority and choose the task with the highest priority to execute. If more than one tasks in ready queue have the highest priority, the scheduler will choose the first one has been put into the ready queue. For the high efficiency of the system, the priorities are distributed to tasks statically.

2.1.2 Counters and Schedule Table Mechanism. AUTOSAR operating system offers at least one system counter. Counters aim to keep track of how many ticks have been elapsed. Every counter has its max allowed value. When it reaches its max allowed value, the counter will reset to 0.

The schedule table mechanism is a new concept introduced in AUTOSAR which could statically define the pattern of tasks activation. Each schedule table maps to a specific counter and encapsulates a predefined set of expiry points. An expiry point contains one or more actions, which can be either activating a task or setting an event. Each expiry point has a unique offset in ticks from the start of the schedule table. In addition, a schedule table has a specific duration which defines the modulus of the schedule table.

At runtime, a schedule table is driven by a counter. One tick on the counter corresponds to one tick on schedule table. Since a schedule table start, if the increment of the counter equals to an offset of an expiry point, the actions of this expiry point will be executed. AUTOSAR also provides three states of schedule table.

A schedule table is initialized as *SCHEDULETABLE_STOPPED* state. Users can activate the schedule table by system services *StartScheduleTableAbs* or *StartScheduleTableRel*. When corresponding counter reaches the preset value, the schedule table will be transferred into *SCHEDULETABLE_RUNNING* state. In this state, expiry points of the schedule table could be iterated in order until schedule table ends. After that, the state will be transferred into *SCHEDULETABLE_STOPPED*. When a user calls the system service *NextScheduleTable*, the schedule table will be transferred into *SCHEDULETABLE_NEXT* state from *SCHEDULETABLE_STOPPED* state, which means the schedule table is waiting for the current schedule table to end. The transition into the *SCHEDULETABLE_STOPPED* state is triggered by system service *StopScheduleTable*.

AUTOSAR offers two types of behavior of schedule tables: single-shot and repeating. On one hand, if a schedule table is configured as single-shot, it will be transferred into *SCHEDULETABLE_STOPPED* state after the final expiry point is processed. On the other hand, for a repeating schedule table, after the final expiry point has been processed, it loops back to the first expiry point. Note that for tasks activated by a repeating schedule table, they can be regarded as periodic tasks and their period equals to the duration of the repeating schedule table.

We give a simple example in Figure 1 to illustrate the construction of schedule table. This schedule table is denoted by st_1 with the duration of 17 ticks. st_1 contains three expiry points ep_1, ep_2 and ep_3, where the initial expiry point ep_1 locates at 4 ticks from the start of st_1 with the action of activating task t_1, ep_2 defines 8 ticks as its offset and could activate task t_2, ep_3 defines 11 ticks as its offset and activates task t_3. The delays between two adjacent expiry points are shown as 4 ticks (from EP1 to EP2) and 3 ticks (from EP2 to EP3).

2.2 Schedulability Analysis

The researches about schedulability analysis focus on the problem whether real-time tasks could satisfy its time property. We declare the meaning of schedulability: schedulability declare whether a

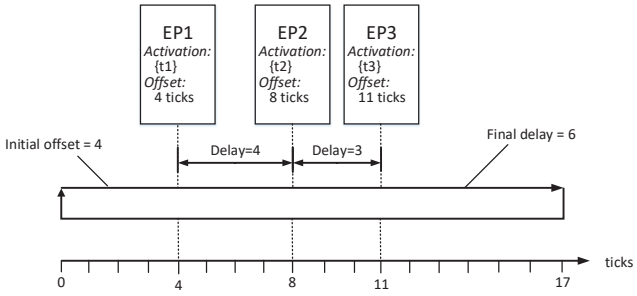


Figure 1: An example repeating scheduel table cotaining three expiry points, and t_1, t_2, t_3 is three tasks.

set of schedule tables is schedulable. Schedulable means all tasks activated by schedule tables could take to execute before its deadline, otherwise, this set of schedule tables is non-schedulable.

In order to figure out the schedulability of real-time systems, several formal models have been presented to abstract the behaviors of tasks. Generally, a model abstracts a task as a tuple (T_i, D_i, C_i, L_i), which represent its period, deadline, worst case execution time and priority level respectively [7].

We can categorize those schedulability analysis methods by the existing scheduling algorithm. One of the most common algorithm is the earliest deadline first algorithm (EDF). EDF is considered as an optimized algorithm on preemptive uniprocessors to analyze a set of independent tasks characterized by activate time, deadline and worst case execution time. For such a dynamic priority scheduling algorithm, the schedulability analysis methods which determines whether a system is EDF schedulable is co-NP in the strong sense [8].

Since our work focuses on the single processor platform with static priority scheduling, we now pay attention the schedulability analysis method based on the response time analysis (RTA) [2]. Response time analysis is suited for fixed priority tasks system. This method calculate the worst case response time, then verify whether the worst case response time of a task is in its deadline.

3 FORMAL MODELS OF SCHEDULE TABLES

For AUTOSAR operating system, there are two main factors that make the schedulability analysis intractable: 1) generally, there are several schedule tables run concurrently, 2) users can manipulate the proceeding of schedule tables by calling system services, i.e., schedule tables may stop or start running at any time, which is totally unpredictable. Those factors bring more complication to schedulability analysis.

The problem of schedulability analysis raises two questions: 1) how to define a proper formal model to describe the behavior of schedule tables, 2) how to propose an algorithm to analyse the schedulability of schedule tables by the formal model. In this section, we state our solution to the first problem.

3.1 Constraints of AUTOSAR OS

On account of the widespread use of embedded automotive systems, different embedded applications may require various characteristics of operating systems as well as hardware environments are also likely to be different. Those factors demand different features of the AUTOSAR operating system. For those features, our work focus on several characteristics among them:

(1) Tasks are allowed to share the same priority level and lower number for higher priority.
(2) We take no account of extended tasks and events.
(3) All tasks are full preemptible.
(4) The scheduling policy is the highest priority first. When there are more than one tasks have the same priority. The scheduler will choose the first one has been put into the ready queue.
(5) A basic task can be activated infinitely many times.

For convenience, if a task is assigned to more than one expiry points, we regard them as different tasks. In additional, we assume all schedule tables are repeating. In fact, a repeating schedule table could be transferred to a single-shot schedule table by converting the final delay into infinitely great.

3.2 Formal Models

We use a directed digraph to describe the behavior of a schedule table and named it Digraph Schedule Table model (DST). Each DST is a tuple $(Node, Tasks, Act, \rightarrow, Delay)$, where

- $Node$ is a finite set of expiry points.
- $Tasks$ is a finite set of tasks. We use a tuple (e, d, l) to represent a task. Assuming $t_k \in Tasks$, then $e(t_k), d(t_k), l(t_k)$ denote the execution time, deadline and priority level of task t_k respectively.
- $Act : Node \rightarrow 2^{Tasks}$ is a function. Act relates a set $Act(ep) \in 2^{Tasks}$ of actions to an expiry point ep.
- $\rightarrow \subseteq Node \times Node$ is a transition relation to describe the execution sequence of expiry points.
- $Delay : Node \rightarrow ticks$ is a function to describe the delay after processing an expiry point. $Delay$ of the last expiry point of a schedule table equals to the final delay plus the initial offset.

Note that we conceal the initial offset in DST. Because in the processing of a repeating schedule table, except the first loop, the initial offset always can be considered as a part of the delay of the last expiry point.

With regard to the definition of $Delay$, the duration of a schedule table can be obtained by

$$\delta(DST) = \sum_{ep_i \in Node} Delay(ep_i).$$

During the execution of a schedule table, expiry points will expire circularly. We record each time of expiring as an instance of expiry point. An instance of expiry point is specified by an expiry point and the time it expires. We use a pair (ep_i, r_i) to represent an instance, where r_i is the time that the expiry point expires. Naturally to point out that an execution of a DST corresponds to an infinite sequence of instances of expiry points. We use a path $\rho = [(ep_1, r_1), (ep_2, r_2), \ldots, (ep_k, r_k)]$ to denote a fragment of an execution. When an expiry point expire, corresponding tasks will be activated then. For each time a task is activated, we record that as an instance of task. We use a 4-tuple (e_i, d_i, l_i, r_i) to denote an instance of a task with execute time e_i, deadline d_i, priority l_i and activate time r_i. Durning a path, expire times of instances of expiry

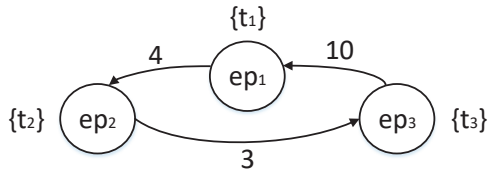


Figure 2: An example of DST for st_1, where $t_1 =< 2, 4, 2 >$, $t_2 =< 2, 3, 1 >$ $t_3 =< 2, 9, 6 >$.

points are constrained by the delay of corresponding expiry points, i.e.,

$$r_{i+1} = r_i + Delay(ep_i).$$

Similar to the calculation of duration, we get the length of a path by accumulating delays of expiry points. For a path $\rho = [(ep_1, r_1), (ep_2, r_2), \ldots, (ep_k, r_k)]$, the length $\iota(\rho)$ is calculating by:

$$\iota(\rho) = \sum_{i=1}^{k-1} Delay(\rho_i),$$

where we use ρ_i to represent the ith instance of expiry point in the path ρ, $ep(\rho_i)$ to represent this expiry point and $r(\rho_i)$ to represent the expire time. For a path which only contains a single expiry point, we make the length of it as 0.

3.3 Example Model

We model the scheduel table which mentioned above in Figure 2 and give the tuple of dst_1 as follow.

$dst_1 = \{Node_1, Tasks_1, Act_1, \rightarrow_1, Delay_1\}$, where

- $Node_1 = \{ep_1,\ ep_2,\ ep_3\}$ which cotains all expiry points in dst_1.
- $Tasks_1 = \{t_1,\ t_2,\ t_3\}$ which cotains all tasks in dst_1.
- Act_1 refers to actions of an expiry point:
  $Act_1(ep_1) = t_1$,
  $Act_1(ep_2) = t_2$,
  $Act_1(ep_3) = t_3$.
- $\rightarrow_1 = \{(ep_1, ep_2), (ep_2, ep_3), (ep_3, ep_1)\}$ which shows the execution sequence of expiry points.
- $Delay_1$: the $Delay$ of expiry points is defined by
  $Delay_1(ep_1) = 4$,
  $Delay_1(ep_2) = 3$,
  $Delay_1(ep_3) = 10$.

We assume there is a path ρ over dst_1 which $\rho_1 = ep_1$ and $r_1 = 0$. The expire time of second instance of expiry point is $0 + Delay(ep_1) = 4$. If $\rho = [(ep_1, 0), (ep_2, 4), (ep_3, 7), (ep_1, 17)]$, the length of ρ can be calculated by

$$\iota(\rho) = Delay(ep_1) + Delay(ep_2) + Delay(ep_3) = 17.$$

4 DEFINITION AND NOTATION

In this section, we will discuss how to verify whether a task could always satisfies its time property. We start by introducing some concepts as preparations, then those concepts are integrated into a complete analysis method.

4.1 Computational Requirement

The main idea of our analysis method is to check whether the time interval between the task's activation and deadline is big enough to accommodate the execution time of itself and other tasks which could interrupt it. But how to delineate the execution of those tasks?

For example, if we are testing an instance of task $(e(t), d(t), l(t), r(t))$, in consideration of the features of schedule policy, this instance could not be affected by tasks with a lower priority than $l(t)$, or tasks have priority $l(t)$ and a larger activate time. Based on this fact, we introduce the concept of computational requirement to accumulate the execution time of tasks in an expiry point and dismiss the tasks which could not affect this instance.

Definition 4.1. For an instance of expiry point (ep, r), we use computational requirement $\omega_{l(t),r(t)}(ep)$ to accumulate the execution time of tasks in $Act(ep)$, except those tasks which have a lower priority than $l(t)$ if $r \le r(t)$. When $r > r(t)$), we also except tasks which have priority $l(t)$:

$$\omega_{l(t),r(t)}((ep, r)) = \sum_{task \in \xi_{l(t),r(t)}(ep)} e(task), \tag{1}$$

where

$$\xi_{l(t),r(t)}((ep, r)) = \begin{cases} \{t' | t' \in Act(ep) \wedge l(t') \le l(t)\} & if\ r \le r(t) \\ \{t' | t' \in Act(ep) \wedge l(t') < l(t)\} & if\ r > r(t) \end{cases}$$

After proposing the computational requirement function, the total execution time during a path can be computed. We still use the name *computational requirement* to denote this function, but a different way to calculate.

Definition 4.2. For a path $\rho = [(ep_1, r_1), (ep_2, r_2), \ldots, (ep_n, r_n)]$, we use computational requirement $\Omega_{l,r}(\rho)$ to calculate how long those expiry points on this path will take to execute and only consider the tasks which could affect an instance of task (e, d, l, r):

$$\Omega_{l,r}(\rho) = \sum_{i=1}^{n} \omega_{l,r}(\rho_i). \tag{2}$$

4.2 Request Function

Due to the speciality of schedule table mechanism, after an expiry point expires, other expiry points cannot expire during delay time. We could use the prefix request function [18] $prf_{l,r}^{ep}(\theta)$ to abstract the relation between length and computational requirement of paths.

Definition 4.3. For an expiry point ep, let prefix request function $prf_{l,r}^{ep}(\theta)$ denotes the maximal computational requirement $\Omega_{l,r}(\rho)$ among those paths ρ which start by $(ep, 0)$ and $\iota(\rho) < \theta$.

$$prf_{l,r}^{ep}(\theta) = max\{\Omega_{l,r}(\rho) | \rho\ start\ in\ (ep, 0)\ \wedge\ length(\rho) < \theta\}. \tag{3}$$

Beacuse no path has a length which is less than 0, we consider the prefix request function is meaningless when $\theta = 0$.

Figure 3 shows an example of prefix request function $prf_{4,0}^{ep_1}(\theta)$ (cf. Section 3.3).

When $0 < \theta \le 4$, the path must be $[(ep_1, 0)]$, which have computational requirement $\Omega_{4,0}([(ep_1, 0)]) = 2$. When $4 < \theta \le 7$,

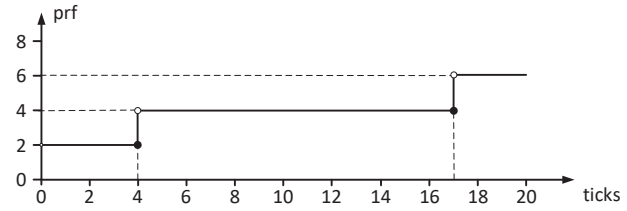
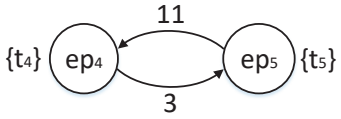
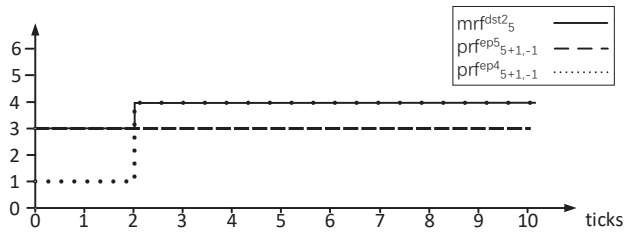


Figure 3: An example for prefix request function of $prf_{4,0}^{ep_1}(\theta)$ according to dst_1 in Figure 2.



(a) An example of dst_2 which contains two expiry points and $t_4 =< 1, 3, 3 >$, $t_5 =< 3, 8, 5 >$.



(b) Maximal request funciton of dst_2 for $l = 5$.

Figure 4: The dst_2 contains two expiry points, so there are two prefix request functions $prf_{5+1,-1}^{ep_4}$ and $prf_{5+1,-1}^{ep_5}$ in Figure 4b. When $0 < \theta \leq 2$, $prf_{5+1,-1}^{ep_5}$ makes more computational requirement. But in time interval (2,10], $prf_{5+1,-1}^{ep_4}$ provides the value of the maximal request function.

the path $[(ep_1, 0), (ep_2, 4)]$ makes the maximum computational requirement $\Omega_{4,0}([(ep_1, 0), (ep_2, 4)]) = 4$. when $7 < \theta \leq 17$, the path ρ can be $[(ep_1, r0)]$, $[(ep_1, r0), (ep_2, 4)]$ or $[(ep_1, 0), (ep_2, 4), (ep_3, 7)]$. But in ep_3, the priority of t_3 is lower than 4, so the prefix request function wouldn't accumulate it, i.e., $\Omega_{4,0}([(ep_1, 0), (ep_2, 4)]) = \Omega_{4,0}([(ep_1, 0), (ep_2, 4), (ep_3, 7)]) = 4$. The function will keep returning 4 until θ exceeds 17.

A DST has as many prefix request functions as expiry points in this DST, so we use the maximal request function to integrate those prefix request functions.

Definition 4.4. For a DST, the maximal request function $mrf_l^{dst}(\theta)$ calculates the maximum execution time caused by a time interval.

$$mrf_l^{dst}(\theta) = max\{prf_{l+1,-1}^{ep}(\theta) | ep \in Node\} \tag{4}$$

An example of a maximal request function is given in Figure 4, where we provide another DST $dst2$ and illustrate the maximal request function of dst_2.

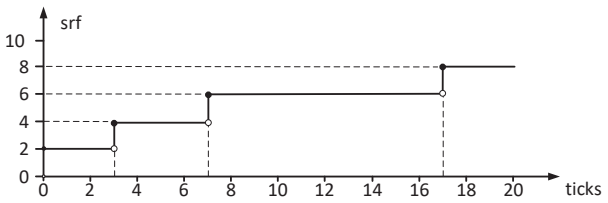


Figure 5: Suffix request function of $srf_4^{ep_3}(\theta)$ according to DST_1 in Figure 2.

Besides considering paths which start with a specific expiry point, we also calculate the computational requirement of paths which end up in a specific expiry point ep by the suffix request function [18]. A slightly different between srf and prf is that the former ignores the expire time.

Definition 4.5. For an expiry point ep, we use a suffix request function $srf_l^{ep}(\theta)$ to calaulate the maximal computational requirement $\Omega_{l+1,-1}(\rho)$ among those paths ending up in (ep, r) but whose length not exceeds θ.

$$srf_l^{ep}(\theta) = max\{\Omega_{l+1,-1}(\rho) | \rho \; end \; with \; (ep, r) \; \wedge \; length(\rho) \leq \theta\}. \tag{5}$$

In particular, we use $\Omega_{l+1,-1}(\rho)$ to accumulate the execution time of tasks which have priority not lower than l, regardless the activate time. We use Figure 5 to illustrate the suffix request function $srf_4^{ep_3}(\theta)$.

4.3 Busy Window

A RTOS (Real-Time Operating System) strictly requires that the finish times of tasks are not allowed to exceed deadline. But in real life, identifying the task which causing time fault actually may be far from obvious. Imagine a task which takes too long to execute, but it does not violate its deadline. The task may block the execution of following tasks. Then the influence will spread throughout the running of tasks, which could makes a normal task misses its deadline. But in a execution of a DST, how to recognize those tasks which could cause or spread influences? In order to find a solution to this problem, we introduce the well-known concept named busy window [13].

Busy window for priority level l is a time interval that processor continuously executes tasks with priority l or a higher priority. If we find the maximal busy window for l, then we could delimit how long a task could spread influence, i.e., an instance of task with priority l can only be affected by tasks within this boundary. We denote the upper bound of busy window for l by σ_l.

Lemma 4.6. *For a DST set τ, we can get the upper bound of busy windows for l by finding the minimum positive number θ which could satisfy*

$$\sum_{dst \in \tau} mrf_l^{dst}(\theta) = \theta. \tag{6}$$

Proof. Firstly, we state a fact that at the point of a busy window start, at least one expiry point expire in that point. So the computation of the upper bound of busy windows is all about finding those expiry points.

If there is a θ which satisfies Equation 6, we can find those paths which offer the maximal request functions at θ from DSTs. Then a set of expiry points is obtained by combining the first expiry points in those paths. When those expiry points expire at the same time, we can get the upper bound of busy windows for l.

Now assuming we use this method to get an expiry point ep' in dst'. But there is another expiry point ep'' in dst' which could make a longer busy window. Because the path start with $(ep'', 0)$ does not offer the value of $mrf_l^{dst'}$ at θ, i.e.,

$$prf_{l+1,-1}^{ep''}(\theta) < prf_{l+1,-1}^{ep'}(\theta).$$

When we choose ep'' instead of ep' to expire with other expiry points simultaneously, the sum of the prefix request function is less than θ, i.e.,

$$prf_{l+1,-1}^{ep''}(\theta) + E < prf_{l+1,-1}^{ep'}(\theta) + E = \theta, \tag{7}$$

where $E = \sum_{dst \in \tau - d1} mrf_l^{dst}(\theta)$.

So the busy window starting with ep'' is shorter than θ, leading to a contradiction. □

In some non-schedulable cases, the upper bound of busy windows is infinitely great. In order to recognize this situations, we introduce the definition of the hyperperiod of schedule tables as follow.

Definition 4.7. For a set of DSTs $\tau = \{dst_1, dst_2, \ldots, dst_n\}$, we assume durations of them as $\delta_1, \delta_2, \ldots, \delta_n$ respectively. The hyperperiod is the least common multiple of all duration of all schedule tables, i.e.,

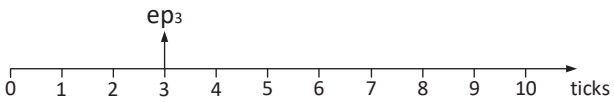
$$lcm(\delta_1, \delta_2, \ldots, \delta_n).$$

Lemma 4.8. *A finite set of schedule tables is non-schedulable if there is a busy window exceeds hyperperiod of those schedule tables.*

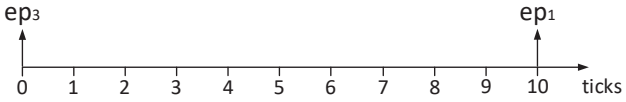
Proof. The scheduling process of a set of repeating schedule tables is cyclic. If any busy window exceeds the hyperperiod, the processor will continuously execute tasks in this hyperperiod. In the next hyperperiod, except tasks which is activated in this hyperperiod, the processor also has to execute those tasks which are activated during the previous hyperperiod. So the tasks with the lowest priority must be blocked. By parity of reasoning, the task with the lowest priority must violate its deadline at some point. □

4.4 Combination of Expiry Points

In order to achieve completeness in schedulability analysis, our method needs to completely cover every schedule situations, including all offsets between schedule tables. Assuming there is a set of DST $\tau = \{dst_1, dst_2, \ldots, dst_n\}$, the amount of offsets between those schedule tables equals to $\delta(dst_1) \times \delta(dst_2) \times \cdots \times \delta(dst_n)$. The various offsets may lead to state explosion, so we try to mitigation this problem by using an expiry point to represent a series of offsets.



(a) ep_3 expires at 3, the computational requirement which is caused by this time interval is $\omega_{l,r}((ep_3, 3))$.



(b) ep_3 expires at 0, ep_1 expires at 10. This time interval makes the computational requirement as $\omega_{l,r}((ep_3, 0)) + \omega_{l,r}((ep_1, 10))$.

Figure 6: In a time interval with length 10, the computational requirement which is caused by scenario (b) is larger than it which is caused by scenario (a), i.e., the time interval in scenario (b) dominates it in scenario (a).

Expiry points shift as the DST changes its offset and now we temporarily restrict the range of shifting into a time interval. Because our schedulability analysis method always focuses on the computational requirement, so, if the computational requirement which is caused by a specific time interval is not less than those which are caused by a set of other time intervals, then we say this time interval dominates the others, cf. Figure 6.

As Figure 6, when ep_3 expires at the begin, this time interval always dominates others that make ep_3 the first appearance. Regardless of the length of time interval, we can use ep_3 to denote this time interval, then represents the set of time interval which imply a series of offsets. So we replace offsets of a DST by its expiry points and replace $\delta(dst_1) \times \delta(dst_2) \times \cdots \times \delta(dst_n)$ by combinations of expiry points.

Definition 4.9. For a set of DSTs $\tau = \{DST_1, DST_2, \ldots, DST_n\}$, we use Φ to represent all combinations of expiry points of τ.

$$\Phi(\tau) = Node_1 \times Node_2 \times \cdots \times Node_n. \tag{8}$$

We pick one expiry point from each DST to form a set which is denoted by ε, then $\varepsilon \in \Phi$.

4.5 Schedulability Analysis of Tasks

As mentioned above, we analyze the schedulability by checking whether the time interval between activation and deadline of a task is big enough to accommodate the execution time of itself and other tasks which could interrupt it. But according to Section 4.3, except the time interval after activation, we also need to consider the time interval before activation. Therefore, we make the time interval grow further to the left. This brings another question, "how far should we extend the time interval?" This answer has been given in Section 4.3, i.e., the length we extended should be less than the boundary of busy windows. Figure 7 shows the original time intervals with length y and the extended time intervals with length $x + y$.

When several DSTs work concurrently, the influence on the execution of a specific task come from three parts: 1) tasks which

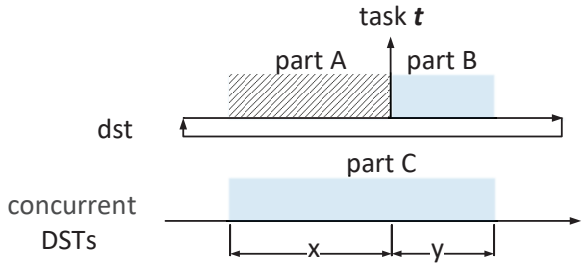


Figure 7: An illustration of schedulability analysis of task t which is activated by expiry point ep. There are three parts may affect the execution of t: A) interruption from dst before task t is activated, B) interruption from dst after task t is activated, C) interruption from other DSTs.

are activated before the specific task from the same DST, 2) tasks which are activated after the specific task from the same DST, 3) tasks which from other DSTs, cf. Figure 7. Note that the three parts are subject to change as DSTs shift their offsets or we change x and y.

After sorting out these three parts, we check whether the extended time interval could accommodate them.

Theorem 4.10. *For a finite set of DSTs τ, if there is a $dst \in \tau$, expiry point $ep \in Node$, task $t \in Act(ep)$. Then task t always meets its deadline if:*

$$\begin{aligned}\forall \varepsilon \in \Phi(\tau - \{dst\}), x \in [0, \sigma_l] : \exists y \in [e(lt), d(lt)] : \\ srf^{ep}_{l(t)}(x) + prf^{ep}_{l(t),-1}(y) - \omega_{l(t),-1}(ep) \\ + \sum_{ep_i \in \varepsilon} prf^{ep_i}_{l(t),x}(x+y) \le x + y,\end{aligned} \tag{9}$$

where $\omega_{l(t),-1}(ep)$ is the overlap between $srf^{ep}_{l(t)}$ and $prf^{ep}_{l(t),-1}$ and we use the dst as the controlled DST but change the offsets of other DSTs.

Proof. Assuming Equation 9 evaluates to false, but task t always meets its deadline. Since Equation 9 is false, there must exists a $\varepsilon \in \Phi(\tau - \{dst\})$ and a $x \le \sigma_l$ makes the value of Equation 10 is larger than $x + y$, where $y = 0, 1, 2, \ldots, d(t)$.

$$\underbrace{srf^{ep}_{l(t)}(x)}_{partA} + \underbrace{prf^{ep}_{l(t),-1}(y)}_{partB} - \underbrace{\zeta^{(ep)}_{l(t),-1}}_{overlap} + \underbrace{\sum_{ep_i \in \varepsilon} prf^{ep_i}_{l(t),x}(x+y)}_{partC}, \tag{10}$$

Equation 10 accumulates the execution time of task t and all tasks could affect it, i.e., t is the last task to be finished among them. Therefore, the minimal y makes Equation 10 not larger than $x + y$ is the worst-case responds time. But due to inexistence of $y \le d(t)$ could satisfy Equation 9, so the WCRT exceed deadline. When expiry points in ε expire x ticks before the activation of t contemporary, we find a situation that task t could not meet its deadline, leading to a contradiction.

Now assuming Equation 9 holds, but task t may violates its deadline. If an instance of t misses its deadline, there must exist a group of instances of tasks which could interrupt the execution of instance of t. We assume that the smallest activate time in this group is x ticks smaller than activate time of the instance of t. According to Lemma 4.3, $x \le \sigma_l$. Since the Equation 9 hold, the execution time caused by this group is less than or equal to $x + y$. So the task t must be finished within y ticks, which leading to a contradiction. □

5 SCHEDULABILITY ANALYSIS METHOD

In this part, we synthesizing those definitions given above into an integrated method to deal with the schedulability analysis problem.

5.1 Stages of Method

The process of our integrated method can be divided into several stages as follows:

Get hyperperiod : We calculate the hyperperiod of the set of DSTs $\tau = \{dst_1, dst_2, \ldots dst_n\}$.

Find a task with the lowest priority : Picking one task with the lowest priority among $Task_1 \cup Task_2 \cup \cdots \cup Task_n$ and denoting it as lt.

Divide the set of DSTs : dividing τ into two parts: i) the DST could activate lt, ii) the set of the other DSTs. Denoting the two parts by $ldst$ and $other\_dsts$ respectively. The next four steps are aimed to verify whether lt could always meet its deadline.

Calculate boundary of busy window : Calculating the upper bound of busy windows for $l(lt)$ and denoting that by σ_l. If σ_l exceed the value of hyperperiod, the set of DSTs is non-schedulable.

Set instances of tasks : After picking one $\varepsilon \in \Phi(other\_dsts)$, we structure a path for each DST: expiry points in ε expire x ticks before releasing task lt at the same time, cf. Figure 7. We set expire time of those expiry points as 0. The rest of paths are structured according to delays of expiry points. Then we set the activate times of instances of tasks according to those paths.

Verify schedulability in single combination : Verifying whether the equation below could be satisfied, where we assume lep is the expiry point which could activate lt:

$$\begin{aligned}\forall x \in [0, \sigma_l] : \exists y \in [e(lt), d(lt)] : \\ srf^{lep}_{l(lt)}(x) + prf^{lep}_{l(lt),-1}(y) - \zeta^{(lep)}_{l(lt),-1} + \sum_{ep_i \in \varepsilon} prf^{ep_i}_{l(lt),x}(x+y) \le x + y\end{aligned} \tag{11}$$

Verify schedulability in all combination : If Equation 11 is satisfied, then we pick other $\varepsilon \in \Phi(other\_st)$ and repeat from setting instances of tasks. Until all ε have been picked, then we move to the next step. Instead, if there exist a situation couldn't satisfy Equation 11, we declare this set of DSTs is non-schedulable, and find a counter example.

Switch to another task : Picking one task with the lowest priority in this set except tasks already passed verification and repeating the procedure from dividing the set of DSTs.

Declared schedulable : After all task in this set of DSTs passed validation, this set is declared as schedulable.

5.2 Integrated Method

We summarize those steps and give two primary algorithms as below. In those two algorithms, τ is a global variable which represents

the set of DSTs. *task_set* is also a global variable which represents the set of all tasks in τ.

Algorithm 1 SCHEDULABILITY

Input:
Output: schedulable or non-schedulable
1: **if** $task\_set = \emptyset$ **then**
2: **return** schedulable
3: **else**
4: $lt \leftarrow GET\_LOWEST\_PRIORITY(task\_set)$
task_set is a global variable
5: **if** $pass = TASK\_SCHEDULABILITY\_ANALYSIS(lt)$ **then**
6: $task\_set \leftarrow task\_set - lt$
7: **return** SCHEDULABILITY()
8: **else**
9: **return** non-schedulable
10: **end if**
11: **end if**

At the begin of Algorithm 1, we implement line 1 and 2 to check whether the algorithm should terminate. Line 4 is corresponding to the second step in section 5.1 which gets the task with the lowest priority. We conceal the implementation details of the function *GET_LOWEST_PRIORITY*.

After getting the target task lt, the function *TASK_VERIFY* in line 5 is the implementation of Theorem 9. According to this theorem, if the function returns *pass*, then the task will always meet its deadline. We will remove lt from the task set *task_set* in line 6 and continue process the algorithm *SCHEDULABILITY*. So whenever the set *task_set* is empty, all tasks have been confirmed to always meet its deadline. Therefore the set of DSTs is schedulable. Instead, if the return result of *TASK_VERIFY* is $not - pass$, that means at least one counter example is detected and the set of DSTs is non-schedulable. We show the detail in Algorithm 2.

Algorithm 2 TASK_SCHEDULABILITY_ANALYSIS

Input: lt
Output: pass or not-pass
1: **if** $\sigma_l \leftarrow BOUNDARY(l(lt)) > hyperperiod$ **then**
2: **return** not-pass
3: **else**
4: **for** $\forall \quad \varepsilon \in \Phi(other\_dsts)$ **do**
5: **for** $\forall \quad x \in [0, \sigma_l]$ **do**
6: **for** $\forall y \in [e(lt), d(lt)]$ **do**
7: **if** $SRF^{lep}_{l(lt)}(x) + PRF^{lep}_{l(lt),-1}(y) - \zeta^{lep}_{l(lt),-1} + \sum_{ep_i \in \varepsilon} PRF^{ep_i}_{l(lt),x}(x+y) > x + y$ **then**
8: **return** not-pass
9: **end if**
10: **end for**
11: **end for**
12: **end for**
13: **return** pass
14: **end if**

Figure 8: An example of DST dst_3, where $t_6 =< 2, 11, 6 >$, $t_7 =< 1, 3, 4 >$.

Algorithm 2 aims to verify whether task lt always meet its deadline. We use *lep* to represent the expiry point which could activate lt. Line 1 and 2 compute the boundary of busy windows and compare with the hyperperiod. The function *BOUNDARY* is an implementation of Lemma 4.3. From line 4 to 7 we implement the Equation 9, which could traverse all situations in scheduling lt. *SRF* and *PRF* represent the suffix request function and the prefix request function respectively. If the task lt could not satisfy Equation 9, the algorithm returns $not - pass$, else returns *pass*.

5.3 Demonstration

We show an example for this method. In this set of DSTs, except the DST dst_1 in Figure 2 and dst_2 in Figure 4a, we add one more DST dst_3 in Figure 8 into this set.

In this case, the set of DSTs $\tau = \{dst_1, dst_2, dst_3\}$. Hyperperiod is calculated as the least common multiple of the duration of three DSTs which equals to 2380 ticks. Then we get the set of tasks, denote as *task_set*:

$$task\_set = Tasks_1 \cup Tasks_2 \cup Tasks_3 = \{t_1, t_2, t_3, t_4, t_5, t_6, t_7\}.$$

Picking a task in *task_set* with the lowest priority, denoting as lt. For example we choose t_3. Task t_3 divides τ into two parts: dst_1 and $\{dst_2, dst_3\}$, which are denoted by *ldst* and *other_dsts* respectively.

After finding lt, we can get the upper bound of busy windows for $l(t_3)$ by Equation 6 and denote it as $\sigma_l = 13$.

The next step, we use $\Phi(\tau)$ to get all combinations of expiry points as $\{\{ep_4, ep_6\}, \{ep_5, ep_6\}\}$. Assuming we pick $\varepsilon = \{ep_4, ep_6\}$ firstly. Then we get the path $\rho_1 = [(ep_4, 0), (ep_5, 3), (ep_4, 14) \dots]$ for dst_2 and $\rho_2 = [(ep_6, 0), (ep_6, 20), (ep_6, 40), \dots]$. According to those paths, we set a series of instances of tasks like: $< 1, 3, 3, 0 >$ and $< 1, 3, 3, 14 >$ of t_4, $< 2, 11, 6, 20 >$ of t_6, $< 1, 3, 4, 20 >$ of t_7 and so on.

Then we verify whether there exists a $y \in (0, d(t_3)]$ for any $x \in [0, \sigma_l]$ could satisfy Equation 11, i.e.,

$$\forall x \in [0, 13] : \exists y \in [2, 9] :$$
$$srf_6^{ep_3}(x) + prf_{6,-1}^{ep_3}(y) - \zeta_{6,-1}^{ep_3} + \sum_{ep_i \in \varepsilon} prf_{6,x}^{ep_i}(x+y) \le x + y.$$

When x = 0, there is y = 9 could satisfy the requirement, when x = 1, y =8 satisfy the requirement, ..., x = 13, y =3 satisfy the requirement. Then we verify other $\varepsilon \in \Phi(other\_dsts)$ until all elements in $\Phi(other\_dsts)$ pass validation.

After verifying schedulability in all combination, we delete t_3 from *task_set*. Then we continue pick another task with the lowest priority in *task_set* and repeat those steps. Finally, the algorithm will stop when *task_set* is empty or a situation could not pass verification.

6 EXPERIMENTAL ANALYSIS

In order to validate the effectiveness of our method, we setup an experiment to compare the result of our method and the execution on the real implemented AUTOSAR OS. In this experiment, we develop a program to generate schedule tables randomly, then implement them on a real-life AUTOSAR while analyzing them by our algorithm. Finally, we compare results to prove the correctness of our method.

6.1 Experiment Setup

We generate a set of schedule tables in seven parameters which are shown as below:

- st_{num}, the number of schedule tables.
- ep_{num}, the maximal number of expiry points that a schedul table could contain.
- $task_{num}$, the maximal number of tasks that an expiry point could activate.
- $Delay$, the delay of an expiry point.
- $e(t)$, the execution time of a task.
- $d(t)$, the deadline of a task.
- δ, the duration of schedule tables.

And the Table 1 gives the ranges of parameters:

st_{num}	ep_{num}	$task_{num}$	$Delay$	$e(t)/d(t)$	$d(t)/\delta$
$[2, 6]$	$[1, 4]$	$[1, 3]$	$[10, 30]$	$[0, 0.16]$	$[0.1, 0.5]$

Table 1: Ranges of parameters to generate set of schedule tables

After generating schedule tables randomly, on the one hand, we transform those schedule tables into DSTs and input them into our method. Then get the result of schedulability. On the other hand, we deploy those schedule tables into an implemented AUTOSAR by a configuration tool and debug the operating system to test the schedulability.

In this experiment, we use an industrial-used AUTOSAR operating system which is developed by iSoft Infrastructure Software Co., Ltd. Our target hardware platform is TC1782 32-Bit Single-Chip Microcontroller and more information about TC1782 in [11].

This industrial-used AUTOSAR OS also provides a matched configuration tool to generate the configuration file and an user interface. We use this tool to generate configuration files follow up with our request and deploy our generated schedule tables into AUTOSAR by modifying the user interface. In order to implement deadline detection, we also setup an alarm for each task, which will expire when task's on deadline. The alarm is assigned to a callback routine which could check the state of the corresponding task. If the state is not suspended, i.e., this task is not finished yet, the callback routine will return false. If the state is suspended, the callback routine will return true. With those alarms, we could find the task which violates deadline as soon as time faults raises.

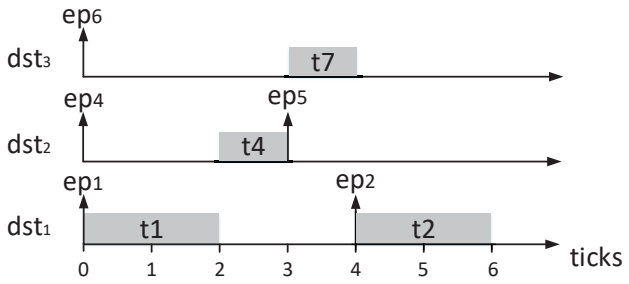


Figure 9: t_7 is activated by ep_6 at 0. But t_7 could not be finished until 4, which makes t_7 violates it's deadline.

6.2 Consistency Analysis

The experiment of comparison shows that the result of our method all compared exactly to an actual running situation on the implemented AUTOSAR. Several examples and their result are synthesized in Table 2, we use $task_{sum}$ represent the number of tasks in each example.

		Schedulability	
st_{num}	$task_{sum}$	our method	industrial OS
3	14	Invalid	Invalid
4	28	Valid	Valid
5	36	Invalid	Invalid
6	27	Invalid	Invalid

Table 2: Comparison of schedulability analysis result

If the schedulability result obtained by our method is non-schedulable, an amending advice is given. Such as the example given in Section 5.3, when we test task t_7, Equation 9 could not be satisfied when $x = 0$ at the scenario $\varepsilon = ep_4, ep_6$. Then we try to widen it's deadline from 3 ticks to 4 ticks and test t_7 repeatedly. Until it pass validation, cf. Figure 9. Then we modify any tasks could not pass validation. Finally, we get an amended example, which is schedulable and it is schedulable on implemented AUTOSAR as well.

7 CONCLUSION

In this paper, we have abstracted the behaviors of schedule tables as formal models and proposed an algorithm based on the models. Our algorithm is to analyze the schedulability of AUTOSAR OS by travelling all the possible offsets between schedule tables. Moreover, we conduct an experiment, which runs a set of testbench on an industrial AUTOSAR OS and the experiment respectively to check the effectiveness of our method. Through the comparison result, we believe that our method could perform well in assisting developers in designing schedule tables.

This work has gotten a promising result in analyzing AUTOSAR OS on uniprocessor. Since AUTOSAR specification has introduced multicore operating systems, we are considering to work on schedulability analysis of them. Besides, schedulability analysis for extended tasks, which are allowed to synchronize through setting

and waiting events, is another interesting topic we would like to pursue.

REFERENCES

[1] Saoussen Anssi, Sara Tucci Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. 2011. Enabling Scheduling Analysis for AUTOSAR Systems. In *ISORC 2011, Newport Beach, California, USA, 28-31 March 2011*. 152–159. https://doi.org/10.1109/ISORC.2011.28

[2] Neil C. Audsley, Alan Burns, Mike M. Richardson, Ken Tindell, and Andy J. Wellings. 1993. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal* 8, 5 (1993), 284–292. https://doi.org/10.1049/sej.1993.0034

[3] AUTOSAR. 2013. Specification of AUTOSAR Operating System.

[4] Theodore P. Baker and Alan C. Shaw. 1989. The Cyclic Executive Model and Ada. *Real-Time Systems* 1, 1 (1989), 7–25. https://doi.org/10.1007/BF02341919

[5] Sanjoy Baruah. 2016. Schedulability Analysis for a General Model of Mixed-Criticality Recurrent Real-Time Tasks. In *2016 IEEE Real-Time Systems Symposium, RTSS 2016, Porto, Portugal, November 29 - December 2, 2016*. 25–34. https://doi.org/10.1109/RTSS.2016.012

[6] Enrico Bini and Claire Pagetti (Eds.). 2017. *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017*. ACM. http://dl.acm.org/citation.cfm?id=3139258

[7] Alan Burns and Robert I. Davis. 2017. A Survey of Research into Mixed Criticality Systems. *ACM Comput. Surv.* 50, 6 (2017), 82:1–82:37. https://doi.org/10.1145/3131347

[8] Michel Goossens, S. P. Rahtz, Ross Moore, and Robert S. Sutor. 1999. *The Latex Web Companion: Integrating TEX, HTML, and XML* (1st ed.). Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] Leo Hatvani and Reinder J. Bril. 2015. Schedulability using native non-preemptive groups on an AUTOSAR/OSEK platform. In *ETFA 2015, Luxembourg, September 8-11, 2015*. 1–8. https://doi.org/10.1109/ETFA.2015.7301449

[10] Yanhong Huang, João F. Ferreira, Guanhua He, Shengchao Qin, and Jifeng He. 2013. Deadline Analysis of AUTOSAR OS Periodic Tasks in the Presence of Interrupts. In *ICFEM 2013, October 29 - November 1, 2013, Proceedings*. 165–181. https://doi.org/10.1007/978-3-642-41202-8_12

[11] INFINEON. 2011. *TC1782 32-Bit Single-Chip Microcontroller*. Technical Report. Infineon Technologies.

[12] Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang (Eds.). 2017. *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*. Lecture Notes in Computer Science, Vol. 10606. Springer. https://doi.org/10.1007/978-3-319-69483-2

[13] John P. Lehoczky. 1990. Fixed Priority Scheduling of Periodic Task Sets with Arbitrary Deadlines. In *Proceedings of the Real-Time Systems Symposium - 1990, Lake Buena Vista, Florida, USA, December 1990*. 201–209. https://doi.org/10.1109/REAL.1990.128748

[14] C. L. Liu and James W. Layland. 1973. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM* 20, 1 (1973), 46–61. https://doi.org/10.1145/321738.321743

[15] OSEK. 2005. OSEK/VDX Operating System Specification 2.2.3.

[16] Yunhui Peng, Yanhong Huang, Ting Su, and Jian Guo. 2013. Modeling and Verification of AUTOSAR OS and EMS Application. In *TASE 2013, 1-3 July 2013, Birmingham, UK*. 37–44. https://doi.org/10.1109/TASE.2013.13

[17] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. 2011. The Digraph Real-Time Task Model. In *RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*. 71–80. https://doi.org/10.1109/RTAS.2011.15

[18] Martin Stigge and Wang Yi. 2015. Combinatorial abstraction refinement for feasibility analysis of static priorities. *Real-Time Systems* 51, 6 (2015), 639–674. https://doi.org/10.1007/s11241-015-9220-5

[19] Jiantao Wang, Kam-yiu Lam, Song Han, Sang Hyuk Son, and Aloysius K. Mok. 2013. An effective fixed priority co-scheduling algorithm for periodic update and application transactions. *Computing* 95, 10-11 (2013), 993–1018. https://doi.org/10.1007/s00607-012-0242-8

[20] Wenhao Wang, Fabrice Camut, and Benoit Miramond. 2016. Generation of schedule tables on multi-core systems for AUTOSAR applications. In *(DASIP), Rennes, France, October 12-14, 2016*. 191–198. https://doi.org/10.1109/DASIP.2016.7853818

[21] Hyunmin Yoon and Minsoo Ryu. 2014. Real-time priority assignment for autosar-based systems with time-driven synchronization. In *RACS 2014, Towson, Maryland, USA, October 5-8, 2014*. 297–302. https://doi.org/10.1145/2663761.2664188

[22] Qingling Zhao, Zonghua Gu, and Haibo Zeng. 2017. Design optimization for AUTOSAR models with preemption thresholds and mixed-criticality scheduling. *Journal of Systems Architecture - Embedded Systems Design* 72 (2017), 61–68. https://doi.org/10.1016/j.sysarc.2016.08.003

[23] Longfei Zhu, Peng Liu, Jianqi Shi, Zheng Wang, and Huibiao Zhu. 2013. A Timing Verification Framework for AUTOSAR OS Component Development Based on Real-Time Maude. In *TASE 2013, 1-3 July 2013, Birmingham, UK*. 29–36. https://doi.org/10.1109/TASE.2013.12