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.

1 INTRODUCTION

Formal methods have been widely and successfully applied to functional properties verification. But 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 [1] 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. [2] 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. [3] give transaction model with three parameters, a fixed priority, 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 [4] analyze schedulability in a general 3-parameter sporadic task model [5] 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 [7] for corrective scheduling to meet stringent certification requirement. C.Deutschbein et al. [8] present the problem of constructing cyclic executive upon multiprocessors.

AUTOSAR (AUTomotive Open System ARchitecture) [9] is an open and standardized layered software architecture and interfaces. The integration of software modules from different vendors forms the complete architecture. Researchers working on AUTOSAR OS are rich in progressive and proactive. There have been many studies on analyzing AUTOSAR OS specification [10]. For schedulability of the AUTOSAR OS, both Zhao et al. [11] and Hatvani et al. [12] 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. [13] 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 [14] 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 [9], which is an important mechanism in AUTOSAR OS. For automobile being a high safety-critical real time system, a practical approach [7] 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 finish in deadline. We propose an algorithm to travel all of the possible schedule situations so that each situation can exactly complete task execution. A time interval between a task releasing and its deadline is checked to confirm 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 [15]. 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

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. 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. A suspended task can be put into ready queue by a system service which will transfer the task into *ready* state. A task in *ready* state is waiting for allocation of the processor. At most one task can be in *running* state at the same time. 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.

The schedule table mechanism is a new concept introduced in AUTOSAR which could statically define the pattern of tasks activation. Each schedule table 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. The process of schedule tables can be manipulated as users call system services at any time.

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 stop 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 an 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.

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 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 theories have been presented. We can categorize those 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

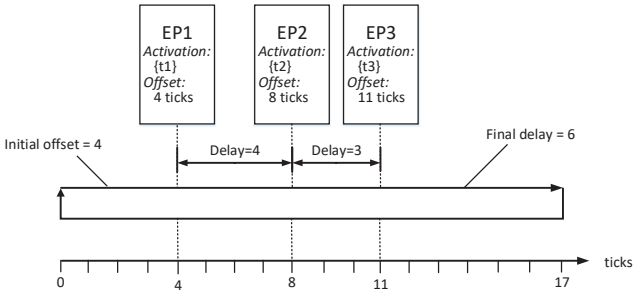


Figure 1: An example repeating scheduel table cotaining three expiry points, and t_1, t_2, t_3 is three tasks.

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 [16].

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) [17]. 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) in AUTOSAR, 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. In order to resolve this, we decide to define a proper formal model to describe the behavior of schedule tables, then propose an algorithm to analyse the schedulability of schedule tables by the formal model. In this section, we state our model firstly.

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. So AUTOSAR provide various features to satisfy requirements. Among them, our work only focus on several characteristics:

(1) Tasks are allowed to share the same priority level
(2) Lower number for higher priority.
(3) We take no account of extended tasks and events.
(4) All tasks are full preemptible.
(5) The first scheduling policy is highest priority first, the second policy is first come first served.
(6) 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 of actions $Act(ep) \in 2^{Tasks}$ 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 equals to the final delay plus the initial offset.

Note that we conceal the initial offset in DST. Because during the running 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 represented by a pair (ep_i, r_i), where r_i is the expiry time. Naturally to point out that the 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 describe 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 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 is calculated by:

$$\iota(\rho) = r_k - r_1.$$

3.3 Example Model

We model the schedule table given before as Figure 2 and give the tuple 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.
- $Tasks_1 = \{t_1,\ t_2,\ t_3\}$ 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 $Delay_1(ep_1) = 4$, $Delay_1(ep_2) = 3$, $Delay_1(ep_3) = 10$.

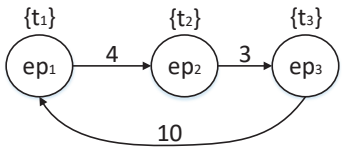


Figure 2: An example of DST for st_1, where $t_1 =< 2, 4, 2 >$, $t_2 =< 2, 3, 1 >$ $t_3 =< 2, 9, 6 >$.

4 DEFINITION AND NOTATION

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

4.1 Computational Requirement

The main idea of our analysis method is to check whether the time interval between a 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 time of tasks could interrupt?

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 interrupted by tasks with a lower priority than $l(t)$, or tasks which have priority $l(t)$ but activate time is larger. Based on this, we introduce the concept of computational requirement to accumulate the execution time on 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 $\omega^{(ep,r)}_{l(t),r(t)}$ to accumulate the execution time of tasks in $Act(ep)$. If $r \leq r(t)$, we remove those tasks which have a lower priority than $l(t)$ from $\omega^{(ep,r)}_{l(t),r(t)}$; when $r > r(t)$, we remove tasks which have priority lower than or equal to $l(t)$ from $\omega^{(ep,r)}_{l(t),r(t)}$:

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

where

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

After proposing the computational requirement function, the execution time during a path can be computed as well. 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):

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

where we use ρ_i to represent the ith instance of expiry point on the path ρ.

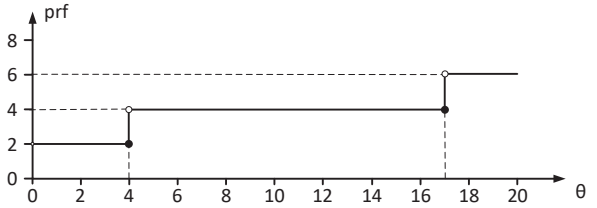


Figure 3: An example for prefix request function of $prf_{4,0}^{ep_1}(\theta)$ according to dst_1 in Figure 2.

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 extend the prefix request function $prf_{l,r}^{ep}(\theta)$ from [18] 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 $\iota(\rho) < \theta$.

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

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 $0 < \theta \leq 4$, the path must be $[(ep_1, 0)]$, which have computational requirement as 2. When $4 < \theta \leq 7$, the path $[(ep_1, 0), (ep_2, 4)]$ makes the maximum computational requirement 4. When $7 < \theta \leq 17$, the path ρ also can be $[(ep_1, 0), (ep_2, 4), (ep_3, 7)]$. But in ep_3, the priority of t_3 is lower than 4, so the prefix request function wouldn't accumulate it. The function will keep returning 4 until θ exceeds 17.

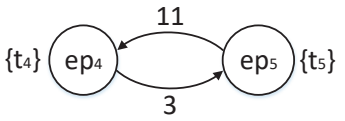
A DST has as many prefix request functions as expiry points in this DST, those prefix request functions can be synthesized as one. We use the maximal request function to integrate those prefix request functions.

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

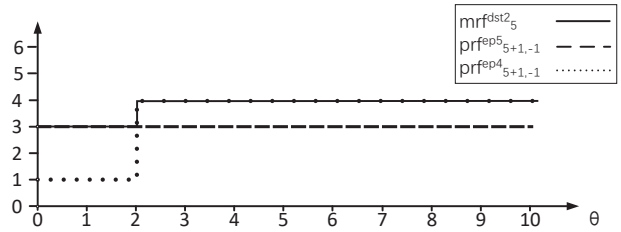
$$mrf_l^{dst}(\theta) = max\{prf_{l+1,-1}^{ep}(\theta) | ep \in Node\} \quad (4)$$

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. Notice that in any point of θ, there are at least one prefix request function offer the value of the maximal request function.

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 which is calculated by the suffix request function [18]. A slightly different between srf and prf is that the former ignores the expire time.



(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.

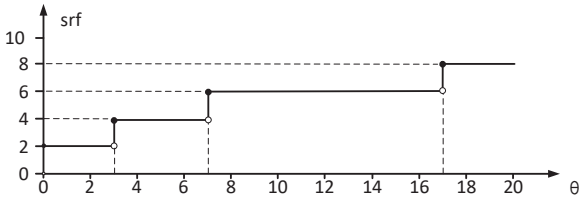


Figure 5: Suffix request function of $srf_4^{ep_3}(\theta)$ according to DST_1 in Figure 2.

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.

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

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 $srf_4^{ep_3}(\theta)$. Figure 5 illustrates $srf_4^{ep_3}(\theta)$. When θ start from 0, we calculate the computational requirement of path $[(ep_3, r_3)]$ and get 2 as the result. When θ reaches 3, path $[(ep_3, r_3)]$ and $[(ep_2, r_2), (ep_3, r_3)]$ meet the requirement and we choose the maximum value of computational requirement as 4. The rest may be deduced by analogy.

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 [19].

Busy window for priority level l is a time interval where 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 least one expiry point expire at the point of a busy window start. So the computation of the upper bound of busy windows is all about finding those expiry points which is named as start expiry points.

If there is a θ satisfies Equation 6, then for each maximal request function, at least one prefix request function offer its value at θ. We pick any one prefix request function from a maximal request function, then a set of expiry points is obtained as superscript of those prefix request functions. 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 and get an expiry point ep' in dst' as start expiry point. But there is another expiry point ep'' in dst' which could make a longer busy window. Because the prefix request function of ep'' 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.,

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

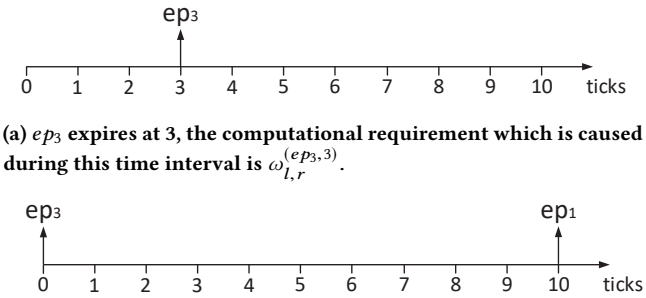
where $E = \sum_{dst \in \tau - d1} mrf_l^{dst}(\theta)$.

So when ep'' expires with other expiry points, the busy window 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 DSTs 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 as $lcm(\delta_1, \delta_2, \ldots, \delta_n)$.

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



(a) ep_3 expires at 3, the computational requirement which is caused during 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 during scenario (b) is larger than it which is caused in scenario (a), i.e., the time interval in scenario (b) dominates it in scenario (a).

Proof. The scheduling process of a set of repeating schedule tables is cyclic. If any busy window exceeds the hyperperiod, the processor cannot finish those tasks which are activated in this hyperperiod. In the next hyperperiod, except tasks which is activated in this hyperperiod, the processor also has to execute those tasks unfinished. 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 expiry points to represent offsets.

Because our schedulability analysis method always focuses on the execution time, so we measure a time interval with computational requirement. If computational requirement which is caused during a time interval is lager than or equal to other intervals, 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. Since a task could always keep time property if it keeps in the worst case, so we only need to consider the time interval which cannot be dominated. 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.

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

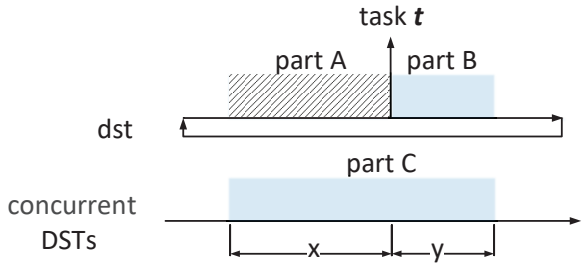


Figure 7: An illustration of schedulability analysis of task t. There are three parts may affect the execution of t: A) interruption from dst before task t is activated, B) interruption from dst after task t is activated, C) interruption from other DSTs.

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, schedule situations before activation also make an impact. Therefore, we extend the time interval towards to before. 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 not be more 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 task come from three parts: 1) tasks which are activated before this task from the same DST, 2) tasks which are activated after this task from the same DST, 3) tasks from other DSTs, cf. Figure 7. The execution time of the three parts change base on x, y and offsets between DSTs. After sorting out these three parts, we check whether the extended time interval could accommodate execution time of the task and interrupt time. Notice that we use prefix request function to calculate part B, but the expiry time of the first instance of expiry point in part B is x instead of 0.

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:*

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

Proof. Firstly, we state that $\omega^{(ep,x)}_{l(t),-1}$ is the overlap between $srf^{ep}_{l(t)}$ and $prf^{ep}_{l(t),-1}$. Now, 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 $y = 0, 1, 2, \ldots, d(t)$.

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

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 Section 4, we have introduced some definition about schedulability analysis. In this section, we synthesize those definitions into an integrated method.

5.1 Stages of Method

When a set of DSTs $\tau = \{dst_1, dst_2, \ldots dst_n\}$ is given, the process of analyzing schedulability can be divided into stages as follows:

(1) : Picking one task with the lowest priority from $Task_1 \cup Task_2 \cup \cdots \cup Task_n$ and denoting it as lt.
(2) : dividing τ into two parts: i) the one DST could activate lt, ii) the other DSTs. We denote them by $ldst$ and $other\_dsts$ respectively.
(3) : Calculating the upper bound of busy windows for $l(lt)$ which is denoted by σ_l. The set of DSTs is non-schedulable if σ_l exceed hyperperiod, .
(4) : Picking one element ε from $\Phi(other\_dsts)$, then we structure a path for every DSTs in $other\_dsts$: expire time of expiry points in ε are setted as 0. The rest are structured according to delays of expiry points. We set the activate times of instances of tasks as well.
(5) : Verifying whether the equation below could be satisfied, where lep is the expiry point which could activate lt:

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

(6) : If Equation 11 is satisfied, then we pick other $\varepsilon \in \Phi(other\_st)$ and repeat from step 4, until every elements in $\Phi(other\_st)$ have been picked, then we move to the next step. Instead, if there exist a situation makes Equation 11 false, we declare this set of DSTs is non-schedulable.

(7) : Picking one task with the lowest priority in this set except tasks already passed verification. Then repeating the procedure from step 2.
(8) : If all tasks pass validation, declaring this set is schedulable.

5.2 Integrated Method

We summarize those steps and give two primary algorithms as below. We assume 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 step 2 which aims to get the task with the lowest priority and divide τ. We conceal the details of this function. The function *TASK_VERIFY* verifies whether a task always meets its deadline. If this function returns *pass*, then we will remove this task from *task_set* and continue run this algorithm. *task_set* is empty means 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 τ is non-schedulable. The detail of *TASK_VERIFY* is shown in Algorithm 2.

Algorithm 2 aims to verify whether task lt always meet its deadline. We use *lep* to represent the expiry point which could activate lt. Firstly, we get the boundary of busy windows and compare with the hyperperiod. Line 4 to 7 is an implementment of Theorem 4.10. *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

As an example for this method, we construct a set of DSTs by uniting dst_1 in Figure 2, dst_2 in Figure 4a and one more DST dst_3 in Figure 8. In this case, the set of DSTs $\tau = \{dst_1, dst_2, dst_3\}$. 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, assuming 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*

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** $SRF^{lep}_{l(lt)}(x) + PRF^{lep}_{l(lt),-1}(y) - \omega^{(lep,x)}_{l(lt),-1} + \sum_{ep_i \in \varepsilon} PRF^{ep_i}_{l(lt),x}(x+y) > x + y$ **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 >.

respectively. Then, we 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 in *other_dsts* as $\{\{ep_4, ep_6\}, \{ep_5, ep_6\}\}$. Assuming we pick $\varepsilon = \{ep_4, ep_6\}$ firstly. Then we build path $\rho_1 = [(ep_4, 0), (ep_5, 3), (ep_4, 14) \dots]$ for dst_2 and $\rho_2 = [(ep_6, 0), (ep_6, 20), (ep_6, 40), \dots]$ for dst_3. 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 a $y \in (0, d(t_3)]$ exists for any $x \in [0, \sigma_l]$ to satisfy Equation 11, i.e.,

$$\forall x \in [0, 13] : \exists y \in [2, 9] :$$
$$srf^{ep_3}_6(x) + prf^{ep_3}_{6,-1}(y) - \omega^{(ep_3,x)}_{6,-1} + \sum_{ep_i \in \varepsilon} prf^{ep_i}_{6,x}(x+y) \le x + y.$$

When x = 0, y = 9 satisfy the requirement, when x = 1, y =8 satisfy the requirement, ..., x = 13, y =3 satisfy the requirement. Then we verify other ε until all elements in $\Phi(other\_dsts)$ pass validation and 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 an implemented AUTOSAR OS. In this experiment, we develop a program to generate schedule tables randomly and implement them on the AUTOSAR while analyzing them by our algorithm. 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 [15].

6.1 Experiment Setup

We generate schedule tables subject to seven parameters as below. Table 1 shows the ranges of parameters.

- st_{num}, the number of schedule tables.
- ep_{num}, the maximal number of expiry points in a schedule table.
- $task_{num}$, the maximal number of tasks in an expiry point.
- $Delay$, 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.

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

After generating schedule tables randomly, we transform them into DSTs and consider them as the input of our method. Meanwhile we deploy those schedule tables into an implemented AUTOSAR by a configuration tool and test the schedulability.

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 $task_{sum}$ to represent the amount of tasks.

		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 will be given. For example, when we test task t_7, Equation 9 could not be satisfied when $x = 0$ and $\varepsilon = ep_4, ep_6$. Then we try to widen it's deadline from 3 ticks to 4 and test t_7 again. Until it pass validation. 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] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 20(1):46–61, 1973.

[2] Martin Stigge, Pontus Ekberg, Nan Guan, and Wang Yi. The digraph real-time task model. In *RTAS 2011, Chicago, Illinois, USA, 11-14 April 2011*, pages 71–80, 2011.

[3] Jiantao Wang, Kam-yiu Lam, Song Han, Sang Hyuk Son, and Aloysius K. Mok. An effective fixed priority co-scheduling algorithm for periodic update and application transactions. *Computing*, 95(10-11):993–1018, 2013.

[4] Sanjoy Baruah. 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*, pages 25–34, 2016.

[5] Alan Burns and Robert I. Davis. A survey of research into mixed criticality systems. *ACM Comput. Surv.*, 50(6):82:1–82:37, 2017.

[6] Enrico Bini and Claire Pagetti, editors. *Proceedings of the 25th International Conference on Real-Time Networks and Systems, RTNS 2017, Grenoble, France, October 04 - 06, 2017*. ACM, 2017.

[7] Theodore P. Baker and Alan C. Shaw. The cyclic executive model and ada. *Real-Time Systems*, 1(1):7–25, 1989.

[8] Kim Guldstrand Larsen, Oleg Sokolsky, and Ji Wang, editors. *Dependable Software Engineering. Theories, Tools, and Applications - Third International Symposium, SETTA 2017, Changsha, China, October 23-25, 2017, Proceedings*, volume 10606 of *Lecture Notes in Computer Science*. Springer, 2017.

[9] AUTOSAR. Specification of autosar operating system, February 2013.

[10] Saoussen Anssi, Sara Tucci Piergiovanni, Stefan Kuntz, Sébastien Gérard, and François Terrier. Enabling scheduling analysis for AUTOSAR systems. In *ISORC 2011, Newport Beach, California, USA, 28-31 March 2011*, pages 152–159, 2011.

[11] Qingling Zhao, Zonghua Gu, and Haibo Zeng. Design optimization for AUTOSAR models with preemption thresholds and mixed-criticality scheduling. *Journal of Systems Architecture - Embedded Systems Design*, 72:61–68, 2017.

[12] Leo Hatvani and Reinder J. Bril. Schedulability using native non-preemptive groups on an AUTOSAR/OSEK platform. In *ETFA 2015, Luxembourg, September 8-11, 2015*, pages 1–8, 2015.

[13] Hyunmin Yoon and Minsoo Ryu. Real-time priority assignment for autosar-based systems with time-driven synchronization. In *RACS 2014, Towson, Maryland, USA, October 5-8, 2014*, pages 297–302, 2014.

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

[15] INFINEON. Tc1782 32-bit single-chip microcontroller. Technical report, Infineon Technologies, 2011.

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

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

[18] Martin Stigge and Wang Yi. Combinatorial abstraction refinement for feasibility analysis of static priorities. *Real-Time Systems*, 51(6):639–674, 2015.

[19] John P. Lehoczky. 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*, pages 201–209, 1990.