02229 E22 Optimization Challenge: Configuring ADAS Applications with Time-Triggered and Event-Triggered Tasks

Silviu S. Craciunas
TTTech Computertechnik AG, Austria

Paul Pop
DTU Compute, Technical University of Denmark (DTU)

Abstract

This document presents the optimization challenge for the course 02229 E22 Systems Optimization. Please also see the "Project" folder in OneDrive shared drive of the course. Document last updated: Sept. 16.

2012 ACM Subject Classification Computer systems organization → Dependable and fault-tolerant systems and networks

Keywords and phrases time-triggered, event-triggered, scheduling, real-time, combinatorial optimization

1 Introduction

Modern vehicles integrate a growing number of complex functions, commonly known as Advanced Driver Assistance Systems (ADAS), which provide driver assistance, e.g. automated or assisted parking, lane changing and emergency brake assistance and even fully autonomous driving. The growing trend to migrate more and more of ADAS functions from hardware to software allows modularization within an integrated hardware platform that can be cooperatively used and centrally managed [12].

The fusion of multiple software functions into the same hardware platform has multiple advantages, like reusability and portability, but also several challenges, especially in terms of real-time, testing and safety. This paradigm enables the integration of functions of different criticality levels in a composable manner with guaranteed temporal and spatial isolation such that they can coexist on the same platform without sacrificing real-time capabilities. However, this mixed-criticality paradigm applied to the automotive domain requires new concepts in terms of safety-critical temporal and spatial isolation, new scheduling results and configurations tools [6, 12].

Hardware architecture: Generally, ADAS platforms are composed of heterogeneous multi-core CPUs and Systems-on-chip (SoCs) of different performance and safety levels, that are interconnected by a (real-time) communication backbone. The work in this project targets a *single computation element*, e.g., a core in a multi-core SoC. However, the algorithms developed are applicable to multiple cores, see [10] for the issues when assigning functions to several cores of an ADAS platform[1].

Software architecture (scheduling policy): *Scheduling* is a method for assigning resources for completing activities (e.g., tasks or messages). The scheduling is carried out by a *scheduler* that implements a policy to achieve goals such as minimizing latency, minimizing wait-time, and maximizing throughput. Most schedulers use policies that are not "real-time";

[1] The work in [10] is an extended version of the optimization challenge proposed in 2019. The paper has received the "Best Paper Award" at the ETFA 2020 conference.

thus, they are not suitable for real-time applications, see chapter 1 in [3] for the basic concepts of real-time computing.

The automotive suppliers will select, based on their own requirements, the scheduling policy to be used in the different components of an ADAS platform. The main approaches to scheduling are: (i) *time-triggered* (TT), such as non-preemptive static cyclic scheduling, also called "timeline scheduling", and (ii) *event-triggered* (ET), such as preemptive periodic scheduling, e.g., using priorities that are fixed, e.g., Rate Monotonic (RM), or dynamic, e.g., Earliest Deadline First (EDF), see chapter 4 in [3] for a discussion of periodic real-time scheduling.

There has been a long debate in the real-time and embedded systems communities concerning the advantages of each approach [9]. Timeline scheduling has been recommended for safety-critical applications. In this method, the scheduler repeats a static schedule determined at design-time. The preemptive periodic scheduling approaches generate the schedule at runtime based on arrival times of periodic activities and their priorities [3]. Integrating event-triggered activities, which are often "aperiodic" or "sporadic", can be done via "periodic servers", see chapters 5 and 6 in [3] for priority servers.

In this project, we consider that the TT tasks on the given core are scheduled with *timeline scheduling*. Statically scheduled systems, which are a natural implementation for TT tasks, have many benefits in terms of predictability, stability, compositionality, and determinism. However, it is difficult to integrate sporadic event-triggered (ET) tasks into a time-triggered system. Hence, we assume that the ET tasks are integrated with the TT tasks via "polling servers", see Section 5.3 in [3] for details.

Application model: Modern safety-critical systems benefit most from combining the two paradigms, allowing a time-triggered system to be flexible enough to respond to sporadic events when needed. For time-triggered systems, where the schedule table is predefined and statically configured at design time, sporadic event-triggered (ET) tasks can only be handled within specially allocated slots or when time-triggered (TT) tasks finish their execution earlier than their worst-case assumption. While there is a significant body of work (c.f.[11] for an extensive survey) concerning pure time-triggered schedule generation, which is an NP-complete problem, most of the methods do not consider the schedulability of sporadic ET tasks. In this project, we consider that the applications are modeled using a combination of TT and ET task sets, which are given as an input to the optimization problem.

2 Problem Formulation

Given: The combinatorial optimization problem you have to solve in this course can be formulated as follows. As an input you are given: (1) An application model consisting of a set of TT tasks and a set of ET tasks, see Section 3. TT tasks are periodic. *Periodic tasks* consist of an infinite sequence of identical activities, called instances or *jobs*, that are regularly activated with a constant period. For each periodic TT task we know its period, worst-case execution time (WCET) and deadline. ET tasks are *sporadic*, i.e., consecutive jobs are separated by a minimum inter-arrival time, which can be seen as a "minimum period". For each ET task we know the minimum inter-arrival time, WCET and deadline, as well as its priority. See chapters 2 and 4 in [3] for details on the periodic and sporadic task models. (2) An architecture model consisting of one core that schedules TT tasks with *timeline scheduling* and ET tasks with *polling servers*, which are themselves TT tasks. All tasks are considered *preemptable*, i.e., the schedule can be constructed such that they are stopped to allowed another task to run, and they are restarted afterwards to complete.

Determine: You will have to design and implement an optimization algorithm that determines an optimized solution which consists of the following: (i) The number of polling servers, which then become extra TT tasks. (ii) For each polling server (task), the period, budget, and deadline, as well as (iii) which sub-sets of ET tasks are handled within the respective polling servers. Moreover, the TT tasks also need to be schedulable, i.e., given the found polling tasks (and their parameters), it shall be possible to find (iv) a TT schedule such that also the TT tasks are schedulable.

Such that: The solution should be optimized such that: (a) Constraints: both the TT and ET tasks are schedulable, i.e., they complete before their deadlines; and (b) Optimization objective: the average *worst-case response times* (WCRT) of all tasks (TT and ET) is minimized. The WCRT is defined as the time it takes for a task to complete once it has been "released" (i.e., also known as its "arrival time", when it is ready for execution), considering potential interruptions by other tasks.

To implement the checking of the constraints we suggest to use the algorithms from Section 4. For example, Algorithm 2 checks if a set of ET tasks handled by a specific polling server are schedulable and Algorithm 1 can be be used to determine a static schedule for the TT tasks. Note that given a static schedule, it is easy to check if a TT task completes before its deadline.

The problem can be extended in several ways.

Extension 1: Before creating a schedule table using Algorithm 1, you could check if the set of TT tasks is schedulable by using the Processor Demand Criterion from Section 4.6 in [3]. Note that for the case when all deadlines are equal to the periods then the simple utilization based test for EDF can be used, i.e., that the utilization of the task set should be below or equal to 100%.

Extension 2: The test cases contain the priorities of the ET tasks as inputs. However, when having multiple polling tasks handling different subsets of ET tasks, it may be beneficial to re-assign priorities within each individual ET task sub-set. Hence, a further optimization is to disregard the assigned priorities from the input files and also determine the priorities of the ET tasks such that schedulability is improved.

Extension 3: In Algorithm 2, we check whether the deadline is missed for any ET tasks in the given input set. However, there is an optimization that can be employed by checking the schedulability of selected ET tasks using the so-called external points method from [1]. As an optional assignment, you can implement the external points method to improve the runtime of the schedulability check of Algorithm 2.

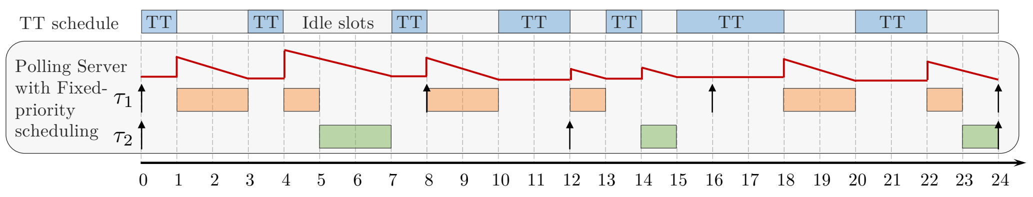


Figure 1 Static schedule table with TT and ET tasks; the ET tasks are handled with "fixed-priority scheduling" in idle slots.

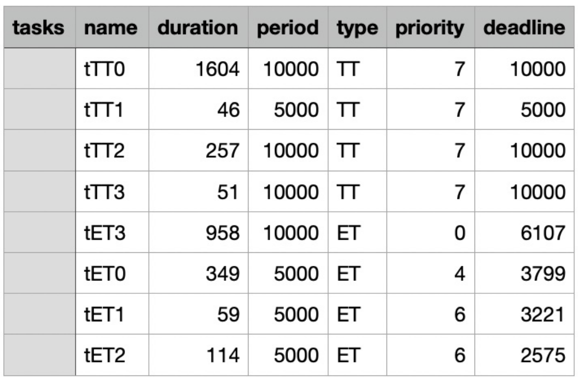
3 Application model

We assume that TT tasks are scheduled based on an offline generated static schedule table (timeline scheduling).In the idle slots of the schedule table there are several options on how to handle the ET tasks. Figure 1 shows a solution where we have a 2^{nd}-level preemptive scheduler based on fixed priorities to handle the ET tasks, similar to "background scheduling" from Section 5.2 in [3]. In this project, we consider that we use polling servers [13] to handle the ET tasks. The polling servers are TT tasks that have to be scheduled together with the other TT tasks.

We denote the set of TT and ET tasks with $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$, respectively. A TT or ET task τ_i is defined by the tuple $(p_i,\, C_i,\, T_i,\, D_i)$ with C_i denoting the computation time, p_i is the task priority, and D_i the relative deadline of the task. For TT tasks, T_i represents the period, while for ET tasks, where we assume a sporadic model, it describes the minimal inter-arrival distance (MIT). For this challenge, both TT and ET tasks have a constrained-deadline model ($T_i \leqslant D_i$). For convenience, we say that all TT tasks share the same (highest) priority. Event-triggered tasks, having a lower priority than TT tasks, are indexed in the order of their relative priority, i.e., τ_i has a higher priority than τ_j ($p_i < p_j$) implies $i < j$, but several tasks may have the same priority ($p_i = p_j$) in which case they are selected in FIFO order.

The timeline of scheduling decisions is divided into equal segments by the microtick mt, representing the smallest scheduling granularity for tasks [7]. Usually, the granularity of the timeline is in the range of hundreds of microseconds to a few milliseconds; however, we do not assume any lower value here as the granularity in, e.g., embedded devices with custom runtime systems can go down to the order of microseconds (e.g. [4]). In the following, we assume that any D, C, T are multiples of mt. A schedule for a finite set of tasks $\mathcal{T} = \mathcal{T}^{TT} \cup \mathcal{T}^{ET}$ is a partial function $\sigma : \mathbb{N} \hookrightarrow \mathcal{T}$ from the time domain to the set of tasks, that assigns to each interval $[t \cdot mt, (t+1) \cdot mt)$ defined by the microtick granularity a task that is running in that time interval. We assume that each schedule σ repeats after a certain time period called the schedule cycle, which is equal to the hyperperiod of the system, defined by $T = lcm_{\tau_i \in \mathcal{T}^{TT}} \{T_i\}$, where lcm is the least common multiple. Furthermore, we assume that the tasks from both sets $\mathcal{T}^{TT}$ and $\mathcal{T}^{ET}$ are scheduled on a single-core CPU that is not overloaded, i.e., $U \leqslant 1$, and hence $\sigma(t)$ is uniquely defined for each point on the microtick timeline.

For the test-cases, we extended the task set generator from [5], to create task sets



tasks	name	duration	period	type	priority	deadline
	tTT0	1604	10000	TT	7	10000
	tTT1	46	5000	TT	7	5000
	tTT2	257	10000	TT	7	10000
	tTT3	51	10000	TT	7	10000
	tET3	958	10000	ET	0	6107
	tET0	349	5000	ET	4	3799
	tET1	59	5000	ET	6	3221
	tET2	114	5000	ET	6	2575

Figure 2 Test-case format

containing TT and ET tasks with a deadline-monotonic priority assignment between 0 and 6 for ET tasks and a priority of 7 for TT tasks. All generated task sets are schedulable if the ET tasks are considered as TT tasks and statically scheduled. Each test file is a comma-separated (.csv) file containing the list of TT and ET tasks (c.f. Figure 2). Each task has a unique name, a duration, a deadline, and a period (in units of $10\mu s$). The type describes whether it is a time-triggered (TT) or event-triggered (ET) task. The priority for TT tasks is always 7 as the highest priority, while for ET tasks it can be $0-6$ with 0 being the lowest priority. All use-cases have a 10 μs microtick, 30 TT and 20 ET tasks per task set with periods selected from the set $\{20, 30, 40\}ms$ ($T = 120ms$), and 100 task sets per test case. For the constrained ET deadline test cases, D_i is uniformly selected in the upper half of the interval $[C_i, T_i]$. We vary the utilization of both ET (u^{ET}) and TT (u^{TT}) task sets $u^k \in \{0.1, 0.2, .., 0.7\}$ such that $u^{TT} + u^{ET} \leqslant 0.9$ resulting in 34 tuples (u^{TT}, u^{ET}).

4 TT schedule construction and ET schedulability test

4.1 EDF simulation for determining the schedule of TT tasks

Algorithm 1 Simulation of TT tasks using EDF

```
   Data: Set of TT tasks $\mathcal{T}^{TT}$ and their properties
   Result: TT schedule table $\mathcal{S}$ and WCRTs of TT tasks
 1 $T \leftarrow$ Least Common Multiple (LCM) of TT task periods;
 2 $ready\_list \leftarrow$ List of ready jobs that are ready to be executed;
 3 $\mathcal{S} \leftarrow$ Empty TT schedule table S;
   /* All tasks are released at time zero */
 4 for $\tau_i \in \mathcal{T}^{TT}$ do
 5     $ready\_list \leftarrow \tau_i$
 6 sort(ready_list) on earliest deadline;
   /* simulate */
 7 for $cycle = 1$ to $n$ with microtick $mt$ increments do
       /* Check if there are any jobs need to be included in the ready_list */
 8     ready_list ← get_ready($\mathcal{T}^{TT}$, cycle);
       /* The job with earliest deadline gets the CPU for the next $mt$ */
 9     current_job ← highest_priority(ready_list);
10     Update($\mathcal{S}$, current_job);
       /* Decrement the execution time of the job */
11     current_job.$C_i$ ← current_job.$C_i$ − $mt$;
       /* Check if the job has finished */
12     if $current\_job.C_i == 0$ then
           /* Calculate the response time of the finished job */
13         current_job.response ← cycle − current_job.start_time;
           /* Update the WCRT if larger than the previously recorded WCRT */
14         remember worst-case response;
           /* Remove the finished task from the ready list */
15         remove(current_job, ready_list);
16         Update($\mathcal{S}$, current_job);
17 return TT schedule table $\mathcal{S}$ and WCRTs of TT tasks;
```

We suggest you use the following EDF simulation algorithm to create a schedule table for TT tasks, see chapter 4 in [3] for how EDF works. However, you may implement any scheduling algorithm, see, e.g., chapter 4 in [14] for task scheduling algorithms. Note that the algorithms in [14] consider non-preemptive tasks.

The EDF simulator takes as input the TT tasks. The output is a schedule table and the worst-case response times observed during the simulation for each TT job (a *job* is an instance of a periodic task). Although EDF scheduling is quite simple, the simulator is not trivial to implement efficiently. Algorithm 1 presents a simple inefficient implementation where time is advanced in every iteration with one microtick. We also show an improved EDF algorithm in Algorithm 3. Note that it is more efficient to advance time to the next event that needs to be handled by the simulation, i.e., a job of a task completes execution.

To create TT schedule table, we simulate up to the hyperperiod of the TT tasks, i.e., the least common multiple of periods of all TT tasks, since after that the schedule will repeat itself (line 1 in Algorithm 1).

A *ready job* at the time moment *cycle* is a job of that has already been released, i.e., the periodic task τ_i has arrived for execution again after its period T_i. The job executing at the time moment *cycle* is the job that has the highest priority (earliest deadline) out of those that are ready.

In our simple simulation, there is no need to "stop" or "preempt" a job already on the processor, since at every time instant *cycle* the simulator checks which ready job has the highest priority and runs it (decrements its execution time, line 11). Once a job has finished executing, it's removed from the ready list and from the jobs list (line 15).

During the simulation we have to remember the worst-case response time (WCRT) R_i of each TT task τ_i, which is the maximum response time over all its jobs (line 14). We also need to update the schedule table with the jobs' execution, e.g., add an entry in $\mathcal{S}$ for each time a job starts and finishes, which is done via the function Update($\mathcal{S}$, current_job).

4.2 ET schedulability analysis using Explicit Deadline Periodic (EDP)

As written in 1, the ET tasks are handled with polling servers [13]. There can be multiple polling servers and several ET tasks assigned to a polling server. We are interested to determine, for a given polling server, if the ET tasks assigned to it are schedulable and to determine their worst-case response times. To ease the notation, we assume for now that there is only one polling task τ_p handling the entire set $\mathcal{T}^{ET}$ of ET tasks for which C_p and T_p have to be determined. We use the Explicit Deadline Periodic (EDP) [8] where the server also has a deadline $D_p \leqslant T_p$.

We use the so-called linear supply lower bound function $lslbf(t)$ (c.f. [8]) with $\alpha = \frac{C_p}{T_p}$ and $\Delta = T_p + D_p - 2 \cdot C_p$, as

$$lslbf(t) = max\{0, (t - \Delta) \cdot \alpha\}. \tag{1}$$

We compute for each ET task $\tau_i \in \mathcal{T}^{ET}$ and for each instant t the maximum load of task τ_i and all higher and equal priority tasks (maximum load of level-i) $H_i(t)$. The maxium load of level-i for constrained deadline tasks is defined as

$$H_i(t) = \sum_{\forall \tau_j \in \mathcal{T}^{ET}, p_j \geq p_i} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j. \tag{2}$$

The schedulability condition for an ET task $\tau_i \in \mathcal{T}^{ET}$ is defined in Lemma 1 of [1]. The worst-case response time R_i for task $\tau_i \in \mathcal{T}^{ET}$ can be calculated by determining the earliest time instant in which the maximum load of level-i $H_i(t)$ intersects the linear supply bound function $lslbf(t)$ of the polling task from Eq.(1), as follows [1]:

$$R_i = \text{earliest } t : t = \Delta + H_i(t)/\alpha. \tag{3}$$

Algorithm 2 shows the pseudo-code of the worst-case response time analysis from Eq. 3, which you will have to call for each polling server (with its of budget, period and deadline) and its ET tasks. The algorithm checks the schedulability of an ET task set given a polling server.

Algorithm 2 Schedulability of ET tasks under a given polling task

```
Data: polling task budget $C_p$, polling task period $T_p$, polling task deadline $D_p$, subset of
      ET tasks to check $\mathcal{T}^{ET}$
Result: $\sigma$
/* Compute $\alpha$ and $\Delta$ according to [2]                                   */
1  $\Delta \leftarrow T_p + D_p - 2 \cdot C_p$;
2  $\alpha \leftarrow \frac{C_p}{T_p}$;
/* The hyperperiod is the least common multiple of all task periods in the set
   $\mathcal{T}^{ET}$                                                                  */
3  $T \leftarrow lcm\{T_i \mid \forall \tau_i \in \mathcal{T}^{ET}\}$;
4  for $\tau_i \in \mathcal{T}^{ET}$ do
5      $t \leftarrow 0$;
       /* Initialize the response time of $\tau_i$ to a value exceeding the deadline   */
6      $responseTime \leftarrow D_i + 1$;
       /* Remember, we are dealing with constrained deadline tasks for the
          AdvPoll, hence, in the worst case arrival pattern, the intersection
          must lie within the hyperperiod if the task set is schedulable.      */
7      while $t \leq T$ do
           /* The supply at time $t$ (c.f. [1])                                     */
8          $supply \leftarrow \alpha \cdot (t - \Delta)$;
           /* Compute the maximum demand at time $t$ according to Eq. 2            */
9          $demand \leftarrow 0$ ;
10         for $\tau_j \in \mathcal{T}^{ET}$ with $p_j \geq p_i$ do
11             $demand \leftarrow demand + \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j$ ;
           /* According to Lemma 1 of [1], we are searching for the earliest time,
              when the supply exceeds the demand                                 */
12         if $supply \geq demand$ then
13             $responseTime \leftarrow t$;
14             break;
15         $t \leftarrow t + 1$;
       /* If for any task the intersection of the demand and supply are larger
          than the task deadline, the task set $\mathcal{T}^{ET}$ is not schedulable using the
          given polling task parameters.                                        */
16     if $responseTime > D_i$ then
17         return false;
18 return true;
```

Appendix A: Improved EDF Algorithm

Algorithm 3 Scheduling TT tasks via EDF simulation

```
Data: TT task set $\mathcal{T}^{TT}$ including polling server tasks
Result: TT schedule table ($\sigma$) and WCRTs of TT tasks ($WCRT_i$)
$T \leftarrow lcm\{T_i \mid \forall \tau_i \in \mathcal{T}^{TT} \cup T^{poll}\}$;
$\forall\ \tau_i \in \mathcal{T}^{TT}$: $c_i \leftarrow C_i$; $d_i \leftarrow D_i$;$r_i \leftarrow 0$;$WCRT_i \leftarrow 0$;
$t \leftarrow 0$;
/* We go through each slot in the schedule table until $T$ */
while $t < T$ do
    for $\tau_i \in \mathcal{T}^{TT}$ do
        if $c_i > 0 \land d_i \geqslant t$ then
            return $\emptyset$; /* Deadline miss! */
        if $c_i == 0 \land d_i \leqslant t$ then
            /* Check if the current WCRT is larger than the current maximum. */
            if $t - r_i \geqslant WCRT_i$ then
                $WCRT_i \leftarrow t - r_i$;
        if $t\%T_i == 0$ then
            /* Task release at time $t$ */
            $r_i \leftarrow t$;
            $c_i \leftarrow C_i$;
            $d_i \leftarrow t + D_i$;
    /* Check if there is any tasks with computation left */
    if $\left[c_i = 0, \forall i \in \mathcal{T}^{TT}\right]$ then
        /* If no task has computation left, schedule idle slot */
        $\sigma$[t] $\leftarrow idle$;
    else
        /* If there are tasks with computation left, schedule the one with the
           earliest deadline */
        $\sigma$[t] $\leftarrow \tau_i = EDF(t, \mathcal{T}^{TT})$;
        $c_i \leftarrow c_i - 1$;
    $t \leftarrow t + 1$;
if $\left[c_i > 0, \forall i \in \mathcal{T}^{TT}\right]$ then
    /* Schedule is infeasible if any TT task has $c_i > 0$ at this point */
    return $\emptyset$;
returns $\sigma$;
```

References

1 Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proc. EMSOFT*. ACM, 2004. doi:10.1145/1017753.1017772.

2 Amir Aminifar, Enrico Bini, Petru Eles, and Zebo Peng. Designing bandwidth-efficient stabilizing control servers. In *Proc. RTSS*. IEEE, 2013. doi:10.1109/RTSS.2013.37.

3 Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

4 Silviu S. Craciunas, Ramon Serna Oliver, and Valentin Ecker. Optimal static scheduling of real-time tasks on distributed time-triggered networked systems. In *Proc. ETFA*. IEEE, 2014. doi:10.1109/ETFA.2014.7005128.

5 Paul Emberson, Roger Stafford, and Robert Davis. A taskset generator for experiments with real-time task sets. available at `https://github.com/jlelli/taskgen`, Accessed on 26.01.2022.

6 M. Hammond, G. Qu, and O. A. Rawashdeh. Deploying and scheduling vision based advanced driver assistance systems (ADAS) on heterogeneous multicore embedded platform. In *Proc. FCST*, 2015.

7 Herman Kopetz. Sparse time versus dense time in distributed real-time systems. In *Proc. ICDCS*, 1992. `doi:10.1109/ICDCS.1992.235008`.

8 Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proc. ECRTS*, 2003. `doi:10.1109/EMRTS.2003.1212738`.

9 Henrik Lonn and Jakob Axelsson. A comparison of fixed-priority and static cyclic scheduling for distributed automotive control applications. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 142–149. IEEE, 1999.

10 Shane D McLean, Emil Alexander Juul Hansen, Paul Pop, and Silviu S Craciunas. Configuring adas platforms for automotive applications using metaheuristics. *Frontiers in Robotics and AI*, 8, 2021.

11 Anna Minaeva and Zdeněk Hanzálek. Survey on periodic scheduling for time-triggered hard real-time systems. *ACM Comput. Surv.*, 54(1), 2021. `doi:10.1145/3431232`.

12 Georg Niedrist. Deterministic architecture and middleware for domain control units and simplified integration process applied to ADAS. In *Fahrerassistenzsysteme 2016*, pages 235–250. Springer Fachmedien Wiesbaden, 2018. `doi:https://doi.org/10.1007/978-3-658-21444-9`.

13 Insik Shin and Insup Lee. Compositional real-time scheduling framework with periodic model. *ACM Trans. Embed. Comput. Syst.*, 7(3):30:1–30:39, 2008. `doi:10.1145/1347375.1347383`.

14 Oliver Sinnen. *Task scheduling for parallel systems.* Wiley&Sons, 2007.