Rubus Offline Scheduler for Resource Constrained Embedded Real-Time Systems

Jukka Mäki-Turja
Arcticus Systems, Järfälla, Sweden
jukka.maki-turja@arcticus-systems.com

Vildan Zivojevic
Arcticus Systems, Järfälla, Sweden
vildan.zivojevic@arcticus-systems.com

Saad Mubeen
Mälardalen University, Västerås, Sweden
saad.mubeen@mdu.se

***Abstract*—In this paper, we present the Rubus tool suite, with a focus on its static (offline) real-time scheduler. The schedules generated by the scheduler are executed by a real-time operating system certified according to the ISO 26262 safety standard. The Rubus tool suite and its scheduler have been utilized in the vehicle industry for model- and component-based software development of resource-constrained embedded systems for over 25 years. Since its introduction in 1998, the scheduler has evolved significantly, transitioning from pure Earliest Deadline First (EDF) heuristics to incorporating priorities, data dependencies, and the ability to distribute the schedule over the entire hyper-period of the software application, among other heuristics. We provide an in-depth discussion on the mechanisms and algorithms that constitute the Rubus offline scheduler. Moreover, we provide an example of its application in generating an offline schedule for a part of software architecture in an industrial setting.**

I. Introduction

In modern vehicles, a significant portion of innovation and customer value is derived from advanced computer-controlled functionalities. As the volume of these functionalities increases, the complexity and size of vehicle software have grown substantially [1]. The size of vehicle software has already reached 100 million lines of code [2]. A study by Jaguar Land Rover projects that this figure will soon escalate to 1 billion lines of code [3]. Ensuring the safety of the vehicle's critical functions such as braking and engine control necessitates an approach that guarantees reliable operation. At the same time, there are numerous vehicle functions that are less critical, posing a challenge for manufacturers to provide both types of functions economically and reliably. Furthermore, the safety-critical nature of several vehicle functions imposes real-time requirements on them, demanding predictability in their execution. Developers must ensure that these functions behave in a timely manner when executed. To manage the software complexity, the vehicle industry often employs the principles of model-based development and component-based software engineering [4], [5]. Timing predictability can be verified through real-time schedulability analysis [6], [7], which ensures that timing requirements are met without the need for exhaustive testing before deploying vehicle software to target platforms. Examples of industrial modeling languages and tool chains include AUTOSAR standard [8], [9], Fraunhofer ESK [10], Infineon's Symtavision [11], and Rubus [12].

To address the aforementioned challenges, Arcticus Systems[1], in collaboration with academia and several industrial partners such as Volvo Construction Equipment and BAE Systems, has developed the theory, concepts, models, and a tool suite for model- and component-based development of vehicle software [12]. This tool suite, known as Rubus, has been in industrial use for over twenty-five years and is complemented by a real-time operating system (RTOS) for execution, certified according to the ISO 26262 - ASIL D safety standard [13]. The Rubus tool suite aims to be resource-efficient and provides means for developing timing-predictable and analyzable control functions in resource-constrained embedded systems.

This paper focuses on the Rubus tool suite, particularly the static (off-line) scheduler, which is central to the suite and possesses unique characteristics. Offline or static scheduling is a continuously evolving research field [14], [15], mainly because there is no optimal method for efficiently scheduling complex tasks with deadlines in a time frame suitable for software development of real-time systems in the industry. Generating offline schedules through optimization methods, such as constrained programming, can take a considerable amount of time to provide a feasible schedule. This is due to the complexity of the constraints and the need to explore a vast solution space to find an optimal or near-optimal schedule. Consequently, scheduler implementation in the industry necessitates trade-offs. In the case of Rubus, the focus is on balancing short run-time with the objective of finding a feasible schedule, even if it requires multiple attempts. Heuristic algorithms, like those used in the Rubus scheduler, are often preferred as they offer a balance between run-time efficiency and the ability to find valid schedules within a reasonable time frame.

Since its introduction to the industry in 1998, the Rubus scheduler has evolved significantly. Initially based on pure Earliest Deadline First (EDF) heuristics, it has become more intricate, incorporating priorities, data dependencies, and the ability to distribute the schedule over the entire hyper-period of time-triggered software functions to avoid front-heavy scheduling, benefiting lower-priority dynamic event-triggered software functions running in the background.

In this paper, we present the Rubus offline scheduler, detailing the mechanisms and algorithms behind it. The schedule generated by the scheduler is executed by the Rubus RTOS. We also provide an example of a part of software architecture in an industrial application where offline schedule is generated by the presented scheduler.

[1]http://www.arcticus-systems.com

II. The Rubus Modeling and Execution Framework

Rubus [16] is an integrated suite of methods, theories, and tools designed for model- and component-based development of predictable, timing-analyzable, and synthesizable control functions in resource-constrained embedded systems. Developed by Arcticus Systems in close collaboration with academia [12], Rubus is utilized by several international vehicle manufacturers in the development of control functionalities such as Volvo Construction Equipment[2], Mecel[3], Knorr-Bremse[4], Hoerbiger[5], and BAE Systems Hägglunds[6], to mention a few.

The Rubus tool suite focuses on defining a system architecture for software functions, essentially creating an execution environment for these functions. It considers three different viewpoints when addressing system architecture, as illustrated in Fig. 1. These viewpoints often highlight different and sometimes conflicting requirements during development. However, in practice, these viewpoints are interdependent and should not be considered in isolation, which is a common approach in traditional software engineering and real-time communities [17].

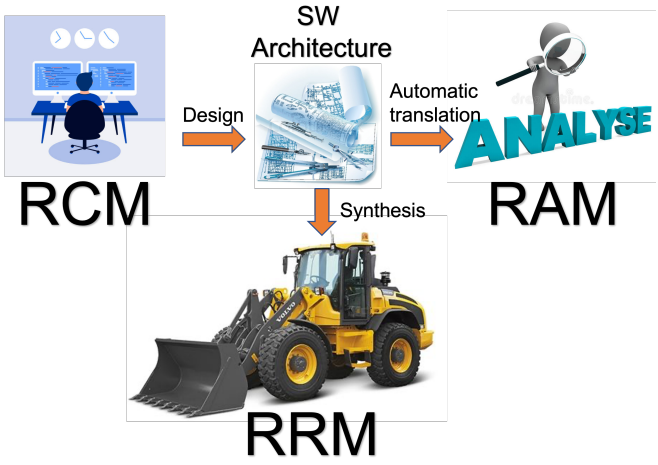

Fig. 1: Three viewpoints to the system architecture.

A. Rubus Component Model – Viewpoint of the Developer

The Rubus graphical designer tool is used to interactively describe applications developed using the Rubus Component Model (RCM). Developers design the system in a platform-independent manner, focusing on the application itself. Timing and resource constraints are specified through attributes of various modeling objects.

RCM defines the infrastructure of software functions in terms of software circuits (SWCs), which represent the interaction of software functions through data and control flow. Data flow and control flow are separated, meaning control flow does not necessarily involve data transmission. Control flow is expressed through triggering objects such as internal periodic clocks, interrupts, and internal and external events.

The execution semantics of an SWC follow the read-execute-write semantics. Upon triggering the SWC:

1) read data on its data in-ports;
2) executes the function;
3) write data on data out-ports;
4) and activate the output trigger.

An example SWC with three input data ports and two output data ports is shown in Fig. 2. The small triangles represent input and output triggers. An example of software architecture with three SWCs and other elements is depicted in Fig.3. This example depicts part of the software architecture in one of the Electronic Control Units (ECUs) in the autonomous cruise control system.

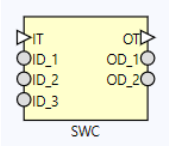

Fig. 2: An example SWC.

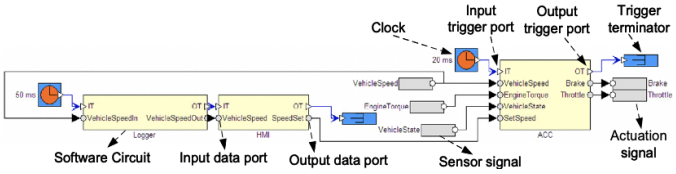

Fig. 3: An example software architecture modeled in RCM.

B. Analysis Model – Viewpoint of the Analysis Framework

The formal design resulting from RCM facilitates static analysis, which is mapped to the actual run-time platform. In this viewpoint, the Rubus Analyzer analyses the type checking, execution order, and real-time requirements. This analysis helps to reduce late, costly, and time-consuming testing efforts. Furthermore, mathematical models and supporting tools provide formal evidence of fulfillment of the timing requirements [18].

The Rubus Analyzer supports pre-runtime timing analysis of the system at various levels. For instance, it analyzes a single node by calculating tasks' response times and comparing them with corresponding deadlines. The Rubus Analyzer implements Response Time Analysis (RTA) for tasks with offsets [19], [20], RTA for Controller Area Network (CAN) and its higher-level protocols [21], RTA of Time Sensitive Networking (TSN) [22], [23], analysis of multiple networks, and end-to-end timing analysis [18], [24], to calculate data age and reaction time delays.

C. Run-time Model – Viewpoint of the Run-time System

The code for the actual run-time platform is synthesized from the system architecture, ensuring automated synthesis

[2]https://www.volvoce.com
[3]https://kvaser.com/company/mecel/
[4]https://www.knorr-bremse.com/en/
[5]https://www.hoerbiger.com/
[6]http://www.baesystems.com

that prevents error-prone and costly integration errors. The Rubus RTOS supports RCM in achieving an optimized real-time software system. It has been utilized in a wide range of industrial real-time applications, including wheel loaders, articulated haulers, excavators, and various four-wheel drive vehicles. Key features of the RTOS include support for the execution and communication among time-, event-, and interrupt-triggered tasks, static allocation of resources, scalability, and portability. The combination of dynamic and static scheduling supported by the Rubus RTOS enables the design of optimized real-time software systems.

In the Rubus platform, all SWCs triggered by a time-triggered (TT) clock are scheduled offline (pre-run-time). The scheduler generates an SWC schedule based on time settings and execution relations, taking interrupt interference into account to ensure that schedules remain feasible under the worst-case specified interrupt load [25]. Additionally, it is possible to distribute the execution evenly over the entire schedule length, thereby reducing response times and the jitter of dynamic (event-triggered) task execution.

Furthermore, it should be noted that the resulting schedule is non-preemptive. Time-triggered tasks can only be preempted by interrupts, not other time-triggered tasks.

Consider an example task set with interrupt arrivals, a static schedule, and dynamic task arrivals, as shown in Fig. 4. The Rubus RTOS prioritizes interrupts first, followed by the static schedule and finally the dynamic (event-triggered) tasks, as illustrated in the resulting execution pattern.

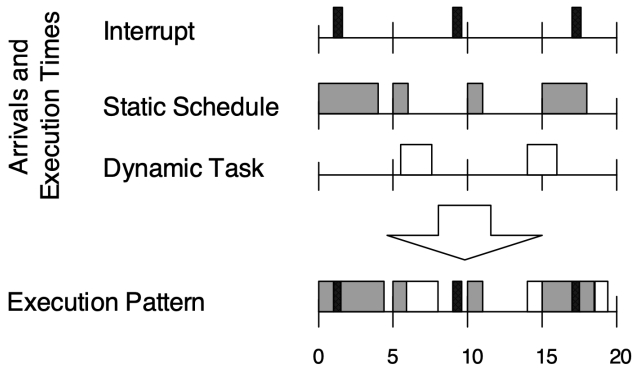

Fig. 4: Execution scenario with interrupts, TT- and ET-tasks

D. Related Works

Offline scheduling has been a fundamental approach to schedule time-triggered tasks and messages in real-time systems and networks respectively for decades [26], [27]. Historically, it has been used to ensure timing predictability of the systems by pre-determining the execution order of tasks and messages before runtime. Two notable comprehensive surveys on time-triggered scheduling of periodic hard real-time tasks are presented in [14], [15].

There are several works that target offline scheduling of tasks and network messages together. For instance, the work in [28] performs offline scheduling of tasks and communications by employing a mixed integer programming model focusing on Ethernet-based networks. Similarly, the work in [29] develops concurrent co-synthesis of switched time-triggered network's schedule and application schedules for preemptive time-triggered tasks. A search-space pruning technique is presented in [30] that uses response times to efficiently generate schedules for large systems, and extend the modular scheduler design to support TTEthernet networks. The work in [31] presents time-triggered co-scheduling scheduling of computation and communication. In comparison, the work presented in this paper primarily focuses on offline scheduling of time-triggered periodic tasks, which operate alongside event-triggered and interrupt-triggered tasks in single-core real-time systems.

Slot shifting [32] is a method for combining time- and event-triggered scheduling. It analyzes offline constructed schedules for the amount and location of unused resources, called spare capacities, which are used to accommodate event-triggered activities, such as aperiodic tasks. Spare capacities are represented to indicate available resources and the flexibility to shift offline scheduled tasks while maintaining their feasibility. Slot shifting, as opposed to Rubus, has a flexible static schedule where time-triggered tasks are shifted, to accommodate event-triggered tasks, as long as they do not violate the deadline of time-triggered tasks. Rubus, on the other hand, has a fixed schedule in which event-triggered tasks run in the background. Slot shifting is therefore more flexible, whereas Rubus focuses more on the predictability of time-triggered tasks.

There are several works that use simulated annealing to schedule time-triggered tasks, e.g., the work by Tindell [33]. The work in [34] focuses on scheduling real-time tasks in distributed systems with the goal of minimizing the tasks' jitter. The schedule proposed in our paper minimizes the tasks' jitter based on the techniques described in the above works. A simulated annealing algorithm adjusts the release times of time-triggered tasks to reduce the gap between their release times and deadlines, while adhering to jitter constraints during the creation of offline schedules.

MAST[7] is an open-source suite of tools for performing schedulability analysis of real-time distributed systems. MAST has an event-driven model, i.e., tasks are considered activated through independent events from outside as well as inside the system [35]. In addition, it has a UML profile, UML-MAST, used to model the system. MAST and Rubus response time analysis of event-driven tasks (not covered in this paper), use the same system model; task model with offsets [19], [20]. MAST does not have an offline scheduler and therefore no off-line schedule can be generated. All tasks are considered priority-based, either EDF or fixed priority. Rubus scheduler, on the other hand, has three different levels of execution; interrupts, offline schedule, and event-triggered tasks.

[7]https://mast.unican.es

III. System Model

This section introduces the system model utilized by the Rubus scheduler, which will also be used to describe the offline scheduler and the heuristics implemented within it. The schedule is generated independently for each node in the system. The **system**, denoted by $\mathcal{E}$, consists of one or more nodes. A node i is denoted by $\mathcal{E}_i$. Each node is considered to be a uniprocessor system. The total number of nodes in the system is represented by $|\mathcal{E}|$. Hence, the system can be formally represented as follows.

$$System := \{\mathcal{E}_1, ..., \mathcal{E}_{|\mathcal{E}|}\} \tag{1}$$

A **node** contains three types of task sets as shown by Eq. 2. For example, node $\mathcal{E}_i$ may contain (1) a task set Γ_i^{TT} that contains all time-triggered (TT) tasks, (2) a task set Γ_i^{IT} that contains all interrupt-triggered (IT) tasks, and (3) a task set Γ_i^{ET} comprising all event-triggered (ET) tasks in the node.

$$\mathcal{E}_i := \langle \Gamma_i^{TT}, \Gamma_i^{IT}, \Gamma_i^{ET} \rangle \tag{2}$$

Each type of **task set** consists of one or more tasks as shown in Eq. 3, Eq. 4 and Eq. 5 for Γ_i^{TT}, Γ_i^{IT} and Γ_i^{ET} respectively. For instance, in Eq. 3, τ_{ix}^{TT} denotes the time-triggered task with ID x that belongs to node $\mathcal{E}_i$.

$$\Gamma_i^{TT} := \{\tau_{i1}^{TT}, ..., \tau_{ix}^{TT}\} \tag{3}$$

$$\Gamma_i^{IT} := \{\tau_{i1}^{IT}, ..., \tau_{iy}^{IT}\} \tag{4}$$

$$\Gamma_i^{ET} := \{\tau_{i1}^{ET}, ..., \tau_{iz}^{ET}\} \tag{5}$$

We consider a one-to-one **mapping between an SWC in the software architecture and a task at runtime**. For instance, the three SWCs in the software architecture depicted in Fig. 3 correspond to three tasks in the system model at runtime. If an SWC is activated by a periodic clock, an interrupt source, or an event source, the corresponding task in the system model will be classified as TT, IT or ET, respectively.

A **time-triggered task** is represented as τ_{ij}^{TT}, where the subscript i indicates the node ID to which the task belongs, and the subscript j denotes the task's ID. This task is characterized by the following tuple.

$$\tau_{ij}^{TT} := \langle C_{ij}^{TT}, BC_{ij}^{TT}, T_{ij}^{TT}, P_{ij}^{TT}, R_{ij}^{TT}, D_{ij}^{TT} \rangle \tag{6}$$

Where C_{ij}^{TT} and BC_{ij}^{TT} represent the worst-case execution time (WCET) and the best-case execution time (BCET) of the task respectively. The period of the task, denoted by T_{ij}^{TT}, corresponds to the period of the time-triggered clock that triggers the software circuit in the software architecture. For instance, two TT tasks correspond to two software circuits, Logger and ACC, which are triggered by TT clocks with periods of 50ms and 20ms, respectively, as illustrated in Fig. 3. The priority and deadline of the TT task are represented by P_{ij}^{TT} and D_{ij}^{TT} respectively. Furthermore, the response time of the TT task is denoted by R_{ij}^{TT}.

Similarly, Eq. 7 and Eq. 8 provide parameters of interrupt- and event-triggered tasks respectively.

$$\tau_{ij}^{IT} := \langle C_{ij}^{IT}, BC_{ij}^{IT}, T_{ij}^{IT}, P_{ij}^{IT}, R_{ij}^{IT}, D_{ij}^{IT} \rangle \tag{7}$$

$$\tau_{ij}^{ET} := \langle C_{ij}^{ET}, BC_{ij}^{ET}, T_{ij}^{ET}, P_{ij}^{ET}, R_{ij}^{ET}, D_{ij}^{ET} \rangle \tag{8}$$

Note that in node $\mathcal{E}_i$, the **priority level** of all tasks in Γ_i^{IT} is higher than that of all tasks in Γ_i^{TT}. Similarly, the priority level of all tasks in Γ_i^{TT} is higher than that of all tasks in Γ_i^{ET}. In other words, the lowest-priority task in Γ_i^{IT} will have a higher priority than the highest-priority task in Γ_i^{TT}. Likewise, the lowest-priority task in Γ_i^{TT} will have a higher priority than the highest-priority task in Γ_i^{ET}. This implies that the generated time-triggered schedule will not be affected by any ET task. Therefore, ET tasks will not be used in any of the presented algorithms for schedule generation. They are included in the system model for completeness.

The **total utilization** of all TT tasks in node $\mathcal{E}_i$ is represented by:

$$U_i^{TT} = \sum_{j=1}^{n} \frac{C_{ij}^{TT}}{T_{ij}^{TT}}$$

where C_{ij}^{TT} is the worst-case execution time of task j, T_{ij}^{TT} is the period of task j, and n is the total number of TT tasks. Similarly, the total utilization of all IT tasks in node $\mathcal{E}_i$ is represented by U_i^{IT}.

The **hyper-period** of all TT tasks in Γ_i^{TT} within node $\mathcal{E}_i$ is denoted by HP_i^{TT}. Note that the hyper-period of a task set is equal to the Least Common Multiple (LCM) of the periods of all tasks in the task set.

$$HP_i^{TT} := LCM(T_{i1}^{TT}, T_{i2}^{TT}, \ldots, T_{in}^{TT}) \tag{9}$$

The **set of release times** of all TT task instances in HP_i^{TT} is represented by RT_i^{TT}. For example, $RT_i^{TT}[1]$, $RT_i^{TT}[2]$ and $RT_i^{TT}[3]$ represent the first, second and third release times in HP_i^{TT} respectively. All release times are precomputed based on the periods of the TT tasks within HP_i^{TT}. Note that multiple tasks may have instances that are ready to execute simultaneously at a release time in RT_i^{TT}, however only one task instance is executed at a time in the uni-processor system. The set of all TT tasks' instances that are ready to execute on a release time RT is denoted by $\Gamma_{i,RT}^{TT}$.

The task model also supports **conditionally-triggered tasks**, which correspond to software circuits (SWCs) in the software architecture that are triggered for execution by conditionally triggering objects. For instance, the task corresponding to SWC_C in Fig. 5 is a conditionally-triggered task that is triggered by a Trig_Sync object. The Trig_Sync object receives input triggers from two TT SWCs, namely SWC_A and SWC_B, with different periods. It triggers SWC_C when it has received both input triggers. Users can specify the deadline for a conditionally-triggered task. If a deadline is not specified, it defaults to the least common multiple of the periods of its predecessor tasks.

The **time-triggered schedule** of all tasks in Γ_i^{TT}, generated by the scheduler, is denoted by $\mathcal{S}_i$. Basically, $\mathcal{S}_i$ consists of an ordered set of pairs, each containing a task instance and its

corresponding release time, for every release time in RT_i^{TT}. The schedule is formally presented as follows.

$$\mathcal{S}_i = \{(\tau_{ij}^{TT}, t_k) \mid \forall \tau_{ij}^{TT} \text{ occurring at } t_k, \forall t_k \in RT_i^{TT}\} \quad (10)$$

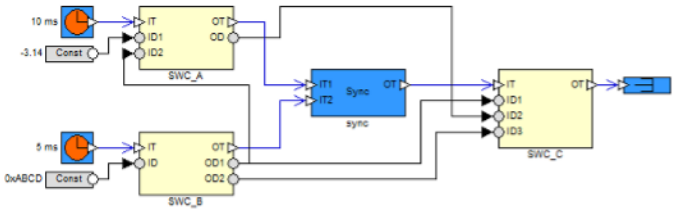

Fig. 5: Example of a conditionally time-triggered SWC and corresponding task.

IV. Rubus Offline Scheduler

This section presents features, algorithms and heuristics that form the foundation of the Rubus offline scheduler.

A. Main Features of the Scheduler

*1) **Established Scheduling Algorithms**:* Rubus employs heuristic algorithms that integrate established scheduling algorithms such as Earliest Deadline First (EDF), Rate Monotonic (RM), and Bin Packing. The algorithms prioritize the most promising combinations, dedicating significant effort to optimizing them. The resulting schedule is characterized as proof-by-construction, meaning that if a schedule is successfully generated, it serves as evidence of its validity.

*2) **Control Dependencies**:* The scheduler takes into account control dependencies among the SWCs (tasks) and resolves them using precedence relations [36]. For instance, the HMI SWC is dependent on the Logger SWC, as illustrated in Fig. 3. This means that the HMI SWC can only be triggered for execution once the Logger SWC has completed its execution and produced an output trigger. Consequently, the scheduler will schedule the task corresponding to the HMI SWC after the task corresponding to the Logger SWC has been scheduled.

*3) **Data Dependencies**:* The software architecture distinctly defines control-flow and data-flow relations between SWCs. This implies that a triggering event does not inherently carry data. When data dependencies are specified within the software architecture, the scheduler takes these dependencies into account when scheduling the corresponding tasks. This tries to ensure that tasks consuming data are not scheduled before their respective data-producing tasks.

*4) **Conditionally Triggered Tasks**:* The scheduler also supports conditionally-triggered tasks. The Trig_Sync object in the modeling framework introduces tasks that cannot be handled as ordinary time-triggered tasks because they do not have a period. This Trig_Sync object triggers its successor task when all of its predecessor tasks have been scheduled. Since all predecessor tasks can have different periods, the conditionally-triggered task does not have any period as it depends on when the predecessor tasks are scheduled/executed. Instead, the conditionally-triggered task becomes eligible for scheduling on the fly when all predecessor tasks have been scheduled.

*5) **Distribute Schedule**:* The scheduler allows the user to optionally select the distribute schedule feature. If this feature is not enabled, the scheduler attempts to maximize execution of time-triggered tasks at each release time by scheduling them as early as possible. This results in "front-heavy" schedules, leaving little slack at the beginning. To address this, the distribute schedule option is introduced, which spreads the schedule more evenly. This improves the responsiveness of background tasks (event-triggered tasks) by creating evenly distributed gaps in the schedule. When the distribute schedule option is enabled, the scheduler uses the combined utilization of time-triggered and interrupt tasks as a scheduling threshold. During the scheduling of time-triggered tasks at a release time, it assesses the remaining execution supply between the current and subsequent release times. If this supply exceeds the scheduling threshold, the scheduler shifts the current task and any unscheduled tasks from the current release time to the next, rather than fully utilizing the execution supply from the current to the next release time.

*6) **Postpone Release Times**:* The scheduler supports an optional feature that allows postponing release times, which can be specified by the user. This feature serves as a performance enhancement, increasing schedulability.

If this feature is not enabled, when scheduling at the current release time, any time-triggered task that cannot be completed before the start of the next release time is moved to the next release time. All remaining unscheduled tasks are also moved to the next release time. Scheduling then continues from there.

When the postpone release times feature is enabled, if a time-triggered task scheduled at the current release time cannot finish before the next release time, the release time is postponed to the finishing time of that task. Instead of moving the task to the next release time, the task is scheduled, and the start of the next release time is adjusted to accommodate the scheduled task. Thus, the finishing time of the task becomes the start of the next release time.

Note that the postpone-release-time and the distribute-schedule features are mutually exclusive and should not be used together.

*7) **Scheduling Task with Minimizing Jitter**:* The user can specify jitter requirements for SWCs corresponding to time-triggered tasks. The scheduler aims to minimize jitter based on the technique described in [34]. A simulated annealing algorithm randomizes the release time of a time-triggered task to create a tighter gap between its release time and deadline. The entire schedule is then evaluated for schedulability. If the system is schedulable, the next release time is selected and the process repeats; otherwise, the release time is discarded.

B. Offline Scheduler Algorithm

The algorithm "Creation of offline schedule", depicted in Algorithm 1, is designed to generate a feasible schedule for time-triggered (TT) tasks within their hyper-period. The

inputs to this algorithm include the set of all TT tasks Γ_i^{TT} and interrupt-triggered (IT) tasks Γ_i^{IT} in the given node $\mathcal{E}i$. Furthermore, it requires the parameters of all TT tasks τij^{TT} and IT tasks τ_{ij}^{IT}.

The algorithm first calculates the hyper-period HP_i^{TT} of all TT tasks in Γ_i^{TT}. Following this, it calculates the release times RT_i^{TT} of all TT task instances within the hyper-period (lines 6-7). Note that all task instances are assigned a release time, which is the earliest point in time when a task instance can be scheduled.

The core of the algorithm involves iterating through each release time in the set RT_i^{TT}. For each task released at that release time, the algorithm selects a TT task τij^{TT} (line 11) based on basic heuristics that are presented in Algorithm 2. If Advanced heuristics are enabled then the TT task is selected using Algorithm 3. If none of the task selection algorithms can identify a task based on the heuristics, it indicates that the time interval between the current release time and the next release time lacks sufficient supply to execute any of the remaining unscheduled tasks. Consequently, all unscheduled tasks at the current release time are deferred to the next release time for scheduling (lines 12-14).

Once a TT task (say τij^{TT}) is selected, the algorithm calculates the task's response time R_{ij}^{TT} (line 15). The response time of the task is calculated considering interference from all previously scheduled tasks at the current release time, as well as the interference from all IT tasks released at the current release time, following the analysis for tasks with offsets [19], [20]. If the response time exceeds the task's deadline D_{ij}^{TT}, the algorithm returns "Unschedulable" (line 17).

If the response time of τ_{ij}^{TT} exceeds the next release time, it indicates that the execution supply between the current and next release times has been exhausted. Hence, if R_{ij}^{TT} falls between the next two release times (line 19), the algorithm checks if postponing the release time is enabled (line 20). If so, it moves the start of next release time to the response time of the task (line 21). Otherwise, it moves τ_{ij}^{TT} and any remaining unscheduled tasks to the next release time (lines 23-24).

If the distribute schedule feature is enabled and the execution supply between the current and the next release times is less than the combined utilization of TT tasks, U_i^{TT}, and IT tasks, U_i^{IT}, (line 27), the scheduler will move τ_{ij}^{TT} to the next release time. Additionally, all remaining unscheduled tasks will also be moved to the next release time for scheduling (lines 28-29).

If τ_{ij}^{TT} is schedulable, the algorithm saves the order of the task and release-time pair in the schedule (line 31). If τ_{ij}^{TT} is conditionally triggered, the scheduler generates its new instance, updates its precedence details, and releases it for scheduling (lines 32-33). This allows the task to be considered for scheduling in the next loop iteration, provided it has become eligible for execution, meaning all its predecessors have been scheduled. If the user does not specify a deadline for a conditionally triggered task, it is calculated from the time it becomes eligible for execution plus the least common multiple of the periods of all its predecessor tasks. Note that

Algorithm 1 Creation of offline schedule

1: **function** CREATEOFFLINESCHEDULE($\mathcal{E}_i$)
2: **Input:** Task sets Γ_i^{TT} and Γ_i^{IT} in $\mathcal{E}_i$
3: **Input:** Parameters of all τ_{ij}^{TT} tasks according to Eq. 6
4: **Input:** Parameters of all τ_{ij}^{IT} tasks according to Eq. 7
5: **Output:** Feasible schedule of Γ_i^{TT} in its hyper-period
6: **Calculate** $HP_i^{TT}(\Gamma_i^{TT})$ ▷ hyper-period of all tasks in Γ_i^{TT}.
7: **Calculate** $RT_i^{TT}(\Gamma_i^{TT})$ ▷ Release times of all TT tasks' instances in HP_i^{TT}.
8: **for** $n : 1$ **to** $|RT_i^{TT}|$ **do**
9: **for** each task released at RT_i^{TT}[n] **do** ▷ Multiple tasks may have instances released at a release time simultaneously.
10: RT = RT_i^{TT}[n]
11: $\tau_{ij}^{TT} \leftarrow$ TaskSelectionHeuristics($\Gamma_{i,RT}^{TT}$) ▷ τ_{ij}^{TT} selected according to heuristics in Algorithms 2 and 3. $\Gamma_{i,RT}^{TT}$ contains all TT tasks' instances at release time RT.
12: **if** ($\tau_{ij}^{TT} = \emptyset$) **then**
13: Move(RemUnschedTasks, RT_i^{TT}[n+1]) ▷ If the returned task is empty then move remaining unscheduled tasks from the current release time to the next release time for scheduling.
14: **end if**
15: CalculateResponseTime(R_{ij}^{TT})
16: **if** $R_{ij}^{TT} > D_{ij}^{TT}$ **then**
17: **return** Unschedulable
18: **else**
19: **if** RT_i^{TT}[n+1]$<R_{ij}^{TT}\leq RT_i^{TT}$[n+2] **then**
20: **if** PostponeReleaseTimeEnabled **then**
21: Move(RT_i^{TT}[n+1], R_{ij}^{TT}) ▷ Move the beginning of the next release time to the response time of τ_{ij}^{TT}.
22: **else**
23: Move(τ_{ij}^{TT}, RT_i^{TT}[n+1])
24: Move(RemUnschedTasks, RT_i^{TT}[n+1]) ▷ Move τ_{ij}^{TT} and remaining unscheduled tasks from the current release time to the next release time for scheduling.
25: **end if**
26: **else**
27: **if** DistributeScheduleEnabled & (Uc_i^{TT}+Uc_i^{IT} < U_i^{TT}+U_i^{IT}) **then**
28: Move(τ_{ij}^{TT}, RT_i^{TT}[n+1])
29: Move(RemUnschedTasks, RT_i^{TT}[n+1])
30: **else**
31: $\mathcal{S}_i \leftarrow$ (τ_{ij}^{TT}, RT_i^{TT}[n]) ▷ τ_{ij}^{TT} is schedulable. Save the order of τ_{ij}^{TT} and RT pair in the schedule.
32: **if** $\tau_{ij}^{TT}$is ConditionalTriggered **then**
33: ReleaseNewInstance(τ_{ij}^{TT})
34: **end if**
35: **end if**
36: **end if**
37: **end if**
38: **end for**
39: **end for**
40: **return** Schedule ($\mathcal{S}_i$)
41: **end function**

multiple instances of a conditionally triggered task cannot be active simultaneously, as a new instance is created only when the previous one has been scheduled.

Finally, the algorithm returns the generated schedule $\mathcal{S}_i$ (line 40).

C. Basic Heuristics Algorithm

The algorithm "Basic heuristics for task selection", presented in Algorithm 2, is designed to select and provide a unique time-triggered task instance to schedule at a given release time. The process begins by taking as input the set of all time-triggered task instances at the specified release time (lines 2-3). To achieve this, the algorithm employs several heuristics in sequence to narrow down the selection.

First, the algorithm uses the Earliest Deadline First (EDF) heuristic to select all task instances at the given RT that have the same deadline. This results in a subset $\Omega_{i,RT}^{TT}$ of the original task set $\Gamma_{i,RT}^{TT}$ (line 4). Next, the algorithm selects all highest-priority task instances from the subset $\Omega_{i,RT}^{TT}$ and updates the subset accordingly (line 5). Following this, the algorithm applies the best-fit heuristic (the task with largest WCET that fits within remaining supply of the current release time) to select task instances from the updated subset $\Omega_{i,RT}^{TT}$ (line 6). It then uses the Rate Monotonic Algorithm to further refine the selection of task instances from the subset $\Omega_{i,RT}^{TT}$ (line 7). After that, the algorithm employs the Earliest Release Time heuristic to select task instances from the subset $\Omega_{i,RT}^{TT}$ (line 8).

Finally, the algorithm selects one task instance from the subset $\Omega_{i,RT}^{TT}$ alphabetically based on its unique ID (line 9). After applying these heuristics, the algorithm returns the unique time-triggered task instance, if it can be found, to schedule at the current release time (line 10). Note that if any of the heuristics on lines 4-8 result in a unique task instance in the subset $\Omega_{i,RT}^{TT}$, the function returns this instance to schedule at the current release time. These multiple checks are omitted in Algorithm 2 for better readability.

D. Advanced Heuristics Algorithm

The user can choose to use advanced heuristics, which is a performance enhancement over the basic one. The basic heuristic is a simple one that makes it easy to understand the resulting schedules, whereas the advanced heuristic is more intricate and may result in schedules that are less obvious. However, with advanced heuristics, there is an increase in schedulability.

The algorithm "Advanced heuristics for task selection", shown in Algorithm 3, is designed to select a unique time-triggered task instance to schedule at a given release time. The algorithm begins by taking as input the set of all TT task instances $\Gamma_{i,RT}^{TT}$ at the specified release time. The output of the algorithm is a unique time-triggered task instance to schedule at the release time. The algorithm uses several advanced heuristics in sequence to narrow down the selection. The primary distinction from the basic heuristics algorithm is the consideration of data dependencies among tasks. Additionally,

Algorithm 2 Basic heuristics for task selection

1: **function** TASKSELECTIONHEURISTICS_BASIC
2: **Input:** $\Gamma_{i,RT}^{TT}$ ▷ Set of all TT tasks' instances at a given Release Time.
3: **Output:** Unique TT task to schedule at RT
4: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_EarliestDeadlineFirst($\Gamma_{i,RT}^{TT}$) ▷ Use EDF to select all tasks' instances at the given RT that have the same deadline. $\Omega_{i,RT}^{TT}$ is a subset of $\Gamma_{i,RT}^{TT}$.
5: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_HighestPriority($\Omega_{i,RT}^{TT}$) ▷ Select all highest-priority tasks' instances from $\Omega_{i,RT}^{TT}$, update $\Omega_{i,RT}^{TT}$.
6: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_BestFit($\Omega_{i,RT}^{TT}$) ▷ Select all tasks' instances from $\Omega_{i,RT}^{TT}$ using the best-fit heuristic, update $\Omega_{i,RT}^{TT}$.
7: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_RateMonotonic($\Omega_{i,RT}^{TT}$) ▷ Use Rate Monotonic Algorithm to select all tasks' instances from $\Omega_{i,RT}^{TT}$, update $\Omega_{i,RT}^{TT}$.
8: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_EarliestReleaseTime($\Omega_{i,RT}^{TT}$) ▷ Use Earliest Release Time algorithm to select all tasks' instances from $\Omega_{i,RT}^{TT}$, update $\Omega_{i,RT}^{TT}$.
9: $\Omega_{i,RT}^{TT} \leftarrow$ SelectTasks_Alphabetically($\Omega_{i,RT}^{TT}$) ▷ Select one task from $\Omega_{i,RT}^{TT}$ alphabetically based on its ID, update $\Omega_{i,RT}^{TT}$.
10: **return** (Task $\in \Omega_{i,RT}^{TT}$)
11: **end function**

if a task is not found to be the best fit after applying EDF and priority, both EDF and priority-based selection are ignored, and the best fit is performed on all tasks.

The basic scheduler focuses on control flow, maintaining precedence relations between tasks, which may result in data consumer tasks being scheduled before their producer tasks. When the data-dependency feature is enabled, heuristics prioritize scheduling tasks with no incoming data dependencies first, as long as schedulability is not affected (lines 4-5). This results in a subset $\Omega_{i,RT}^{TT}$ of the original task set $\Gamma_{i,RT}^{TT}$. If the data-dependency feature is not selected, heuristics proceed without considering data dependencies. Enabling the data-dependency feature does not impact schedulability.

Next, the algorithm selects tasks with the shortest deadline for data to propagate to successor tasks and updates the subset accordingly (line 6). Following this, the algorithm uses the Earliest Deadline First (EDF) heuristic to select all task instances at the given release time that have the same deadline, further refining the subset $\Omega_{i,RT}^{TT}$ (line 8). It then selects all highest-priority task instances from the updated subset $\Omega_{i,RT}^{TT}$ (line 9). The algorithm then applies the best-fit heuristic to select task instances from the subset $\Omega_{i,RT}^{TT}$ (line 10).

If the subset $\Omega_{i,RT}^{TT}$ becomes empty, the algorithm fetches the original set of all TT tasks' instances at the given release time, ignores the EDF and highest-priority selection steps, and proceeds to the best-fist heuristic (lines 11-13) and then rate-monotonic heuristic (line 14) to select task instances from the subset. It then employs the Earliest Release Time heuristic to further refine the selection of task instances from the subset (line 15).

Finally, the algorithm selects one task from the subset $\Omega_{i,RT}^{TT}$

Algorithm 3 Advanced heuristics for task selection

1: **function** TASKSELECTIONHEURISTICS_ADVANCED
2: **Input:** $\Gamma^{TT}_{i,RT}$ ▷ Set of all TT tasks' instances at a given Release Time.
3: **Output:** Unique TT task to schedule at RT
4: **if** (DataDependenciesConsidered) **then**
5: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_NoDataInDependency($\Gamma^{TT}_{i,RT}$) ▷ Select tasks with no incoming data-dependencies if schedulability of other tasks is not affected. $\Omega^{TT}_{i,RT}$ is a subset of $\Gamma^{TT}_{i,RT}$.
6: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_ShortestDataDeadline($\Omega^{TT}_{i,RT}$) ▷ Select tasks with shortest deadline for data to propagate to the successor tasks, update $\Omega^{TT}_{i,RT}$.
7: **end if**
8: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_EarliestDeadlineFirst($\Omega^{TT}_{i,RT}$) ▷ Use EDF to select all tasks' instances at the given RT that have the same deadline, update $\Omega^{TT}_{i,RT}$.
9: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_HighestPriority($\Omega^{TT}_{i,RT}$) ▷ Select all highest-priority tasks' instances from $\Omega^{TT}_{i,RT}$, update $\Omega^{TT}_{i,RT}$.
10: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_BestFit($\Omega^{TT}_{i,RT}$) ▷ Select all tasks' instances from $\Omega^{TT}_{i,RT}$ using the best-fit heuristic, update $\Omega^{TT}_{i,RT}$.
11: **if** ($|\Omega^{TT}_{i,RT}| = \emptyset$) **then**
12: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_BestFit($\Gamma^{TT}_{i,RT}$) ▷ Task selection from the original set of input TT tasks' instances at the given Release Time
13: **end if**
14: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_RateMonotonic($\Omega^{TT}_{i,RT}$) ▷ Use Rate Monotonic Algorithm to select all tasks' instances from $\Omega^{TT}_{i,RT}$, update $\Omega^{TT}_{i,RT}$.
15: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_EarliestReleaseTime($\Omega^{TT}_{i,RT}$) ▷ Use Earliest Release Time algorithm to select all tasks' instances from $\Omega^{TT}_{i,RT}$, update $\Omega^{TT}_{i,RT}$.
16: $\Omega^{TT}_{i,RT} \leftarrow$ SelectTasks_Alphabetically($\Omega^{TT}_{i,RT}$) ▷ Select one task from $\Omega^{TT}_{i,RT}$ alphabetically based on its ID, update $\Omega^{TT}_{i,RT}$.
17: **return** (Task $\in \Omega^{TT}_{i,RT}$)
18: **end function**

alphabetically based on its unique ID (line 16). Note that if any of the heuristics on lines 4-18 result in a unique task instance in the subset $\Omega^{TT}_{i,RT}$, the function returns this instance to schedule at the current release time. These multiple checks are omitted in Algorithm 3 for better readability.

After applying these advanced heuristics, the algorithm returns the unique time-triggered task instance, if it can be found, to schedule at the release time (line 17).

V. EVALUATION

We demonstrate the proof of concept for the offline scheduler by modeling a segment of the software architecture of an industrial application using the Rubus Component Model and generating a feasible offline schedule with the scheduler. To ensure clarity and illustrate the step-by-step schedule generation, we focus on only four SWCs in the software architecture, as depicted in Fig. 6. Furthermore, we have replaced the actual names of the SWCs with more generic names for IP protection, while keeping the rest of the parameters in the software architecture unchanged. Towards the end of this section, we will show a real schedule generated by the Rubus scheduler for time-triggered tasks within a proprietary Electronic Control Unit (ECU) in a contemporary vehicle.

The part of the application in Fig. 6 consists of 4 SWCs:

- One SWC triggered by a 20 ms TT clock: *Sensor* with a WCET of 3 ms.
- One SWC triggered by a 30 ms TT clock: *Control* with a WCET of 7 ms.
- One SWC triggered by *Control* SWC: *Actuate* with a WCET of 4 ms.
- One SWC triggered by an interrupt with a minimum inter-arrival time of 5 ms: *Interrupt* with a WCET of 1 ms.

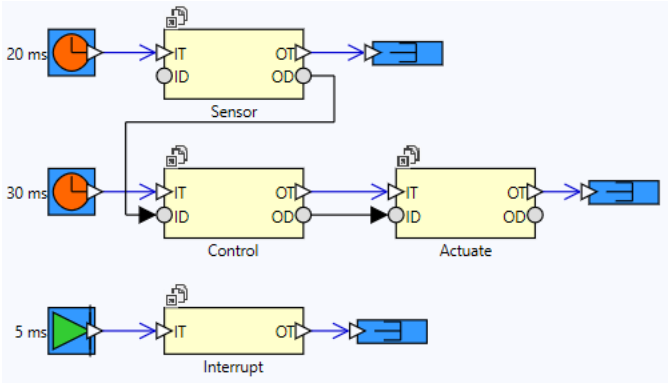

Fig. 6: Partial software architecture of an industrial application modeled with Rubus.

Furthermore, data flows from *Sensor* to *Control* to *Actuate* SWCs, with all time-triggered SWCs having the same priority. Note that we consider a one-to-one mapping between an SWC and a task. Hence, there are four tasks in the system with the corresponding names. The hyper-period of the task set, and thus the schedule length, is 60 ms (with time-triggered task periods of 20 ms and 30 ms), and the release times are 0 ms, 20 ms, 30 ms, and 40 ms. Table I shows the tasks assigned to each release time. This table, along with interrupts, serves as input to Algorithm 1, where the scheduling starts at release time 0 ms. The heuristics can choose either the *Sensor* or *Control* task since they have no predecessors. However, due to the data dependency of *Control* task on the *Sensor* task, the *Sensor* task is selected first.

The *Sensor* task is scheduled with a finishing time of 4 ms (preempted by a 1 ms interrupt). The heuristics are called again, and this time the *Control* task is selected due to its triggering of the *Actuate* task. The *Control* task is scheduled with a finishing time of 13 ms (considering interference from the *Sensor* task and three instances of the interrupt, each 3 ms). Finally, the *Actuate* task is scheduled with a finishing time of 18 ms. Next, we schedule the release time at 20 ms, which contains only the *Sensor* task, which is scheduled and gets a finishing time of 24 ms.

At the 30 ms release time, there are two tasks to schedule. The *Control* task is scheduled first, finishing at 39 ms. When

attempting to schedule the *Actuate* task, it is found to be unschedulable at the current release time due to the next release time at 40 ms. Thus, the *Actuate* task is moved to the 40 ms release time. At 40 ms, we schedule the *Sensor* task first, finishing at 44 ms, followed by the *Actuate* task, finishing at 49 ms. With no more release times or tasks to schedule, the final schedule and schedulable result are returned. The schedule generated from the Rubus offline scheduler is shown in Fig. 7. The first column shows various instances of the tasks that are scheduled at the four release times. The second and third columns provide the best-case and worst-case execution times of the tasks respectively. The fourth and fifth columns show the deadlines and finishing times of all task instances. The sixth column lists the utilization of time-triggered tasks, whereas the seventh column provides the combined utilization of both time-triggered and interrupt-triggered tasks of the corresponding release time.

Release Time	Tasks Assigned
0 ms	Sensor Control Actuate
20 ms	Sensor
30 ms	Control Actuate
40 ms	Sensor

TABLE I

Name	BCET	WCET	Deadline	Finishing Time	TT Util	TT+IT Util
Releasetime 0	3 us	14 ms			70 %	90 %
Sensor	1 us	3 ms	20 ms	4 ms	15 %	20 %
Control	1 us	7 ms	30 ms	13 ms	50 %	65 %
Actuate	1 us	4 ms	30 ms	18 ms	70 %	90 %
Releasetime 20 ms	1 us	3 ms			30 %	40 %
Sensor	1 us	3 ms	40 ms	24 ms	30 %	40 %
Releasetime 30 ms	1 us	7 ms			70 %	90 %
Control	1 us	7 ms	60 ms	39 ms	70 %	90 %
Releasetime 40 ms	2 us	7 ms			35 %	45 %
Sensor	1 us	3 ms	60 ms	44 ms	15 %	20 %
Actuate	1 us	4 ms	60 ms	49 ms	35 %	45 %

Fig. 7: Schedule generated by the Rubus scheduler for the application in Fig. 6.

We also demonstrate that the Rubus scheduler can construct schedules for a large number of tasks in just a few seconds. Fig. 8 displays a Gantt chart of an actual schedule generated by the Rubus scheduler for approximately 700 time-triggered software circuits (TT tasks) within a proprietary ECU in a contemporary vehicle. The schedule length is 100 ms, which corresponds to the hyper-period of all tasks in the schedule. There are 10 release times, starting at 0 and occurring every 10 ms. Each task is uniquely colored in the schedule, with the remaining information abstracted for IP protection.

VI. Conclusion

In this paper, we presented the features, mechanisms and algorithms that form the core of the Rubus offline real-time scheduler, which is central to the Rubus tool suite. We validated the offline scheduler by modeling a segment of the software architecture of an industrial application using the Rubus Component Model and generating a feasible offline schedule with the scheduler.

The schedules generated by the presented scheduler are executed by the Rubus real-time operating system certified according to the ISO 26262 safety standard. The Rubus tool suite and its scheduler have been utilized in the vehicle industry for model- and component-based software development of resource-constrained embedded systems for over 25 years.

Since its introduction in 1998, the scheduler has evolved significantly, transitioning from pure Earliest Deadline First heuristics to incorporating various other heuristics and intricate features such as priorities, data dependencies, and the ability to distribute the schedule over the entire hyper-period of the software application. These enhancements help avoid front-heavy scheduling, benefiting lower-priority dynamic event-triggered software functions running in the background. These enhancements ensure that both critical and non-critical vehicle functions are executed reliably and economically. In conclusion, the Rubus tool suite, with its static real-time scheduler, has proven to be a robust solution for the vehicle industry, addressing the complexities of modern vehicle software.

Acknowledgment

We thank our partners, whose assistance and discussions have significantly contributed to the evolution of the Rubus scheduler over the years, particularly Volvo Construction Equipment and BAE Systems, Sweden. The work in this paper is partially supported by the Swedish Governmental Agency for Innovation Systems (VINNOVA) via the INTERCONNECT, PROVIDENT, AORTA & FLEXATION projects, and by the Swedish Knowledge Foundation via the SEINE project.

References

[1] L. Lo Bello, R. Mariani, S. Mubeen, and S. Saponara, "Recent advances and trends in on-board embedded and networked automotive systems," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 2, 2019.

[2] S. Ain, "Safeguarding the code that drives modern vehicles," *Cyber Defense Magazine*, 2024, https://www.cyberdefensemagazine.com/-safeguarding-the-code-that-drives-modern-vehicles.

[3] Land Rover Newsroom. https://media.jaguarlandrover.com/news/2019/04-/jaguar-land-rover-finds-teenagers-writing-code-self-driving-future.

[4] T. Vale, I. Crnkovic, E. S. de Almeida, P. A. da Mota Silveira Neto, Y. C. Cavalcanti, and S. R. de Lemos Meira, "Twenty-eight years of component-based software engineering," *Journal of Systems and Software*, vol. 111, pp. 128 – 148, 2016.

[5] K. Petersen, D. Badampudi, S. M. A. Shah, K. Wnuk, T. Gorschek, E. Papatheocharous, J. Axelsson, S. Sentilles, I. Crnkovic, and A. Cicchetti, "Choosing component origins for software intensive systems: In-house, cots, oss or outsourcing?—a case survey," *IEEE Transactions on Software Engineering*, vol. 44, no. 3, pp. 237–261, March 2018.

[6] L. Sha, T. Abdelzaher, K.-E. A. rzén, A. Cervin, T. P. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. P. Lehoczky, and A. K. Mok, "Real Time Scheduling Theory: A Historical Perspective," *Real-Time Systems*, vol. 28, no. 2/3, pp. 101–155, 2004.

[7] M. Becker, D. Dasari, S. Mubeen, M. Behnam, and T. Nolte, "End-to-end timing analysis of cause-effect chains in automotive embedded systems," *Journal of Systems Architecture*, vol. 80, pp. 104 – 113, 2017.

[8] The AUTOSAR Consortium, "Autosar technical overview," in *Version 4.3.*, May 2016, http://autosar.org.

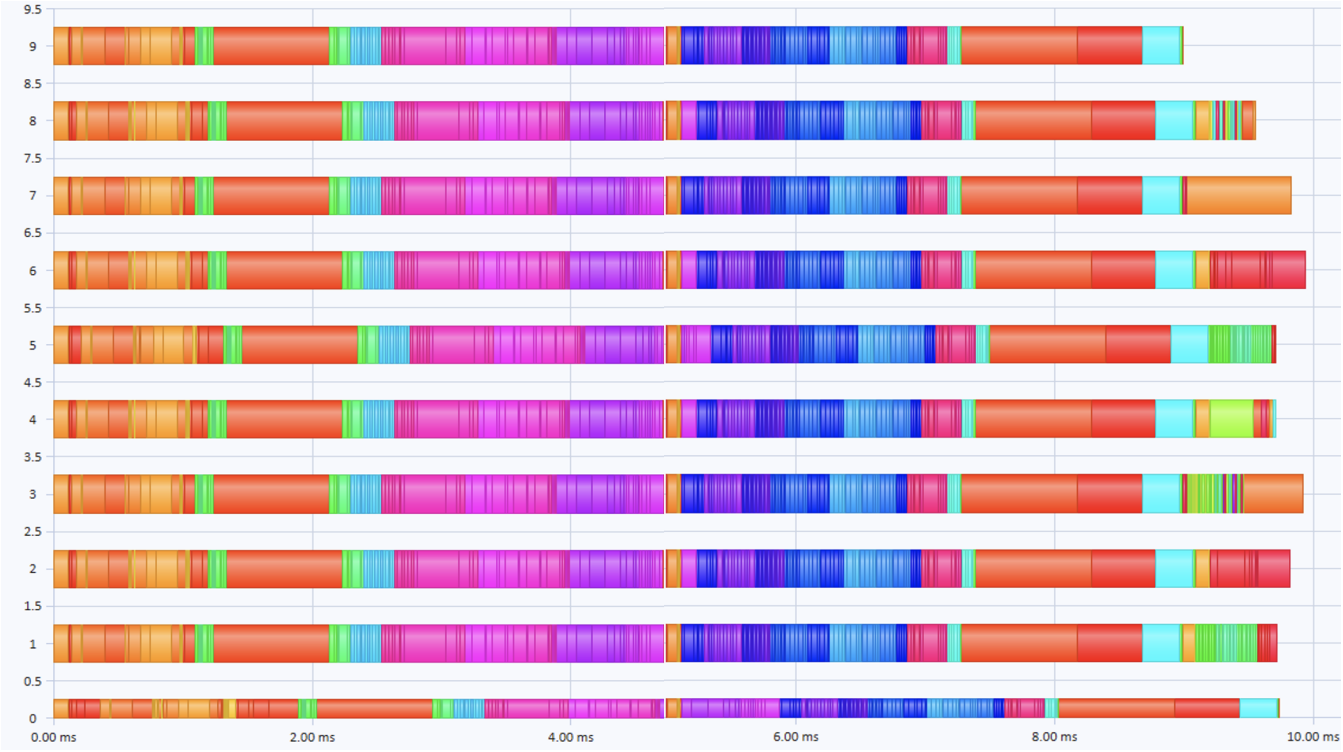

Fig. 8: A schedule generated by the Rubus scheduler for time-triggered tasks within a proprietary ECU in a modern vehicle.

[9] S. Fürst and M. Bechter, "Autosar for connected and autonomous vehicles: The autosar adaptive platform," in *46th Annual International Conference on Dependable Systems and Networks Workshop*, 2016.

[10] Fraunhofer ESK, Safe Adaptive Software for Fully Electric Vehicles, https://www.iks.fraunhofer.de/en/projects/safeadapt.html.

[11] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," *Computers and Digital Techniques*, vol. 152, no. 2, pp. 148–166, March 2005.

[12] S. Mubeen, H. Lawson, J. Lundbäck, M. Gålnander, and K. L. Lundbäck, "Provisioning of predictable embedded software in the vehicle industry: The rubus approach," in *IEEE/ACM 4th International Workshop on Software Engineering Research and Industrial Practice (SER&IP)*, 2017.

[13] International Organization for Standardization (ISO), *ISO 26262: Road Vehicles - Functional Safety*, ISO Std., 2011.

[14] Y. Laalaoui and N. Bouguila, "Pre-run-time scheduling in real-time systems: Current researches and artificial intelligence perspectives," *Expert Systems with Applications*, vol. 41, no. 5, pp. 2196–2210, 2014.

[15] A. Minaeva and Z. Hanzálek, "Survey on periodic scheduling for time-triggered hard real-time systems," *ACM Comput. Surv.*, vol. 54(1), 2021.

[16] K. Hänninen, J. Mäki-Turja, M. Nolin, M. Lindberg, J. Lundbäck, and K.-L. Lundbäck, "The Rubus Component Model for Resource Constrained Real-Time Systems," in *3rd IEEE International Symposium on Industrial Embedded Systems*, June 2008.

[17] S. Mubeen, "Timing predictability and performance standoff in component-based vehicle software on multi-core," in *20th IEEE International Conference on Software Architecture Companion (ICSA-C)*, 2023.

[18] S. Mubeen, J. Mäki-Turja, and M. Sjödin, " "Support for end-to-end response-time and delay analysis in the industrial tool suite: Issues, experiences and a case study," *Computer Science and Information Systems*, vol. 10, no. 1, 2013.

[19] J. Palencia Gutiérrez and M. Gonzáles Harbour, "Schedulability Analysis for Tasks with Static and Dynamic Offsets," in *Proc. 19^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 1998.

[20] J. Mäki-Turja and M. Nolin, "Fast and Tight Response-Times for Tasks with Offsets." in *17^{th} EUROMICRO Conference on Real-Time Systems*. IEEE, July 2005.

[21] S. Mubeen, J. Mäki-Turja, and M. Sjödin, "Integrating Mixed Transmission and Practical Limitations with the Worst-Case Response-Time Analysis for Controller Area Network," *Journal of Systems and Software*, pp. 66–84, 2015.

[22] L. Lo Bello, M. Ashjaei, G. Patti, and M. Behnam, "Schedulability analysis of time-sensitive networks with scheduled traffic and preemption support," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 153–171, 2020.

[23] D. B. Mateu, J. Proenza, A. Papadopoulos, T. Nolte, and M. Ashjaei, "An improved worst-case response time analysis for avb traffic in time-sensitive networks," in *45th IEEE Real-Time Systems Symposium*, 2024.

[24] N. Feiertag, K. Richter, J. Nordlander, and J. Jonsson, "A compositional framework for end-to-end path delay calculation of automotive systems under different path semantics," in *Workshop on Compositional Theory and Technology for Real-Time Embedded Systems*, December 2008.

[25] K. Sandström, C. Eriksson, and G. Fohler, "Handling Interrupts with Static Scheduling in an Automotive Vehicle Control System," in *Proc. of the 5^{th} International conference on Real-Time Computing Systems and Applications (RTCSA'98)*, 1998.

[26] K. Ramamritham, "Allocation and scheduling of complex periodic tasks," in *Proceedings.,10th International Conference on Distributed Computing Systems*, 1990, pp. 108–115.

[27] H. Kopetz, "Time-triggered real-time computing," *Annual Reviews in Control*, vol. 27, no. 1, pp. 3–13, 2003.

[28] L. Zhang, D. Goswami, R. Schneider, and S. Chakraborty, "Task- and network-level schedule co-synthesis of ethernet-based time-triggered systems," in *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2014, pp. 119–124.

[29] S. S. Craciunas and R. S. Oliver, "Smt-based task- and network-level static schedule generation for time-triggered networked systems," in *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, ser. RTNS '14, 2014, p. 45–54.

[30] A. Syed and G. Fohler, "Efficient offline scheduling of task-sets with complex constraints on large distributed time-triggered systems," *Real-Time Syst.*, vol. 55, no. 2, p. 209–247, Apr. 2019.

[31] A. Minaeva, B. Akesson, Z. Hanzálek, and D. Dasari, "Time-triggered co-scheduling of computation and communication with jitter requirements," *IEEE Transactions on Computers*, vol. 67, no. 1, 2018.

[32] G. Fohler, "Joint Scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems," in *Proc. 16^{th} IEEE Real-Time Systems Symposium (RTSS)*, December 1995, pp. 152–161.

[33] K. W. Tindell, A. Burns, and A. J. Wellings, "Allocating hard real-time tasks: an np-hard problem made easy," *Real-Time Syst.*, vol. 4(2), 1992.

[34] M. D. Natale and J. Stankovic, "Scheduling distributed real-time tasks with minimum jitter," *IEEE Transactions on Computers*, April 2000.

[35] J. Pasaje, M. Harbour, and J. Drake, "Mast real-time view: a graphic uml tool for modeling object-oriented real-time systems," in *Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS)*, 2001, pp. 245–256.

[36] J. Palencia and M. Harbour, "Exploiting precedence relations in the schedulability analysis of distributed real-time systems," in *Proceedings 20th IEEE Real-Time Systems Symposium*, 1999, pp. 328–339.