

Multi-Partner Project: Scheduling-Deployment Workflow for Autonomous RoboRacer Driving Stacks in the HAL4SDV Project

Matthias Stammler^{*1}, Henrik Scheidt^{*}, Tanja Harbaum^{*}, Jürgen Becker^{*}, Konstantin Dudzik^{†1}, Victor Pazmino[†], Federico Gavioli^{‡1}, Paolo Burgio[‡], Arvind Easwaran[§] and Andreas Eckel[¶]

^{*}Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, {stammler, henrik.scheidt, harbaum, becker}@kit.edu

[†]Forschungszentrum Informatik (FZI), Karlsruhe, Germany, {dudzik, pazmino}@fzi.de

[‡]Universita di Modena e Reggio Emilia, Modena, Italy, {paolo.burgio, federico.gavioli}@unimore.it

[§]Nanyang Technological University, Singapore, arvinde@ntu.edu.sg

[¶]TTTech Computertechnik, Vienna, Austria, andreas.eckel@tttech.com

Abstract—The European-funded HAL4SDV project aims to advance European solutions in software-defined vehicles by introducing a hardware abstraction layer positioned between executed software and execution units. HAL4SDV includes over 60 partners across 12 countries and receives funding within the Chips Joint Undertaking under Horizon Europe since April 2024 and is coordinated by TTTech Computertechnik. The proposed hardware abstraction layer includes safety-critical scheduling and platform deployment of software tasks, and is motivated by the requirement for abstracted hardware with unified interfaces in centralized automotive architectures.

This work presents a correct-by-construction workflow which is developed by academic partners to schedule and deploy periodic software tasks onto diverse execution units. The workflow facilitates the execution of the same task stack on multiple unit architectures and consists of a task model and scheduling algorithm, which is followed by platform deployment for diverse hardware units, ensuring safe execution. In this multi-partner project, a bandwidth regulation unit for hardware accelerators and a RISC-V-based multicore system with tightly coupled memories are used as target platforms.

A RoboRacer driving stack is chosen for evaluation, showing the viability of our workflow to schedule autonomous driving functions. To show generalization capability, synthetic task sets are additionally used to validate our deployment workflow.

Index Terms—Real-time Scheduling, Platform Deployment, Memory Bounding Unit, RISC-V, HAL4SDV

I. INTRODUCTION

Modern automotive vehicles observe a trend towards a centralized electric/electronic (E/E) architecture. This trend is motivated by the requirements from powerful multi-sensor functionality and the development of advanced software functionality [1]. Heterogeneous execution units are deployed inside these centralized execution platforms [2], supporting advanced driver assistance systems (ADAS) [3], [4], machine

learning based tasks like driver fatigue monitoring [5], and infotainment tasks in mixed-criticality systems [6].

This centralized E/E architecture facilitates the deployment of a hardware abstraction layer (HAL) to decouple hardware and software development [7]. The European research project *Hardware Abstraction Layer for Software-Defined Vehicles (HAL4SDV)* brings together more than 60 partners from 12 countries to collaborate on a unified HAL specification, contribute building blocks, and implement demonstrations [8]. One feature of the HAL is a reliable task scheduling and deployment method. International standards like ISO 26262 necessitate (timing) isolation between tasks, which is facilitated and guaranteed by supporting operating systems and hardware modules [9]. Schedulers using a correct-by-construction paradigm, and memory bounding units are deployed to support this isolation. Furthermore, phased execution is used to reduce timing interference between tasks by limiting the memory access of execution units. Configuring these memory units via a scheduler gives an upper bound on task latency.

A tool chain consisting of a task model and scheduler [10] and a deployment tool for a bandwidth regulation unit for custom accelerators [11], as well as for a RISC-V-based phased execution platform with tightly coupled memories [12], is developed in this multi-partnered collaboration in the context of the HAL4SDV project. This scheduling and deployment toolchain implements the correct-by-construction paradigm by providing an end-to-end solution.

As a testbed, we employ the 1:10 autonomous racing vehicles used in the RoboRacer competition [13], previously known as F1tenth. This project was born to research autonomous driving (AD) in racing contexts and is backed by a large research community. Races and competitions are frequently held at academic conferences, allowing for easy comparison between works on a unified testbed. RoboRacer provides an intermediate step for prototyping AD stacks before scaling to full-scale cars, which represents a reality-adjacent use case that brings designs to real-world driving vehicles.

In this work, we present the results from our academic

¹Shared first authorship for this work.

This work has received funding from the European Chips Joint Undertaking under Framework Partnership Agreement No 101139789 (HAL4SDV) including the national funding from the German Federal Ministry of Research, Technology and Space (BMFTR) under grant number 16MEE0467. The responsibility for the content of this publication lies with the authors.

cooperation in the context of the HAL4SDV project. We introduce an end-to-end toolchain of scheduler and platform deployment. The rest of this paper is structured as follows:

- We give a short overview of HAL4SDV and position this multi-partner collaboration in the project context (Section II).
- We introduce the used scheduling algorithm (Section III) and target platforms (Section IV, Section V).
- We provide an overview of the used evaluation in the form of a RoboRacer driving stack and generated synthetic task sets for generalizability (Section VI).

II. OVERVIEW

This multi-partner collaboration is part of the European-funded HAL4SDV project. In Section II-A, we give the context of the project. We explain the project structure in Section II-B and present an overview of the collaboration in Section II-C.

A. Context

The Software-Defined Vehicle (SDV) represents the shift from vehicle hardware as the unique selling point towards the software becoming the differentiating factor. This vehicle architecture aims to decouple software from physical components, enabling over-the-air (OTA) updates and lifecycle optimization through software. Shifting automotive architectures to SDVs enables the easier implementation of advanced driver assistance systems (ADAS) and other compute-intensive software components [8].

Additionally, SDV development and research aim to decouple software and hardware development, resulting in the requirement to introduce a hardware abstraction layer (HAL). This HAL aims to provide a standardized software interface that separates higher-level applications from the vehicle's computing platform. In the vehicle architecture, the HAL additionally abstracts sensors, actuators, and processing units by unifying APIs and interfaces. This allows application software to run independently of the hardware used, improving portability and future updates. This layer further aims to simplify system integration and accelerate deployment, as well as the reuse of software modules [8].

B. HAL4SDV Project

The HAL4SDV project aims to drive European innovation in next-generation software-defined vehicles by establishing unified software interfaces and development methodologies that decouple software from specific hardware platforms. Through this approach, HAL4SDV seeks to enable flexible and scalable software configuration for both safety-critical and non-safety-critical vehicle functions, laying the foundation for software-defined vehicles. The project aims to pioneer new methods, technologies, and processes for future vehicle development and makes use of advancements in microelectronics, communication technologies, software engineering, and artificial intelligence. Systems, Safety, Security, and Software are defined as the main pillars of the project. HAL4SDV aspires to

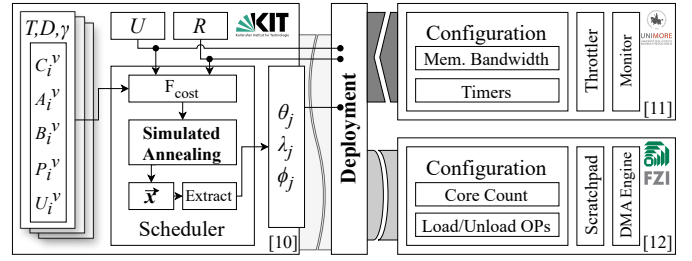


Fig. 1: Collaboration Structure. HM2GP [10] is used to generate schedules. Generated schedules are deployed onto memory bandwidth regulators [11] or RISC-V-based DMA engines [12].

shape the future of intelligent, adaptable, and secure automotive systems in Europe. The consortium comprises 53 partners and 9 associated partners from 12 countries, is coordinated by TTTech Computertechnik AG, and started in April 2024 with a funding period of three years. HAL4SDV is expected to conclude in April 2027. The scientific research in HAL4SDV is organized into six transversal activities (TAs). These are TA-A: *HW/SW Abstraction*, TA-B: *API and Interfaces*, TA-C: *Mixed Criticality Integration*, TA-D: *Cybersecurity Integration*, TA-E: *Process Development Tools*, and TA-F: *Integration, Testing, Simulation Tools*. Each transversal activity represents one collection of research topics relevant to software-defined vehicles. Our collaboration is positioned at the intersection of TA-A and TA-C and provides a mature workflow and use case evaluation of scheduling and deployment of tasks in mixed-criticality systems [8], [14].

C. This Collaboration

Our collaboration is positioned at the intersection of the application layer and the middleware layer, as well as at the intersection of hardware, virtualization, and the operating system. We combine the phased execution model and scheduler HM2GP [10] with the automatic deployment of these schedules onto a bandwidth regulator for custom accelerators [11] and a RISC-V-based phased execution platform [12].

We target the *Systems* and *Software* pillars of the project directly and capture the *Safety* pillar through a correct-by-construction approach. Our workflow implements a vertical slice with a model instance consisting of tasks, units, and resources as the input and an output consisting of a configuration, as well as deployment onto our hardware units. Fig. 1 shows the structure of this collaboration. A platform description as a set of execution units \mathcal{U} as well as the shared resources \mathcal{R} goes into building a cost function for a simulated annealing solver. Additionally, the task set \mathcal{T} for the current execution mode is used to generate the cost function. The result in the form of the start times, job versions, and execution units is extracted from the optimized \vec{x} vector. These are then forwarded together with the execution unit and resource description to the deployment. In the deployment component, the scheduler's outputs are mapped onto the target hardware units by generating configuration files for either the bandwidth regulator or DMA engines, respectively. Our collaboration thus

TABLE I: Model Components [10]

Model component	Description
$\mathcal{T} = \{\tau_i \mid i \in [0, N_{\mathcal{T}}]\}$	Task set
$\mathcal{U} = \{u_i \mid i \in [0, N_{\mathcal{U}}]\}$	Unit set
$\mathcal{R} = \{r_i \mid i \in [0, N_{\mathcal{R}}]\}$	Resource set
$D : \mathcal{T} \rightarrow \mathbb{Q}^+ \cup \emptyset$	Task period
$\gamma : \mathcal{T} \rightarrow [0, 1]$	Allowed Slack
$\mathcal{M}_n : \mathcal{T} \rightarrow \mathcal{P}(\mathbb{N})$	n th execution mode description
$C_i^v \in \mathbb{Q}^+$	WCET estimation of task t_i , version v
$A_i^v : R \rightarrow \mathcal{P}(\mathbb{Q}^+)$	Access phase of task t_i , version v
$B_i^v : R \rightarrow [0, 1]^{A_i^v(r)}$	Mem. bounds for acc. phases of τ_i , vers. v
$P_i^v : \mathcal{T} \rightarrow \mathcal{P}(\mathbb{Q}^+)$	Precedence constraints of task t_i , version v
$U_i^v \subseteq \mathcal{U}$	Valid exec. units for task t_i , version v
t_P	Hyperperiod
$\theta_j : \mathcal{T} \rightarrow [0, t_P)$	j th start time (job start)
$\lambda_j : \mathcal{T} \rightarrow [0, N_V)$	Chosen version for j th job
$\phi_j : \mathcal{T} \rightarrow [0, N_U)$	Chosen execution unit for j th job

acts as an adaptation mechanism in modeling real-world execution platforms (from [11], [12]) and subsequently generates platform deployment configuration from model instances.

This *deployment* and the adaptation module between the scheduler and hardware units are the main contributions of our collaboration inside the HAL4SDV project. An introduction of the scheduler is provided in Section III, followed by an overview of the memory bandwidth regulator (Section IV) and the phased execution platform (Section V) as well as the corresponding platform architectures. Throughout this paper, we highlight the collaboration points and interfaces in *italics*.

III. TASK SCHEDULING

The used scheduler is an extension of HM2GP [10]. HM2GP supports the modeling of heterogeneous execution units, multiple shared resources, multi-version tasks, gang tasks, and precedence constraints. We extend this work to support multiple execution modes. Table I shows an overview of the scheduling model.

This model consists of a task set \mathcal{T} , a unit set \mathcal{U} , and resources \mathcal{R} . Tasks are assumed to be periodic. A task subset is assigned a period D with a given allowed slack γ . If a task is not assigned a deadline, our tool does not inherently schedule the task. Because tasks are periodic, the period can be interpreted as an implicit deadline. Execution modes are introduced via \mathcal{M}_n . \mathcal{M}_n specifies the n th execution mode by assigning a list of available versions to each task $\tau_i \in \mathcal{T}$, for the given mode. Each version v of each task $\tau_i \in \mathcal{T}$ is described by five functions. The first is a WCET estimation C_i^v . A list of resource access phases with arbitrary time placement is given by A_i^v . Each resource access phase is bounded by B_i^v with respect to resource accesses. In each time step, the phase of the task uses between 0 and all available resource accesses. Each phase in $A_i^v(r)$ is given a value in $[0, 1]$, specifying these accesses. A list of precedence constraints P_i^v describes the allowed time difference between the start of each job of two tasks. The valid execution unit for this version is given by U_i^v . With these descriptions, the scheduler creates a schedule by setting $\theta_j(\tau_i)$, $\lambda_j(\tau_i)$, and

$\phi_j(\tau_i)$ for each job j of task τ_i . Here, θ_j describes the start time of the j th job, λ_j describes the chosen version of the j th job, and ϕ_j sets the execution unit of the j th job.

The schedule creation is constrained to bound the sum of all resource accesses at each time step to 1 and prevent the simultaneous execution of tasks on the same execution unit. This is facilitated by ensuring that for every resource $r \in \mathcal{R}$, at each time step $t \in [0, t_P)$, the sum of all values in B_i^v , summed up over all access phases A_i^v of all jobs j of all tasks τ_i scheduled in that time step, does not exceed 1. We furthermore constrain the schedule to ensure that each job j of each task τ_i is scheduled exclusively on the given execution unit $\phi_j(\tau_i)$, by forbidding that the execution times of two jobs overlap on the same execution unit. The schedule is constrained to ensure that each job j of a task τ_i , which is necessitated via a precedence constraint, has scheduled jobs of other tasks according to the given precedence constraints $P_i^{\lambda_j(\tau_i)}$ of the scheduled version $\lambda_j(\tau_i)$. Lastly, we ensure that the given execution unit $\phi_j(\tau_i)$ is in the valid execution units for the given version.

The concrete schedule (as a configuration of θ_j , λ_j , and ϕ_j) is created by the following steps. First, we generate the hyper period t_P as the least common multiple of all task periods. Afterward, we calculate the number of scheduled jobs for each task $N_{\theta}(\tau_i)$ as $t_P/D(\tau_i)$ for jobs τ_i with $D(\tau_i) \neq \emptyset$. For tasks with $D(\tau_i) = \emptyset$, N_{θ} is calculated as the maximum number of the sum of defined precedence constraints between τ_i and all other tasks as $N_{\theta}(\tau_i) = \sum_{\tau_x \in \mathcal{T} \setminus \{\tau_i\}} \max_v(|P_x^v(\tau_i)|) \cdot N_{\theta}(\tau_x)$. To avoid a circular dependency for calculating N_{θ} , the precedence constraints of a task must not indirectly or directly reference the same task itself.

With this, we can use the vector

$$\vec{\mathbf{x}} = [-\theta_j(\tau_i) \mid -\lambda_j(\tau_i) \mid -\phi_j(\tau_i) \mid -]$$

as the solution vector, with $i \in [0, N_{\mathcal{T}})$, and $n \in [0, N_{\theta}(\tau_i))$, and $|\vec{\mathbf{x}}| = 3 \cdot \sum_{\tau \in \mathcal{T}} N_{\theta}(\tau)$, in schedule creation. In $\vec{\mathbf{x}}$, every element in the first third $\theta_j(\tau_i)$ specifies the start time of the j th job of the i th task, the version of the j th job start is given in $\lambda_j(\tau_i)$ and the used execution unit is specified by $\phi_j(\tau_i)$.

Because the given constraints are nonlinear and non-continuous, a simulated annealing solver is chosen and the given constraints are formulated as penalty terms in the cost function. Our cost function F_{cost} is defined as:

$$F_{\text{cost}}(\vec{\mathbf{x}}) = w_{\text{ov}} \cdot p_{\text{ov}}(\vec{\mathbf{x}}) + w_{\text{ac}} \cdot p_{\text{ac}}(\vec{\mathbf{x}}) + w_{\text{pr}} \cdot p_{\text{pr}}(\vec{\mathbf{x}}) + w_{\text{u}} \cdot p_{\text{u}}(\vec{\mathbf{x}}) - w_{\text{oc}} \cdot p_{\text{oc}}(\vec{\mathbf{x}}) - w_{\text{ty}} \cdot p_{\text{ty}}(\vec{\mathbf{x}}).$$

Here, w_* denotes the weights of the penalty and bonus functions. With the $p_*(\vec{\mathbf{x}})$ we model the constraints as well as give bonuses. The term $p_{\text{ov}}(\vec{\mathbf{x}})$ adds a penalty if two execution times overlap on the same execution unit, $p_{\text{ac}}(\vec{\mathbf{x}})$ adds a penalty if the sum of all accesses towards a shared resource exceeds 1, and $p_{\text{pr}}(\vec{\mathbf{x}})$ adds a penalty if the precedence constraints between two tasks are not met. The penalty $p_{\text{u}}(\vec{\mathbf{x}})$ adds a penalty, should a job version not match the specified

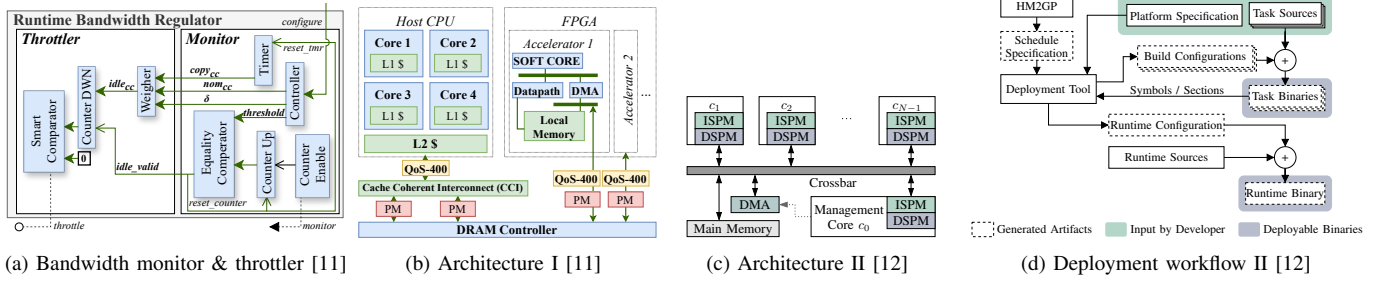


Fig. 2: Deployment architectures and workflows.

execution units. The bonus term $p_{oc}(\vec{x})$ gives a bonus if an execution unit is left without scheduled jobs, and $p_{ty}(\vec{x})$ gives a bonus for task timeliness.

Because we ensure that the smallest penalty is not exceeded by the largest possible bonus, we can guarantee that a value of $F_{cost}(\vec{x}) \leq 0$ signifies a valid schedule because a value of $F_{cost}(\vec{x}) > 0$ signifies that at least one constraint is not met. Once $F_{cost}(\vec{x})$ reaches a predetermined value, optimization can be stopped and the generated schedule can be classified as valid. This scheduling is repeated N_M times, once for every execution mode. A task τ_i is scheduled, if $M_n(\tau_i) \neq \emptyset$ for $n \in [0, N_M)$. After a schedule is created for each execution mode, transitions are created using a similar approach.

This schedule generation for multi-version tasks on heterogeneous execution platforms is based on HM2GP [10]. From the output of the scheduler (the combination of θ_j , λ_j , and ϕ_j), configuration files for the bandwidth regulator [11] or RISC-V-based platform [12] are generated in the deployment step.

IV. MEMORY BANDWIDTH REGULATION UNIT

Memory interference emerged as the main showstopper for predictability in safety-critical (hard real-time) systems [15], [16]. Several solutions have been proposed to tackle this at different levels of the memory hierarchy, i.e., L1/L2 caches [17], shared buses and memory banks [18].

In the considered heterogeneous multicore platforms, shared memory banks between the multicore host subsystem and energy-efficient accelerators (e.g., NVIDIA GPUs, Neural Processing Units, or FPGAs) emerged as the main bottleneck. These are introduced by collisions by simultaneous memory accesses and lead to delays. Delays, resulting from these collisions, need to be mitigated or limited. To tackle this, state-of-the-art techniques rely on synergistic bandwidth *monitoring* and *throttling* mechanisms. These emerged as an effective way of limiting the allowed bandwidth for a particular computing element, typically at the granularity of small bursts of few hundred bytes. With this, we limit the available bandwidth to each computing unit at a given time and reduce possible delays. We employ recent research in the field [11], namely a special hardware device that can be instantiated on custom accelerators inside FPGAs that, in the domain of the HAL4SDV project, is a COTS System-on-Chip by the AMD

Xilinx Ultrascale+ family [19]. Fig. 2a and Fig. 2b show the schematic of this device, and its integration into the first target platform, respectively.

The configuration API of these components lets designers transparently specify the memory requirements of each real-time task and each scheduled job. Reconfigurability is completely managed by the accelerator cores, and happens at the finest grain of a few tens of microseconds, and can thus be used in synergy with most host-side techniques, such as the level of real-time OS scheduler, cache partitioning, bus arbiters, and memory controllers.

In this collaboration, the use of HM2GP automates the configuration by generating the previously manually configured schedule, which in our case consists of θ_j , λ_j , and ϕ_j . For tasks τ_i which can be adjusted during runtime, i.e., their memory bandwidth can change, schedules featuring multiple task versions (via different λ_j) are generated. These schedules represent different execution modes and can be exchanged during runtime.

V. RISC-V-BASED PHASED EXECUTION PLATFORM

This section gives a short overview of our second target platform, a phased execution platform developed in previous work [12]. This platform comprises a RISC-V multicore system with tightly coupled scratchpad memories, as depicted in Fig. 2c, and a deployment tool that automatically generates the deployment artifacts based on the schedule specification generated by the HM2GP scheduling tool.

The execution unit's access to shared memory is restricted to mutually exclusive *memory phases* to prevent interference, while computation is limited to memories local to each core. Memory transfers are orchestrated by a runtime environment, which is mapped onto a dedicated management core. To improve memory transfer performance, the platform features a dedicated DMA engine. The hardware platform is generated using the Chipyard framework [20], allowing a wide range of configurations with respect to the on-chip memory capacity and core count. For this work, a configuration of four Rocket cores [21], each with a 120 KiB instruction scratchpad and a 256 KiB data scratchpad, is implemented on a VCU118 FPGA board [22].

The input to the deployment tool comprises the schedule description of one hyper period given by $\theta_j(\tau_i)$, $\lambda_j(\tau_i)$, and

$\phi_j(\tau_i)$ for all $\tau \in \mathcal{T}$ as well as the source code for all scheduled tasks and their respective versions. The resource set for this platform \mathcal{R} is limited to the main memory of the platform, which implicitly includes the shared interconnect. Furthermore, since each memory phase is exclusively mapped to one core, the resource access bounds are limited to $\{0, 1\}$. Each job includes a resource access phase to load the specified version of the task into the scratchpad memory of the respective execution unit. This load operation must occur within the timespan during which the execution unit is occupied by said job. The configuration of this access phase is handled automatically by the deployment tool. However, for subsequent access phases to exchange data between the local memory of the execution unit and the main memory, the developer must provide the configuration of the DMA transfers contained in each access phase. Each transfer is described by its source and destination addresses, as well as the size of the transfer. A partition is reserved in main memory to facilitate these load or unload operations.

From these inputs, the deployment tool generates the deployment artifacts as shown in Fig. 2d. This includes the task binaries as well as a runtime environment to facilitate the scheduling of all data transfers necessary for the resource access phases. In a first step, the specified versions of each task are compiled for all execution units on which they are scheduled. The resulting binaries are linked to be initially loaded into main memory but executed from the tightly coupled memories of the respective core. In the second step, the runtime configuration is generated based on the input provided by the scheduling tool described above. The information necessary to configure the tasks' load operations is extracted directly from the generated task binaries. A list of all resource access phases and their configurations within one hyper period defines the system's operation as executed by the runtime.

VI. USE CASE AND GENERALIZABILITY

We evaluated our approach on a RoboRacer task stack and tested the generalizability using synthetic task sets.

A. RoboRacer Use Case

Our target application scenario is the RoboRacer autonomous racing competition. Teams compete in head-to-head races with the goal of navigating the circuit as fast as possible while tracking and safely overtaking their opponent. This competition defines a reference platform based on a 1:10 scaled vehicle equipped with multiple sensors (a 2D LiDAR rangefinder, a stereoscopic camera, and an inertial measurement unit) and actuators (a motor controller and a steering servomotor). The platform is a valuable tool for research and teaching in autonomous vehicles (AVs), as it closely resembles a real platform without incurring the high costs and extensive space requirements associated with full-scale AV prototypes.

The autonomous racing stack provided by UNIMORE [23] is developed on top of Robot Operating System 2 (ROS2)

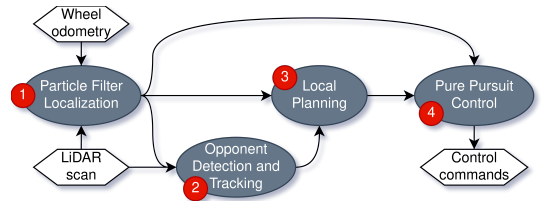


Fig. 3: The dependency graph of the proposed autonomous racing stack.

[24], a well-known robotic development framework that provides common abstractions for computational units and a performance-oriented inter-process communication runtime.

The autonomous stack relies on a global map approach to localize inside the racetrack. In a first offline phase, a mapping software generates a map using sensor recordings of the vehicle inside the racetrack. This map is then used in a minimum-time offline race trajectory optimization software in order to generate an optimal reference race line. The resulting map and race line are the only pre-computed inputs to the real-time embedded racing stack. It follows a perception-planning-control paradigm, where the vehicle first perceives the environment, then plans its movement in both space and time, and finally calculates control commands to follow the planned trajectory. Given the requirements of the competition, the embedded autonomous racing stack (Fig. 3) is defined in the following task set:

1) *Particle Filter Localization*: In order to localize the vehicle in the map, we leverage a particle filter: a Monte Carlo-based random sampling algorithm. This task activates each time a new LiDAR scan is available, sampling pose candidates (particles) around the previously estimated pose. Given the precomputed map, the algorithm models the sensor reading of each particle and compares it with the real LiDAR point cloud, calculating a similarity-based score for each candidate. The output of this computational node is the average of the particle set weighted on the similarity score, which represents the estimated 2D pose of the vehicle. This localization result is a dependency of all subsequent tasks.

2) *Opponent Detection and Tracking*: Given the latest LiDAR scan and the localization output, the detection task splits the input point cloud into clusters, which are subsequently filtered and classified to detect opponent vehicles. The detected vehicles are associated and tracked over time and pushed forward to the local planning algorithm.

3) *Local planning*: The local planner leverages the pre-computed map, the reference optimal trajectory, and detected opponents to generate a quick local path, avoiding potential collisions with the opponents while also aiming to minimize the lap time. To do so, this node models the racetrack space on a field-based representation, modeling the globally optimal path with attractive fields and the tracked opponents as repulsive fields. From this intermediate representation, the local planner generates a fast collision-free local path, which will be followed by the control task.

TABLE II: RoboRacer Task Stack

Task	Exec. Time	Prec. Constr.
τ_1 Particle Filter Localization	24 ms	\emptyset
τ_2 Opponent Detection and Tracking	13 ms	τ_1
τ_3 Local Planning	24 ms	τ_1, τ_2
τ_4 Pure Pursuit Control	2 ms	τ_1, τ_3

TABLE III: Generalizability Experiments

Experiment	(1)	(2)	(3)	(4)
$N_{\mathcal{T}}$	60..120	60	60	112
$N_{\mathcal{U}}$	6	6	6	8
Unit Types	2,3	2,3	2,3	1
Acc. Win.	$N_{\mathcal{T}}$	$N_{\mathcal{T}}$	$N_{\mathcal{T}}$	2..50
$N_{\mathcal{R}}$	2	2	2	1..4
Mult. Ver. [%]	10	0..50	0	0
Pre. Con. [%]	0	0	0..60	0
Util. [%]	60,70	60,70	60,70	70
Alg. Runtime [s]	460..6816	600 \pm 100	550 \pm 80	320 \pm 80

4) *Pure Pursuit Control*: A well-known Pure Pursuit controller [25] is the last node in the vehicle’s control graph. This node depends on the current position of the vehicle and the local path calculated during the local planner stage. It geometrically calculates the required control commands (engine speed and steering angle) to navigate the local path in a fast and safe manner.

Since the task set needs to work on real-time sensor data, the task activation occurs at the period of the input data streams. The LiDAR data stream is modeled with a period of 25ms between each activation. The precedence constraints are modeled according to the data dependencies shown in Fig. 3 and Table II. The opponent detection and tracking task is activated once the particle filter localization is completed. The local planning task is activated once the opponent detection and tracking are completed. Finally, the control task is activated once all other nodes terminate their executions.

B. Generalizability

We evaluated the generalizability of our approach by generating and deploying synthetic task sets with varying model characteristics. These task sets were generated to be schedulable on our target platforms and were created by generating $N_{\mathcal{T}}/N_{\mathcal{U}}$ random time periods for each execution unit, totaling t_P . For half of these tasks, we ensure that we generate two start times with a distance of $t_P/2$. For a quarter, we generated three start times with a $t_P/3$ distance, and for the last quarter, we generated one start time, introducing variability in task periods. Afterward, we generated the specified amount of access phase starts. If an access phase start was positioned during a task with multiple start times, the access window was replicated to all other executions of this task. If necessary, other access phases were moved. The predefined percentage of tasks were connected via precedence constraints that equaled 0.25 to 1.75 of the difference in generated start time. For the given percentage of tasks, we generated alternative versions with an execution time equal to 1.5 times the original execution time. All values were then multiplied with the utilization value, resulting in processor idle time. Table III shows the

varied characteristics and the algorithm runtime for generating schedules. Experiment (1) varied the task amount, (2) varied the number of task versions, (3) varied the ratio of tasks with precedence constraints, and (4) varied the number of shared resources.

Our approach had a success rate of 100 % in finding schedules for the aforementioned schedulable task sets. Because of this, we observed the algorithm runtime as being the indicator for the complexity in creating the schedules. We observed an increase in algorithm runtime with an increase in task amount and observed a maximum runtime of 6816 s for 120 tasks in experiment (1). An increase in multi-version task ratio or an increase in the number of precedence constraints did not lead to a considerable runtime penalty, as shown in experiments (2)-(4) in Table III [10]. To evaluate the deployment, dummy tasks are created to facilitate the phased execution models. These tasks are created to adhere to the generated task descriptions introduced above and deployed to the RISC-V-based phased execution platform [12]. Memory phases are implemented via repeated writing and reading of dummy values, and execution phases are facilitated by incrementing variables until a given maximum value.

VII. CONCLUSION

We describe a collaboration implementing the combination of components from KIT, UNIMORE, and FZI in the context of the European-funded HAL4SDV project. The HAL4SDV project is currently undergoing the full research and implementation phases after its second project year ended in April 2026. We present a mature collaboration, which describes an evaluated workflow and highlights the collaborative efforts.

Our work combines a multi-version task scheduler for heterogeneous execution platforms and an accompanying task model with the deployment of generated schedules onto a bandwidth regulation unit for custom accelerators, as well as on a RISC-V-based phased execution platform. The scheduler takes a description of tasks as well as the platform description in the form of execution units and shared resources as the input. The output of the scheduler is the job starting times, task versions, and execution units. Depending on the input and the chosen platforms, these outputs are then turned into configuration files for the presented hardware units.

Our work is positioned in HAL4SDV’s Transversal Activities (TA)-A: HW/SW abstraction as well as TA-C: mixed-criticality integration. We tested our method for a RoboRacer task stack and evaluated the generalizability on synthetic task sets. We were able to schedule the tasks onto the deployed platforms and achieved a success rate of 100 % for the generalizability tests. The schedule algorithm runtime was measured between 240 s and 6816 s, depending on configuration, and scaled with the amount of scheduled tasks. Scheduling and deploying the RoboRacer task stack furthermore allowed us to evaluate the real-world applicability of our workflow for autonomous driving vehicles. Our collaboration provides an input into the HAL4SDV project environment by implementing a vertical slice in the platform deployment.

REFERENCES

- [1] V. Bandur *et al.*, “Making the Case for Centralized Automotive E/E Architectures,” *IEEE Transactions on Vehicular Technology*, vol. 70, pp. 1230–1245, Feb. 2021.
- [2] W. Wang *et al.*, “Review of Electrical and Electronic Architectures for Autonomous Vehicles: Topologies, Networking and Simulators,” *Automotive Innovation*, vol. 7, pp. 82–101, Feb. 2024.
- [3] S. Teng *et al.*, “Motion Planning for Autonomous Driving: The State of the Art and Future Perspectives,” *IEEE Transactions on Intelligent Vehicles*, vol. 8, pp. 3692–3711, June 2023.
- [4] A. Moujahid *et al.*, “Machine Learning Techniques in ADAS: A Review,” in *2018 International Conference on Advances in Computing and Communication Engineering (ICACCE)*, (Paris), pp. 235–242, IEEE, June 2018.
- [5] S. Sinaei, M. Mohammadi, R. Shrestha, M. Alibeigi, and D. Eklund, “PRIV-DRIVE: Privacy-Ensured Federated Learning using Homomorphic Encryption for Driver Fatigue Detection,” in *2024 27th Euromicro Conference on Digital System Design (DSD)*, (Paris, France), pp. 427–434, IEEE, Aug. 2024.
- [6] Z. Jiang *et al.*, “Re-Thinking Mixed-Criticality Architecture for Automotive Industry,” in *2020 IEEE 38th International Conference on Computer Design (ICCD)*, (Hartford, CT, USA), pp. 510–517, IEEE, Oct. 2020.
- [7] O. Burkacky, J. Deichmann, and J. P. Stein, *Automotive Software and Electronics 2030: Mapping the Sector’s Future Landscape*. New York: McKinsey & Company, 2023.
- [8] M. Paulweber, A. Eckel, and P. Azzoni, “Multi-partner project: Shaping the Vehicle of the Future: How FEDERATE and HAL4SDV are steering Europe’s Software-Defined Vehicle Ecosystem,” in *2025 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (Lyon, France), pp. 1–6, IEEE, Apr. 2025.
- [9] “Road vehicles — Functional safety,” standard, International Organization for Standardization, Geneva, CH, Dec. 2018.
- [10] M. Stammler, F. Lesniak, N. Kumar, A. Easwaran, and J. Becker, “HM2GP: A Multi-Version task scheduler with extended precedence constraints on COTS SoCs,” in *2025 IEEE 38th International System-on-Chip Conference (SOCC) (SOCC 2025)*, (Dubai, United Arab Emirates), p. 5.99, Sept. 2025.
- [11] G. Brilli, G. Valente, A. Capotondi, P. Burgio, T. Di Masciov, P. Valente, and A. Marongiu, “Fine-Grained QoS Control via Tightly-Coupled Bandwidth Monitoring and Regulation for FPGA-based Heterogeneous SoCs,” in *2023 60th ACM/IEEE Design Automation Conference (DAC)*, (San Francisco, CA, USA), pp. 1–6, IEEE, July 2023.
- [12] K. Dudzik, M. Kirschner, and J. Becker, “Automated deployment of Real-Time tasks for phased execution on Scratchpad-Based multi-core platforms,” in *18th IEEE International Symposium on Embedded Multicore/Manycore SoCs (MCSoc-2025) (18th IEEE MCSoc-2025)*, (Singapore, Singapore), p. 7.74, Dec. 2025 (accepted).
- [13] M. O’Kelly, V. Sukhil, H. Abbas, J. Harkins, C. Kao, Y. V. Pant, R. Mangharam, D. Agarwal, M. Behl, P. Burgio, and M. Bertogna, “F1/10: An open-source autonomous cyber-physical platform,” 2019.
- [14] Teraglobus, “Hardware Abstraction Layer for a European Software Defined Vehicle Approach - HAL4SDV [Online],” 2024. Available under: <https://www.hal4sdv.eu/about> [Accessed: 29 October 2025].
- [15] N. Capodiceci, R. Cavicchioli, I. S. Olmedo, M. Solieri, and M. Bertogna, “Contending memory in heterogeneous SoCs: Evolution in NVIDIA Tegra embedded platforms,” in *2020 IEEE 26th International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 1–10, 2020.
- [16] G. Brilli, A. Capotondi, P. Burgio, and A. Marongiu, “Understanding and Mitigating Memory Interference in FPGA-based HeSoCs,” in *2022 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, (Antwerp, Belgium), pp. 1335–1340, IEEE, Mar. 2022.
- [17] S. Mittal, “A survey of techniques for cache partitioning in multicore processors,” *ACM Comput. Surv.*, vol. 50, May 2017.
- [18] R. Mancuso, R. Pellizzoni, M. Caccamo, L. Sha, and H. Yun, “Wcet(m) estimation in multi-core systems using single core equivalence,” in *2015 27th Euromicro Conference on Real-Time Systems*, pp. 174–183, 2015.
- [19] Xilinx, *Zynq UltraScale+ Technical Reference Manual*, 2022.
- [20] A. Amid *et al.*, “Chipyard: Integrated design, simulation, and implementation framework for custom socs,” *IEEE Micro*, vol. 40, no. 4, pp. 10–21, 2020.
- [21] K. Asanović *et al.*, “The rocket chip generator,” Tech. Rep. UCB/EECS-2016-17, Apr 2016.
- [22] Xilinx, *VCU118 Evaluation Board User Guide*, 2023.
- [23] F. Gavioli, F. Moretti, A. Russo, A. Capotondi, and P. Burgio, “Thundershot: an open-source autonomous vehicles research platform for embedded heterogeneous mpsoCs,” in *Proceedings of the 22nd ACM International Conference on Computing Frontiers: Workshops and Special Sessions, CF ’25 Companion*, (New York, NY, USA), p. 26–29, Association for Computing Machinery, 2025.
- [24] S. Macenski, T. Foote, B. Gerkey, C. Lalancette, and W. Woodall, “Robot operating system 2: Design, architecture, and uses in the wild,” *Science Robotics*, vol. 7, no. 66, p. eabm6074, 2022.
- [25] C. Coulter, “Implementation of the pure pursuit path tracking algorithm,” 1992.