
# Physics-Informed Mixed-Criticality Scheduling for F1Tenth Cars with Preemptable ROS 2 Executors

Kurt Wilson, Abdullah Al Arafat, John Baugh, Ruozhou Yu, Zhishan Guo
North Carolina State University
{kwilso24, aalaraf, jwb, ryu5, zguo32}@ncsu.edu

2025 IEEE 31st Real-Time and Embedded Technology and Applications Symposium (RTAS) | 979-8-3315-4340-2/25/$31.00 ©2025 IEEE | DOI: 10.1109/RTAS65571.2025.00030

***Abstract*—Autonomous systems are increasingly used in safety-critical domains, including industrial automation, autonomous vehicles, and the industrial Internet of Things. Verifying both the functional and temporal correctness of these systems is essential to ensure safety before deployment. However, end-to-end verification is challenging due to the interaction of continuous-time physical processes with discrete-time computational systems. Existing formal methods often assume simplified or static computational models, while traditional real-time systems focus on meeting timing constraints without explicitly linking them to physical safety. We address this gap by proposing a physics-informed mixed-criticality (MC) verification framework for cyber-physical systems, which allows the integration of computational and physical models for dynamic, fine-grained safety assurance. Our framework incorporates feedback from the local environment to guide criticality-based mode switching, ensuring adaptive responses to real-time physical states rather than relying on global worst-case assumptions. We demonstrate the feasibility of our approach with a prototype implementation on an autonomous F1Tenth vehicle using preemptive EDF scheduling on ROS 2. Verification is conducted using UPPAAL to validate system behavior, mode transitions, and physical safety constraints. Results show that our framework effectively manages MC requirements, enhancing responsiveness and safety in dynamic environments.**

***Index Terms*—Mixed-Criticality Systems, Temporal Verification, Formal Methods, F1Tenth cars, UPPAAL**

## I. Introduction

Cyber-physical systems, such as industrial control systems, autonomous vehicles, and the industrial Internet of Things, autonomously and seamlessly interact with the physical world, making them safety-critical in the sense that failure can lead to catastrophic effects on human lives and physical well-being. Therefore, verifying the temporal and functional correctness of these systems before deployment is essential. Traditional safety analysis of such systems typically treats timing requirements and functional correctness in the physical world as two separate steps [1]: (i) verifying physical correctness using formal methods and deriving safe timing bounds from control theory and (ii) designing and verifying computing systems independently to meet these timing bounds. While such approaches are widely used for verifying these systems, they often result in overly conservative designs, as all timing bounds are derived based on worst-case scenarios, including the operating environment.

Mixed criticality (MC) systems [2] deal with changes in the execution time of tasks in the system by considering more than one worst-case execution time (WCET) model for each task depending on its criticality to the system's safe operation. For instance, in a dual critical system, in normal operation, the scheduler assumes all tasks use the lower of the two WCETs. Once a critical task in the system exceeds its expected execution time without signaling completion, the scheduler performs a mode switch. The mode switch changes the properties of some tasks in the system to allow the critical task to be completed before its deadline. In this work, we consider changes in the *physically required reaction time* instead of temporal overrun of critical tasks as the indicator of mode switch requirement. A controller may only be able to ensure system safety in most states with a lower reaction time than is required in some other states. For example, finding a single timing bound for the system in Fig. 1 will result in a value lower than is required for most scenarios, limiting the time available to perform other tasks on the same system.

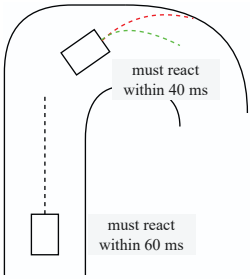



Fig. 1: *An autonomous vehicle shown at two positions on a racetrack, illustrating varying event reaction time requirements for safety. On a straight section of track, the controller can safely operate with a 60 ms reaction time. However, on a curved section, a 40 ms reaction time is necessary to maintain safety. Traditional verification methods would mandate a uniform 40 ms reaction time for the entire track. By using mixed-criticality scheduling, however, a dual-critical system can be devised, allowing the vehicle to operate in one mode on straight segments and switch to a more demanding mode on curves.*

**Contribution.** We propose an MC system where the criticality source is the variations in environmental states. Hence, we propose using multiple timing bounds that are determined by environmental properties and the overall system state. Specifically, the system should use one timing bound for scheduling the system tasks during normal operation and then identify another set of timing bounds for extreme/emergency scenarios, with the goal of reducing the system reaction time to allow it to

2642-7346/25/$31.00 ©2025 IEEE
DOI 10.1109/RTAS65571.2025.00030

Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

reach a safe state. By formally modeling the physical system, controller, and environment, we can identify situations where temporarily reducing the reaction time allows the system to recover safely. We show that this system of scheduler mode switching can be combined with a Simplex [3], [4]-like system, where the system switches to a known-reliable controller when a potentially unreliable controller places the system in an unsafe state.

We demonstrate our proposed MC framework on an F1Tenth [5] car, as it allows us to demonstrate real-world algorithms and software on a small-scale system and is commonly used for research in path planning [6], perception [7], and control [8]. By using a formal model of the car's driving behavior, a racetrack environment, and two controllers, we find a lower reaction time bound that allows the car to safely navigate a racetrack. We then find a second higher timing bound that allows the car to drive in *most* scenarios but crashes in narrow spaces and around corners. We show by determining when the system is entering one of these unsafe states, switching to the lower timing bound allows the system to return to a safe state.

To implement scheduler mode switching on a realistic system, we create a custom ROS 2 executor that runs tasks with the earliest deadline first (EDF) scheduler and supports changing the task parameters at runtime. We also add support for full preemption, which is not included in the default ROS 2 system, adding more flexibility in the tasks that can be run. By implementing scheduler mode switching in ROS 2 and demonstrating that environmentally-driven mode switches can keep an F1Tenth car from crashing, we show that this system is applicable to real-world software and hardware.

Using scheduler mode switching and our modified preemptable ROS 2 executor along with a reinforcement learning-based track centerline estimator, a path following controller, and a follow-the-gap controller, we find that scheduler mode switching can keep the car from crashing in a varying environment, maintaining high average quality of service. Without the scheduler mode switch, the system would always have to run with a lower reaction time by either removing some tasks entirely or reducing the system utilization.

**Organization.** First, in Sec. II, we use a simple system to demonstrate where a change in the system state or environment can result in a different event reaction time bound. In Sec. III, we introduce our formal modeling approach and components used in verification. We describe the task and scheduler parameters, as well as the controller, physics, and environmental models that describe the physical system. We describe the concept of event reaction time, and how it differs from latency. We show the steps used to find the scenarios in which the system can fail if it were to use a higher timing bound, and how to use the scheduler model to find a task configuration that allows the system to meet the lower timing bound. Then, we describe the end-to-end verification approach using UPPAAL [9] that verifies that switching scheduler modes ensures safe behavior. In Sec. IV, we give details on our implementation of the system on an F1Tenth car and ROS 2,

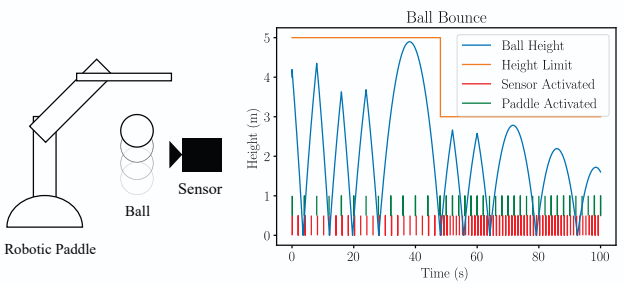



Fig. 2: *Example simulation of the ball-and-paddle system with a changing $\mathbb{H}$. The period of the sensor and paddle change to adapt to the decreased $\mathbb{H}$, allowing the system to keep the ball under the limit.*

showing that the system can be used on real world hardware and software. In Sec. V, we outline the models used to represent the system components, including scheduling properties of the Preemptive EDF scheduler and physical behavior of the F1Tenth car. Finally, in Sec. VI, we show the taskset properties as found by the scheduling model, and how the implementation performed on the physical system.

## II. Motivating Example

Before introducing the formal problem formulation, we present a motivating example related to an MC system, where system criticality levels shift in response to environmental feedback, illustrated by the classical bouncing ball problem.

Consider a simple system consisting of a ball, a ground surface, a robotic paddle, and a sensor. A height limit, $\mathbb{H}$, is defined. The ball is initially thrown downward from a height $h_0 < \mathbb{H}$ toward the ground surface. With no intervention, the ball may bounce off the ground and reach or exceed $\mathbb{H}$. Due to changing external conditions, $\mathbb{H}$ may vary over time. The sensor periodically reports the ball's position and velocity, along with the current value of $\mathbb{H}$. To keep the ball below $\mathbb{H}$, the robotic paddle can redirect the ball downward at any point during its travel. The sensor and paddle are controlled by a computer system and managed by a software scheduler. Both sensor data processing and arm actuation are periodic tasks. When executed, the sensor task reads the ball's position and velocity. If the ball is traveling upward, it predicts the maximum height the ball will reach on this bounce and the time remaining until it reaches the height limit. The paddle task reads the remaining time provided by the sensor task. If the ball is predicted to exceed $\mathbb{H}$ before the next scheduled paddle task, the paddle strikes the ball, redirecting it back toward the ground.

The system's ability to consistently keep the ball below the height limit $\mathbb{H}$ depends on the ball's initial velocity and position, the periods of the sensor and paddle tasks, and the range of $\mathbb{H}$. Additionally, if other tasks are running under the same scheduler, the sensor and paddle tasks may experience additional latency due to competition for processing time. The event reaction times of the sensor and paddle tasks depend on



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

the response times of each task, the scheduler state, and the presence of other tasks in the system.

For a range of starting positions and velocities, the system *safety property*—defined as the ability to keep the ball below the height limit—can be evaluated and verified for a specific configuration by modeling the system and performing safety queries. It is necessary to verify the system's safety rather than rely on a single trial run, given the range of starting configurations for the ball's position and velocity, the different combinations of possible latencies, and the uncertainty in sensor measurements. Formal methods, including symbolic and stochastic model checking as well as deductive verification techniques, can be used to ensure that the system operates safely within the specified range of conditions.

While the system's safety depends on the range of factors discussed above, it is particularly sensitive to the $\mathbb{H}$ parameter. If $\mathbb{H}$ is lowered, the required reaction time from sensing to paddle actuation is decreased. The paddle task depends on the height and time estimates generated by the sensor task, which are only produced if the sensor task detects the ball traveling upward. For even lower values of $\mathbb{H}$, the sensor task's period may be insufficient to capture the ball's upward movement before it reaches the limit. Therefore, the system's required reaction time is governed by the environmental input $\mathbb{H}$. To ensure that the sensor and paddle can always react in time, their reaction times can be adjusted by modifying scheduling parameters—such as increasing priorities or reducing periods—or by dropping non-critical tasks in the system.

While previous work has introduced systems capable of dynamically adjusting task scheduling properties at runtime in response to changes in execution times, we extend this capability to adapt to changing environmental conditions. This is where the MC framework becomes particularly valuable.

The MC framework allows implementers to assume optimistic execution times for some tasks under normal conditions. If a critical task exceeds its optimistic execution time, the system responds by increasing the deadlines of other tasks to ensure the critical task meets its deadline. Typically, mixed-criticality scheduling assigns two WCETs to each task: LO and HI. The LO value represents an optimistic estimation, while the HI value provides a more conservative upper bound. The scheduler operates under LO execution times during normal conditions. However, if a task exceeds its LO execution budget, the system switches to HI mode, where tasks are scheduled using their HI execution times. To maintain schedulability under HI execution times, the scheduler can drop or deprioritize tasks deemed less critical by the system designer.

In the context of our motivating example, the paddle task corresponds to a high-criticality task, as its timely execution is essential to maintain system safety by preventing the ball from exceeding the height limit. Extending these concepts to our intended system, the F1Tenth platform, similar criticality distinctions can be applied to tasks like obstacle detection and control. For example, navigating sharp turns may demand high-criticality scheduling, while operating on straight paths could utilize lower-criticality modes. By leveraging the MC framework, we aim to dynamically adjust task parameters to balance performance and safety in real-world, high-speed environments.

## III. Proposed Verification Framework

To support physics-informed mixed-criticality in a system, we propose a formal modeling framework to determine the necessary parameters for safe and efficient operation. These models enable us to identify critical physical states that require a lower reaction time and to select appropriate parameters for scheduler mode switching based on environmental conditions.

### A. System Model

Our system model, $\mathcal{M} := \{\mathcal{M}_W, \mathcal{M}_S, \mathcal{M}_C, \mathcal{M}_P\}$, consists of a workload model $\mathcal{M}_W$, a scheduler model $\mathcal{M}_S$, a controller model $\mathcal{M}_C$, and a physical model $\mathcal{M}_P$. The relationships among these components are illustrated in Fig. 3. To ensure safety, we verify that the model $\mathcal{M}$ operating in an environment $\mathbb{E}$ satisfies ($\models$) a property $P$, or $\boldsymbol{\mathcal{M}} \,\|\, \mathbb{E} \models \boldsymbol{P}$, where $\mathcal{M}$ and $\mathbb{E}$ are composed in parallel and $P$ is a system-level property.

In our implementation, $\mathcal{M}$ represents an F1Tenth racing car, $\mathbb{E}$ represents obstacles and other geometric features like the racetrack, which may be time-varying, and $P$ defines a safety property, such as collision avoidance.

**Workload Model ($\mathcal{M}_W$)** consists of a set of $n$ independent periodic tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ on a uniprocessor system. For the scheduler model $\mathcal{M}_S$ to accurately determine worst-case latency, $\Gamma$ should include all tasks in the system that could impact the timing of controller execution, even if they do not directly influence the controller's computed values. Each task is represented as $\tau_i = (C, T_{\text{LO}}, T_{\text{HI}}, D_{\text{LO}}, D_{\text{HI}}, \chi)$, where $C$ is the worst-case execution time (WCET), $T_{\text{LO}}$ and $T_{\text{HI}}$ are the periods in *low* and *high* modes, $D_{\text{LO}}$ and $D_{\text{HI}}$ are the deadlines in *low* and *high* modes, and $\chi$ denotes the task's criticality, The criticality flag $\chi$ indicates whether a task should be dropped in *high* mode.

**Scheduler Model ($\mathcal{M}_S$)** models the behavior of the scheduler used in the system to schedule the workloads in the underlying hardware platform. Scheduling policies could be directly implemented in operating systems (*e.g.*, RTOS, RTLinux) or using middleware such as ROS 2, and AUTOSAR [10] for better composability and modularity in complex autonomous systems. Scheduler model $\mathcal{M}_S$ precisely models the scheduling policies interacting with the workload model to safely compute the worst-case latencies for the workloads used in the system. It is essential to validate the functionality of the model before use to ensure that all necessary properties of the scheduler are correctly modeled in the scheduler model.

The scheduler is a preemptive EDF (earliest-deadline-first) scheduler, where (depending on the current system mode) tasks are scheduled by absolute deadlines governed by either $D_{\text{LO}}$ or $D_{\text{HI}}$. The scheduler supports changing the parameters of tasks at runtime. These changes, known as mode switches, are triggered via a signal sent by some task in the system, or from an external source. Mode switches are processed during



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

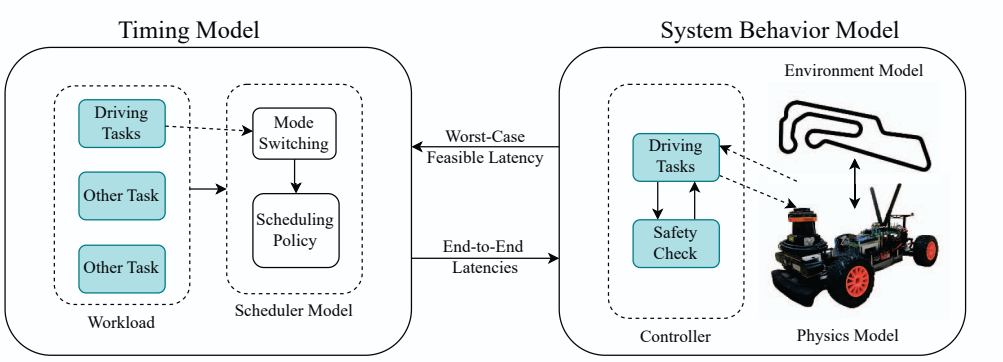



Fig. 3: *End-to-end verification framework. The timing model (on the right) uses the system workload, scheduler policy, and mode-switch decisions from the driving task to determine the end-to-end latency of the system tasks, both before and after a mode switch. The driving tasks in the controller model run according to latencies calculated by the scheduler model and read values from the physics model and environment. The controller tasks provide control inputs by writing to the physics model, moving it through the environment. The safety check, one of the driving tasks, reads the output of the centerline estimator, along with the vehicle and environment state, to determine whether a mode switch is necessary. The driving tasks are delayed based on the latency values determined by the timing model and the selected mode. The scheduler supports full preemption, and since the driving tasks share an absolute deadline, they can be represented as a single task.*

the earliest idle instances—times when no tasks are queued and running—after the mode switch signal. We impose this limitation to add flexibility to mode switches. If a mode switch decreased the absolute deadline of a queued or running job, the job may become unschedulable. Furthermore, stopping a currently running job if the task were dropped during a mode switch could cause the system to enter an invalid state, and complicate the design of the system tasks. Delaying mode switches to the next idle instance solves these two problems, at the expense of a potentially long delay between the mode switch request, and its completion. To aid design, the scheduler model calculates the longest possible delay.

If the scheduler is in *low* mode and receives the switch signal, the scheduler switches to its *high* mode. This changes the periods and deadlines of all tasks to reflect the current mode. If the system is switching from *low* to *high*, all tasks are scheduling using their $T_{\text{HI}}$ and $D_{\text{HI}}$ values. If the $\chi$ flag for a task is not set, the task is dropped. After some time, the scheduler can receive another signal to return to LO mode, which causes new tasks to be scheduled with their $T_{\text{LO}}$ and $D_{\text{LO}}$ values. Any tasks with their $\chi$ flag not set are added back to the system.

**Controller model ($\mathcal{M}_C$)** defines an algorithm that reads state values from the Environment model ($\mathbb{E}$), and outputs values that affect the Physics model ($\mathcal{M}_P$), with the goal of meeting the property $P$. The controller has a latency value, which is the time between its release time and the completion of its execution, and also a reaction time, which is the maximum it can take to react to an event in the environment. The controller observes the state of the environment at the beginning of its execution, and outputs a control signal at the end, which is applied to the physics model. The controller model does not simulate the scheduler, nor does it determine the latency value—instead, the latency is computed separately in $\mathcal{M}_S$, where one of the tasks in $\Gamma$ represents the controller model. During verification, the controller model uses the timing properties generated by the scheduler model. The controller model selects the appropriate parameters during mode switches, and also simulates the potential delay between the mode switch request and the mode switch application.

**Physics Model ($\mathcal{M}_P$)** is user-provided, and its functionality is validated independently. The physics model describes how the physical system moves through and interacts with the environment over time. The physics model may be defined by differential equations that describe the motion of objects. The physics model should expose parameters that can be controlled from the controller model $\mathcal{M}_C$.

**Environment Model ($\mathbb{E}$)** defines the scenario within which the physics model operates. The environment model exposes state variables that can be read by the controller model as input. The environment model may also abstract some of the sensing processes that would happen on a real physical system, such as localization or object detection, into simpler tasks. We assume the environment model is known *a priori*.

### *B. End-to-End Verification*

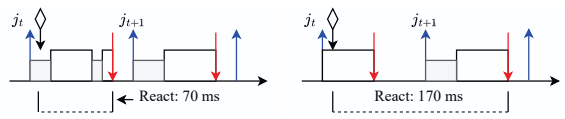
We first define a task response time, an event reaction time, and the system modes for mixed-criticality scheduling.

**Definition 1.** (Task Response Time) *The task response time, or latency, is the amount of time between a task being released (becomes allowed to run) and completing execution (producing a result and yielding to the scheduler).*

Assume an autonomous system with a time-driven driving task that polls a sensor value, performs some computation and outputs a new control value. If an event happens just before the task job begins, it will be recognized by the job



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.




(a) (b)

Fig. 4: **Event Reaction Time:** *The upward blue arrow means the release instant of a job, and the downward red arrow means the completion time instant of a job. Task $j$ has a period of $100ms$ and an execution time of $50ms$. Due to interference from other tasks, it can be blocked for up to $30ms$. If an event occurs after $j_t$ is released, but before $j_t$ begins execution and polls the sensors, the state change in the environment caused by the event will be captured by $j_t$, and a response will occur before the end of the period (Fig. a). If the event occurs after $j_t$ polls the sensors, then the next instance, $j_{t+1}$ will respond to the event (Fig. b).*

and handled by the end of the execution time. If the event occurs just after the job begins and the sensor is polled, it will not be recognized by this job - only the *next* job instance will react to it. Therefore, the maximum event reaction time in a system is the maximum amount of time between job $j_{t-1}$ begins execution and $j_t$ completes execution.

**Definition 2.** (Event Reaction Time) *The reaction time is the amount of time between some event occurring in the environment, and the system recognizing and responding to the event. In an autonomous driving system, this would be time between an obstacle entering the range of the sensors, and the system responding by decelerating or changing direction. The reaction time of a task is affected by the period, execution time, and blocking time. This is distinct from* response time *in that it reflects how quickly the system responds to physical events, not timing events. When we refer to a task's* reaction time*, we are referring to the event reaction time of that task.*

**Definition 3.** (System Modes) *The scheduler must expose configuration parameters that change the scheduling parameters of each task. A set of parameters is referred to as a mode. For a dual-critical system, the scheduler may have the option to run the driving task at a longer period (and therefore a lower frequency), allowing all tasks in the system to run. We can refer to this behavior as the system's* low *mode. In the system's* high *mode, the scheduler could run the driving task with a shorter period (a higher frequency) to react to environment events sooner. Since running the driver task more frequently increases the utilization of the system, the scheduler may increase the period of some less-important tasks, and even drop some unnecessary tasks. Even if the system can run all tasks in the* high *mode, dropping some tasks will still be beneficial as the response time (and therefore reaction time) of the driving task will decrease.*

System modes allow the scheduler to adapt task execution to changing environmental conditions, ensuring that the critical tasks can meet physically imposed reaction time requirements.

Using formal methods, we can determine the control system's worst-case response time, required reaction time, and the proper system mode for acceptable reaction time to operate safely. The design and verification steps are as follows:

- **Step 1:** Derive the models that represent the system components. The timing model (*i.e.*, workload and scheduling model) will represent the scheduler for the system, and determine its timing properties, including the worst-case reaction times of the control task. The environment model describes where the physical system will operate, and should expose values to be read as sensor inputs in the controller model. The controller model should describe the tasks required to manipulate the physical system to achieve the objective and also use the input signals to determine whether a mode switch is necessary. The physical system model should be derived from system dynamics [11].
- **Step 2:** Use the physics, controller, and environment models to find scenarios where the system can fail. It may not be possible for a given controller to succeed in *all* situations, so the environment models must be constrained to represent situations in which the physical system will be deployed.
- **Step 3:** Find a reaction time and latency that allow the system to perform safely in as many of the previous scenarios as possible. If there are scenarios where a lower reaction time and latency can not guarantee system performance, adjustments to the controller are required (A Simplex controller design could switch to a safer controller if the system is approaching these states). For the states that can be avoided with a lower reaction time and latency, determine a mode-switching criteria to detect these states.
- **Step 4:** Use the scheduling model to find appropriate values for $T$, $D$, and $\chi$ that provide the required reaction time for the driving task. In the *low* mode, all tasks in the system should run with a high enough period to support their required functionality, but in *high* mode, some tasks may have reduced frequencies (higher $T$ and $D$ values) or may not run at all. For each set of parameters, the scheduling model finds reaction times for the driving task during *low* and *high* modes, as well as the time required to switch between the modes.
- **Step 5:** Use the physics, controller, and environment models, with the timing values found in the scheduler model, to determine whether applying the mode switch ensures system safety.

## IV. Implementation

This section presents the implementation details outlining the system specifications, centerline generation, controller implementation, and scheduler implementation[1].

[1]Our implementation source files are available at https://github.com/RTIS-Lab/ROS-Phys-MC



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

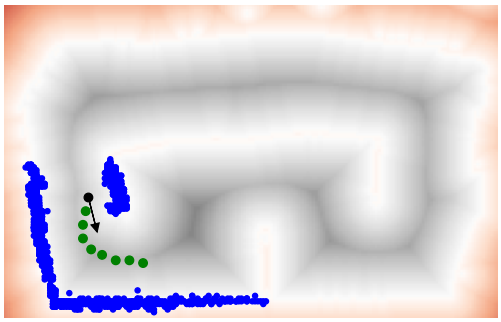


Fig. 5: *Illustration of centerline model training process. The back point is the car's position, and the arrow represents the car's direction. The blue points represent the LiDAR scan, which provides a partial view of the track borders. The green points are the points generated by the centerline model. The background color represents the distance transform of the map, which is used to calculate the training rewards. Each output point is scored by the value in the distance transform at that point. Areas inside the track, marked in gray, have negative values, and points outside the track, marked in red, have positive values. The model is scored by the sum of the point scores. We add a penalty for extreme curvatures. By training the model to maximize reward, the model learns to place points on the centerline that the car can follow.*

### A. System Specifications

Our system is an F1Tenth [5] car running ROS 2 on an Nvidia Jetson Xavier AGX. The car is equipped with a LiDAR sensor to detect obstacles and track borders. Our driving code, the LiDAR interface, and motor control drivers all run on ROS 2. We restrict the ROS 2 executor and the callback threads to run on a single CPU core, and fix the clock frequency to 2265 MHz. We use the GPU for centerline model inference.

### B. Centerline Generation

We trained a neural network model using reinforcement learning to estimate the position of the track centerline relative to the car. We split the LiDAR scan into 30 segments, take the minimum and mean distance of each segment, and use these as the model inputs. The network has two fully-connected layers of 256 neurons each. The output is 8 evenly spaced points representing the two meters of centerline ahead of the car. We used Soft Actor-Critic [12] to train the model on a mix of scans from our physical track, and some tracks traced from real-world racetracks [13]. The training process places the vehicle at random points and orientations on the track. The model proposes 8 yaw values that control the relative direction from each point to the next. We apply a distance transform on the map, where points on the map farther from the nearest obstacle have higher values. The model is rewarded with the values of the distance transform at each generated point. This encourages the model to find points that track the centerline of the track. The training environments for the centerline estimator did not include obstacles, but when faced with convex obstacles, usually generates paths that avoid them. We show an example of a LiDAR scan, generated centerline, and the distance transform used for centerline model training in Fig. 5.

### C. Controller Implementation

We use two controllers to drive the car: a *center-line following controller* that attempts to follow an estimate of the track's centerline, and a *follow-the-gap controller* that drives towards the farthest visible and reachable space.

**Center-line Following Controller.** The centerline-following controller uses the predicted centerline and treats it as a pure pursuit driving line [14]. Using the car's heading and speed, it picks a point on the centerline and sets the steering angle to a value that will drive the car in that direction.

**Follow-the-Gap Controller.** The follow-the-gap controller uses the LiDAR scan to find the farthest reachable space that can be reached by driving in a straight line. A point is considered reachable if no visible obstacle is within some $n$ units. This controller does not guarantee progress along the track, but assumes convex obstacles and a uniform track width to prevent crashing into static obstacles.

**Safety Checking.** Using the LiDAR scan, we can test whether driving on the generated centerline will cause a collision—if the generated centerline is within some distance of any point observed by the LiDAR, then the centerline is unsafe to use. The opposite is not true—a centerline that does not intersect with observed LiDAR points is not guaranteed to be a safe path. If the car gets close enough to an obstacle that it is in danger of crashing, the controller will issue an emergency stop command.

### D. Scheduler Implementation: Making ROS 2 Preemptive EDF

We use fully preemptive EDF scheduling, a widely adopted approach for real-time systems. Sensing and actuation tasks are implemented as ROS 2 callbacks, which are selected and run by an executor. In ROS 2, callbacks can originate from various sources: Timer callbacks are triggered by a periodic timer, Subscriber callbacks activate in response to incoming messages, and so on. Callbacks of any type can send messages over topics, which can be subscribed to by Subscribers. The default ROS 2 executor schedules tasks non-preemptively, and callback priorities are set by a combination of registration order and callback type [15]. In order to apply mixed-criticality scheduling to this system, we create a modified ROS 2 executor that runs callbacks in separate threads. Each callback has a unique thread, and each thread uses Linux's `SCHED_DEADLINE` scheduling class. Rather than executing callbacks directly, the executor passes a reference of the source event (eg. a timer or subscriber) to the respective thread.

To simplify the implementation, we require that Callbacks must not be in reentrant callback groups. Reentrant callback groups allow multiple instances of the same callback to be simultaneously. Since each callback has a dedicated thread, only one instance of a callback can run at a time. We also



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

**Algorithm 1:** Preemptive EDF Executor Callback Dispatch

```
1  while rclcpp::ok() do
2      wait_for_work();
3      sched_mutex.lock();
4      next_executable ← get_next_executable();
5      if next_executable == nullptr then
6          if mode_switch_flag changed and
             !any_running(thread) then
7              modeswitch();
8          end
9          continue;
10     end
11     if !next_executable.group.can_take() then
12         continue;
13     end
14     this_thread ← threads[next_executable];
15     if this_thread = nullptr then
           /* first time this callback has been
              run                                */
16         this_thread ← new Callback-
             Thread(getPropertiesForCB(next_executable));
17         sched_setattr(this_thread, SCHED_DEADLINE,
             next_executable.sched_params[mode]);
18     end
19     if this_thread.running then
           /* one of the constraints has been
              broken - all callbacks should be
              non-reentrant                      */
20         warn();
21         return;
22     end
23     this_thread.executable ← next_executable;
24     this_thread.running ← true;
       /* thread will handle retrieving data
          */
25     this_thread.semaphore.release();
26     sched_mutex.unlock();
27 end
```

**Algorithm 2:** Callback Threads

```
1  while rclcpp::ok() do
2      sem.acquire();
3      if !executable.group.can_take() then
4          continue;
5      end
6      execute_executable(executable);
7      executable.group.reset();
8      running ← false;
9  end
```

require that only one publisher can publish to a topic in a single period.

To run callback instances in their own threads, we create a mapping of callbacks to threads. Initially, the mapping is empty. Whenever a callback becomes eligible, the executor checks whether a thread has been created for the callback. If not, it creates a new thread, sets its scheduling class to `SCHED_DEADLINE`, sets the period, runtime, and deadline parameters, and passes the callback information to the thread. The executor adds the thread to the callback→thread mapping and moves on to the next eligible callback. If a thread already exists, and the thread is running another job from that callback,

**Algorithm 3:** Mode Switch

```
   /* assumes nothing is running             */
1  for thread in threads do
2      params = all_params[thread.callback][mode];
3      sched_setattr(thread, SCHED_DEADLINE, params);
       /* also set the timer period          */
4      if thread.callback is TIMER then
5          thread.callback.period ← params.period;
6      end
7  end
```

then some restrictions were violated (the callback missed a deadline, or multiple callbacks were published to the same topic). In this case, the executor will leave the callback in the queue. If the thread has been completed, then the executor passes the new callback data to the thread, which will run when it is selected by the kernel scheduler.

In ROS 2, timer and subscriber callbacks can be combined to create time-driven processing chains. In this model, each processing chain is driven by a single-timer callback, and all callbacks in the chain share an absolute deadline. When running under a normal ROS 2 executor, each callback in the chain serves as a preemption point. In our modified executor, callbacks can be preempted at any point, so to simplify the analysis, each callback chain is abstracted into a single task.

## V. Modeling and Verification

We use UPPAAL models to verify the scheduling policy, mode switching, and physical system behavior.

### A. Modeling

Our scheduling model uses an EDF scheduler with full preemption, adapted from the `SchedulingFramework` example provided by UPPAAL. We incorporate the possibility of a mode switch, which can be requested at any time. The mode switch can add or remove tasks and adjust task periods or deadlines. It is applied at the next idle instant of the scheduler. We introduce a verification query to determine the maximum time between any two idle instances. If this time is excessively long or unbounded, it indicates that waiting for an idle instant to perform a mode switch could delay the switch beyond acceptable limits.

The scheduler employs two templates, outlined in Model 1 and Model 2.

The scheduler model represents the behavior of an EDF scheduler. It remains in the idle state until some task emits a ready synchronization, upon which it adds the task to the queue. insertTask checks the absolute deadline of the task, and places it the appropriate queue position. Tasks with the same absolute deadline are sorted non-deterministically - either task could be run before the other. Once a task is placed in the queue, the scheduler switches to its InUse state. Any newly released tasks are placed in the queue. When the running task is completed, it is removed from the queue. Once the queue is depleted (empty()=true), the scheduler moves back to its Idle state, and sets the time_between_idle clock to 0. By allowing time_between_idle to tick at the InUse state, the supremum



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

**Model 1:** EDF Executor

```
1  while true do
2      State Idle Initial
3          time_between_idle′ = 0;
           /* emit when a task becomes
              ready                          */
4          Synchronization ready?
5              insertTask(readyTask);
6              go to InUse;
       /* Some task is running               */
7      State InUse
           /* Task at head of queue
              completed                      */
8          Synchronization finished
               /* Removes the head           */
9              removeTask();
10             if empty() then
11                 time_between_idle := 0;
12                 go to Idle;
               /* Otherwise, the queue isn't
                  empty. The task at the head
                  is now running             */
13         Synchronization ready?
14             insertTask(readyTask);
15         time_between_idle′ = 1;
16 end
```

**Model 2:** Task

```
1  State PeriodDone Initial
2      x′ = 0 && time ≤ period();
3      if time ≥ period() then
4          x = 0;
5          time = 0;
6          readyTask = id;
7          emit ready!;
8          go to PreReady;
9      end
10 State PreReady
11     x′ = isRunning() && x ≤ 1;
12     if x = 1 then
13         event_reaction_time := reaction_time;
14         reaction_time = 1;
15         go to Ready;
16     end
17     if time > deadline() then
18         go to deadline_error;
19     end
20 State Ready
21     x′ = isRunning() && x ≤ WCET();
22     if x ≥ WCET() then
23         emit finished!;
24         go to PeriodDone;
25     end
26     if time > deadline() then
27         go to deadline_error;
28     end
```

of time_between_idle at InUse represents the maximum time between scheduler idle instances.

The task model represents tasks running under the EDF scheduler. Tasks begin in the PeriodDone state, where the time clock counts towards period(). Once a period has passed, the task emits ready!, causing the scheduler to add it to the queue, and move to the PreReady state. The x clock increments only when the task is running, which is when the task is at the front of the queue. If some other task takes its place, x will not increment. Once the task has run for at least one-time unit (x=1), it copies reaction_time to event_reaction_time, resets reaction_time, and moves to Ready. In both the PreReady and Ready states, if the time ever exceeds the deadline, then the task enters the error state. x evolves in the same way as it did in PreReady. Once the task has been run for its execution time, it removes itself from the queue by emitting finished! and moves to the PeriodDone state.

To understand the roles of reaction_time and event_reaction_time, remember that the worst-case response time of a task is the maximum amount of time between a job beginning execution (where sensors are polled), and the completion (where control outputs are sent) of the *next* job. The PreReady state is designed to indicate whether the task has been run at all by the scheduler since the task has been released. Once the task has at least begun execution, reaction_time, which was set at the last time the task began, is copied to event_reaction_time, and reset back to 1 (since the task has run for 1 time unit, the smallest time length in this model). By the time the task exits the Ready state, event_reaction_time will include all the time passed since the *last* job began and the time *this* job completed. By finding the supremum of event_reaction_time at Ready, we can find the worst-case reaction time of a task.

A mode switch signal comes from the driving task. To prevent a mode switch from placing the system in an invalid state, the mode switch flag is only checked when all callback threads are idle - which means that no callbacks are running and the scheduler is idle. While can cause a mode switch to happen sometime after the switch was requested, this allows greater flexibility in what actions can be taken during a mode switch. If a deadline were shortened while a callback is queued or running, the new absolute deadline could be in the past, causing a deadline miss. Additionally, if a callback were dropped, the executor would not have to stop an in-progress thread. For tasksets with a very long time between scheduler idle periods, or tasksets where the scheduler is never idle, the scheduler would have to be modified to apply a mode switch while some tasks are running, complicating the implementation and analysis.

The vehicle model is implemented with 5 states such as $x$ position, $y$ position, yaw, velocity, and steering angle, each governed by an ODE [16]. The $x$ and $y$ states evolve using the velocity and steering angle: $\dot{s}_x = v\cos(\Psi)$, $\dot{s}_y = v\sin(\Psi)$, and the yaw evolves using the steering angle: $\dot{\Psi} = \frac{v}{l_{wb}}\tan(\delta)$. The longitudinal velocity $v$ can be changed with the throttle



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

input, and the steering angle $\delta$ moves to match the input steering angle. LiDAR inputs are generated using the scan simulator from [5]. We use the FFI[2] feature in UPPAAL to call our prediction and driving code, which uses the LiDAR scan to predict the centerline, make a safety evaluation, and make a control decision. The safety evaluation and control inputs are passed back to UPPAAL, which updates the physics model. Delays built into the controller model use values computed by the full scheduler to simulate the sample times and output times of the driving task. Since the values computed by the full scheduler include the effects of other tasks in the system on the driver, the controller model only needs to simulate the driver. When the safety evaluation determines the need for a mode switch, a flag is set. After some delay (up to the idle time gap found by the scheduler model), the appropriate values (either the *high* or *low* set) are copied to the delays representing blocking and runtime.

### B. Verification

Using these models, the properties of the taskset under the EDF scheduler can be determined using UPPAAL symbolic queries:

- `A[] forall (`$i$` : tasks) not Task(`$i$`).Error`: This determines whether all deadlines will be met. Tasks go to the Error state whenever the absolute deadline has passed and the current job has not yet finished.
- `supTask(`$i$`).Ready: time[`$i$`]`: For a specific task $i$, this finds the worst-case latency by finding the supremum of a time[$i$] at Ready.
- `supTask(`$i$`).Ready: event_reaction_time[`$i$`]` : For a specific task $i$, this determines the worst-case reaction time by finding the supremum of event_reaction_time at Ready. The event_reaction_time value records the time between the beginning of a job $j_t$ and the completion of the next job $j_{t+1}$.
- `supTask(`$i$`).Ready: reaction_time[`$i$`]`: Determines the maximum possible time between the first instance of job $j$ executing (when sensors are polled) to the last instance of $j$ executing (when control outputs are sent). This value represents data freshness.
- `sup: time_between_idle`: Finds the maximum time that can pass between times when the scheduler is idle. Mode switches are queued until an idle instant occurs.

To find the probability of a crash in the system, we use the following query: `Pr[`$< T; N$`] (◇ crashed)`. $T$ is the maximum time for a single episode, and $N$ is an optional value to limit the number of episodes tested by UPPAAL.

## VI. Evaluation

We use the taskset shown in Table I. The relative deadline is always equal to the period for the respective mode. Using the scheduler and task model, we find the latencies and reaction times for each task in both *low* and *high* modes.

[2]Foreign Function Interface - this allows UPPAAL to call code written in other languages and receive a response. This enables us to use the same code both on the real system and the verification process.

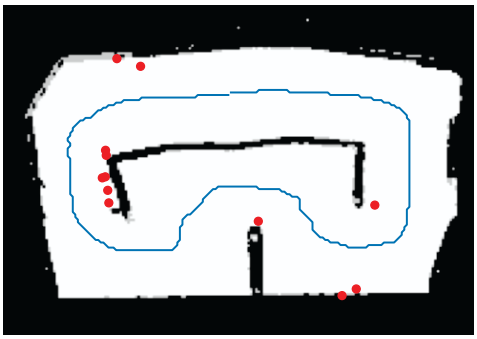


Fig. 6: *Red dots are crash locations on one map when running in* low *mode, without any mode switches. These are locations where using different scheduler mode to reduce reaction time may allow the vehicle to safely navigate the track.*

| Task Name | $C$ | $T_{\text{LO}}$ | $T_{\text{HI}}$ | $\chi$ | LO Resp | LO React | HI Resp | HI React |
|---|---|---|---|---|---|---|---|---|
| Driver | 15 | 100 | 25 | 1 | 94 | 170 | 16 | 41 |
| Health | 1 | 25 | 25 | 1 | 19 | 44 | 16 | 41 |
| Dummy0 | 21 | 40 | 80 | 1 | 34 | 74 | 69 | 149 |
| Dummy1 | 8 | 30 | | 0 | 24 | 52 | | |

TABLE I: LO Resp *and* HI Resp *are the task* response times *in* low *and* high *modes.* LO React *and* HI React *are the* low *and* high *mode* reaction-times *of each task. The Driving task is responsible for polling the sensors, computing an action, checking the safety of the current state, and writing a control action. Health is responsible for polling the state of the joystick (used to start/stop the car), as well as checking whether the Driving task has output a control signal in the past 200 ms. Dummy0 and Dummy1 add additional interference to the Driving task.*

To determine the *high* and $\chi$ parameters for this system, we used the scheduler and task models to determine the latency and reaction time parameters for the driving task, and applied these to the controller model. Using the controller, physics, and environment models, we found that the controller needed a 45ms reaction time to safely navigate a track.

To achieve the required reaction time, we shortened[3] the period of the driving task to 25 ms, which increases the system utilization to 1.432. To make the system schedulable again, we drop the Dummy1 task and increase the period of Dummy0 to 80. With these changes, utilization is 0.903, making the system schedulable, and the worst-case reaction time of the Driver task is 41 ms, satisfying the 45 ms reaction time requirement.

### A. System Verification Results

Due to the large state space of the system, we use Statistical Model Checking in UPPAAL. We set the car to start in random positions and orientations on the track and use the query `Pr[`$\geq 20$`] (◇ crashed)` to find the probability of a crash

[3]The LiDAR sensor cannot be polled faster than 40 Hz, so shortening the driving task period to any less than 25 ms would further increase the utilization of the system for no extra benefit.



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

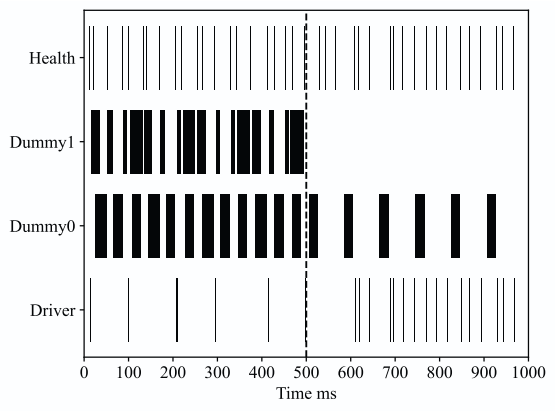



Fig. 7: *Timeline of task execution before and after a mode switch. The mode switch, sent from the driver task, occurred 2 ms before the idle instance. As expected, the Dummy1 task was dropped, and the period of the Dummy0 task was reduced. After some time, the driver's period decreased to the set 25 ms. Due to limitations in how ROS 2 handles changes in Timer periods, it takes one period from the previous mode for the new period to apply.*

happening in a single run. We set the episode length limit to 20 seconds - enough to complete two laps. With mode switching enabled, UPPAAL reports the chance of a crash having to be in the interval of $[0.0024, 0.03397]$ with a confidence interval of 95%. With mode switching disabled, the window becomes $[0.4771, 0.6780]$. This shows that performing a scheduler mode switch that reduces the reaction time of the critical task drastically reduces the chances of a crash.

### B. Physical System Results

We ran the taskset with our EDF ROS 2 executor on the F1Tenth platform on a straight track with barriers placed in the sides and middle that make the track narrower and require the car to turn in order to continue safely. We place the car at the end of the track and allow it to drive towards the barriers. We tested the system with mode switching enabled and disabled. When driving in empty sections of the hallway, the car stays in *low* mode, since the centerline predictor can easily track the center of the hallway, and provides a relatively stationary target for the line-following driver. Even with a straightforward path, in *low* mode, the reaction time of the Driver task is high enough that the car oscillates around the centerline. The empty part of the track is wide enough that with the oscillations, it still has room to recover. As the car approaches the narrow section of the track, the predicted centerline is closer to the observed track borders. If mode switching is enabled, once the predicted centerline gets close enough to the borders (0.5 m), the safety checker sets a flag. This flag is received by the scheduler, which will perform a mode switch to *high* at the next instant. Switching to *high* causes the scheduler to drop Dummy1, increases the period of Dummy0, and shorten the period of Driver. These changes significantly reduce the reaction time of the Driver task, allowing the car to accurately track the centerline without overshooting it. The system remains in the *high* mode until the car and centerline are both sufficiently far from any visible obstacle.

If mode switching is disabled, the car continues oscillating around the centerline, putting it in danger of collisions.

Oftentimes, the centerline predictor is not able to output a path that follows the centerline or even avoids obstacles. In this case, changing the timing parameters is not enough to prevent a crash. If the safety checker finds that the predicted centerline intersects an obstacle visible to the LiDAR sensor or the car itself is too close to an obstacle (0.5 m), the switch reduces the throttle and switches to the 'safe' gap-following controller until the predicted centerline and car state is safe. If the car gets closer still to an obstacle (0.4 m), the safety checker sets the throttle to 0, which brings the car to a stop. Both driver algorithms and the emergency stop action are available to the safety checker whether or not mode switching is enabled and run as part of the Driver task. Like the line-following driver, the gap-following driver and emergency stop are also sensitive to latency and reaction time.

We ran the car through the track 20 times with mode switching enabled and 20 times without. We consider the car crashed if it collided with a barrier or track border. If the safety checker stops the car, and the car comes to a stop without colliding with anything, we consider this an emergency stop, but not a collision. When the mode switching was enabled, the car completed the track 17 times. 3 runs ended with a safe emergency stop. No crashes occurred. Without mode switching, (where the system stays in *low* mode), the car crashed 4 times, and 3 runs ended with a safe emergency stop. The car completed the track in the other 13 runs.

We collected metrics about the tasks' runtime and reaction times during the *high* and *low* modes, as well as the time taken between a mode switch request, and a mode switch occurs. We calculate the reaction times of the tasks on the physical system in the same way as in the model - the difference between the beginning of a job and the completion of the next job.

While the real workload closely mirrors the behavior predicted by the scheduler model, there were some differences due to the implementation. Some tasks ran for slightly longer than predicted, possibly due to scheduler, kernel, or preemption overheads. We used libtorch for inference in the driving model, which, on rare occasions, ran longer than the 15 ms bound we had set. There was a delay in applying new task periods during a mode switch - tasks may wait for one period before the new period is applied.

In *low* mode, the Driver task had a maximum reaction time of 131 ms and a mean of 27 ms. With the exception of some outliers, the Driver had reaction times under the predicted *high* mode worst-case reaction time of 44 ms—in our experiments, the mean reaction time was 27 ms with a standard deviation of 10 ms. Dummy0 had a mean reaction time of 64 ms in *low* mode, and 109 ms in *high* mode. Dummy1 had a mean reaction time of 48 ms in *low* mode, and did not run at all in *high* mode. The HealthCheck task had a mean reaction time of 27 ms in *low* mode and 25 ms in *high* mode.



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

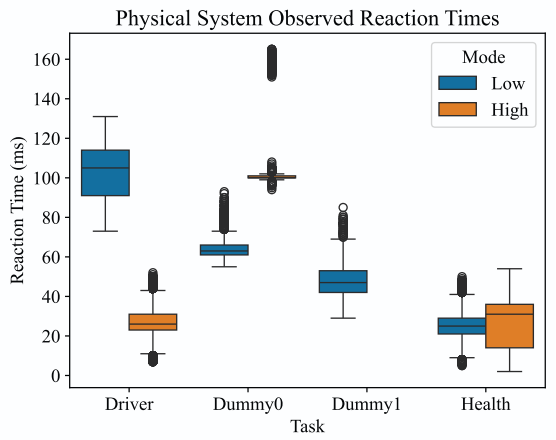



Fig. 8: *Observed reaction times from the physical system. The reaction times of the driver times match the predicted reaction times from the scheduling model. The other tasks mostly match the predictions, but there exist some outliers that exceed the expected reaction times, likely due to unaccounted overhead such as preemption cost, mode-switch overhead, underestimation of WCET, etc.*

Most tasks had outliers in their reaction times that were above the expected worst-case reaction times found in the scheduling model. We believe these came from unaccounted-for overheads from the scheduler and logging system.

## VII. Discussion

Our implementation and analysis assume a single-core execution environment with one sensor, where sensor processing and actuation outputs are managed by a single chain. While is is directly applicable for simple systems, real world systems have many task and task chains running in parallel on multiple CPU cores, and have multiple sensors, possibly polled at different rates. In this section, we discuss how the proposed framework can be extended to multi-core systems with multiple sensors and processing chains.

**Multiple Cores.** For simplicity, we use a single-core system in our case study example, but the models and ROS 2 implementation can be expanded to support multiple cores. To further reduce the reaction times of the case study tasks, we could place some task chains on different CPU cores, reducing interference between tasks and reducing reaction times of task chains.

To allow the use of multiple CPU cores for task execution, the executor and models have to be adjusted to support multi-core scheduling.

For EDF with a shared queue, the ROS 2 executor implementation could be modified to resemble the default multi-threaded executor, where the callback selection process happens in each executor thread, and the queue is protected by a mutex lock. Since ROS 2 natively supports triggering callbacks between executors, partitioned EDF could be done by creating multiple executors, and attaching tasks to the appropriate executor.

With execution on multiple cores, there may be a long time between instances where all cores are idle, if it exists at all. We may have to support performing a mode switch even when a task is currently executing—we leave this to future work.

The current UPPAAL model assumes that the task at the head of the queue is currently executing. To support a shared queue with multiple EDF executor threads, the executor model has to be adjusted to remove their selected tasks from the queue or otherwise mark them as currently executing so that multiple executor threads do not erroneously attempt to run the same task. For partitioned EDF, each EDF Executor model would pull tasks from a different queue. Supporting parallel task execution could significantly increase the search space during model checking—we leave model checking time analysis to future work.

**Multiple Sensors.** Multiple sensors can be polled in various arrangements: within the same task, by different tasks within the same chain, or independently across multiple chains. The most trivial case is multiple sensors polled together on the same task - each sensor is read at (almost) the exact same time, and processed together. In this scenario, the implementation and analysis remain the same. If sensors are polled and reacted to at different times but within the same chain, then the chain may have multiple different event reaction times - one for each sensor input $\leftrightarrow$ action output combination. In this case, the model has to be adjusted to record and report each possibility.

**Multiple Chains.** The overall technique could be applied to systems that are driven by multiple task chains, or where multiple task chains have dynamic reaction time requirements. The steps to determining the required event reaction time for each chain and how the scheduler modes are managed (which sensor inputs can trigger a mode switch, or how each chain is affected by mode switches, or even the possibility of a hierarchy of multiple modes) is dependent on the target system and its architecture.

For example, consider the dummy chains in the case study example. If each chain is polling and reacting to different sensors, then it may be necessary to ensure that each chain meets its required reaction times. To do so, multiple mode switches may be needed - whenever a task chain detects that it needs a lower reaction time according to its incoming sensor data, it may instruct the scheduler to reduce the period or priority of other task chains to meet the requirement. This introduces new complexities, such as how to handle conflicting requirements, or scenarios where one chain determines that the response time requirement of another chain has changed.

## VIII. Related Work

*Dual Controllers.* Our proposed framework can be compared with the Simplex architecture [3], [4]. In the Simplex architecture, an unverified High-Performance Controller (HPC) is paired with a safe High-Assurance Controller (HAC), and includes a verified safe Monitoring and Decision Logic (MDL). When the HPC's control action risks leading the system to an *unrecoverable state*—where the HPC can no longer guarantee safety—the MDL automatically switches to



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

the HAC to maintain safety. The MDL may be verified or validated through runtime reachability analysis or statistical simulations. [17] used formal methods to identify unsafe state space and showed that switching to a safe safe controller allows their system to recover. [18] is a working progress version of our framework.

*Mixed-Criticality Systems.* After Vestal's seminal work on Mixed-criticality (MC) systems [2], MC systems have been heavily studied over the years. A rigorous survey on MC systems can be found in [19]. The original MC model was studied in various settings, including fixed-priority scheduling (e.g., [2]) and dynamic-priority scheduling (e.g., [20], [21]). There are many variations proposed over the years to handle additional resource demand for high critical tasks in critical systems mode (e.g., [22], [23], [24]). [25] generalized the MC systems for both resource supply and execution time uncertainties. However, most of these works are limited to temporal overrun-based mode switch scenarios. Unlike the existing works, our work presents a model that uses variations in the environmental state to initiate the mode switch.

*Formal Methods for CPS and F1Tenth Cars.* In a recent survey of autonomous systems and the challenges they pose, Wing [26] asks how we can address scenarios that have life-critical consequences for people and society, and suggests that we require "new formal methods techniques" to do so. With respect to autonomous ground vehicles in particular, Kopylov *et al*. [27] verify a "safety net" for a waypoint navigation controller using ModelPlex [28] to synthesize a monitor using theorem proving. Lin *et al*. [29] extend the work by combining theorem proving and reachability analysis with Flow* [30] for synthesizing switching monitors. In the context of F1Tenth vehicles, Ivanov *et al*. [31] verify the safety of a neural network controller using their Verisig tool [32]. Vehicles operate at constant throttle in a structured environment, and the effect of missing LiDAR rays due to reflections is evaluated. A systematic literature review on verification and validation for safe autonomous cars is given by Rajabli *et al*. [33]. Soteria [34] presents a verification framework combining both the timing and physical model of a system for a specific operating environment.

*ROS 2 and Schedulability Analysis.* ROS 2 recently received significant attention from the real-time systems community after the pioneer work by Casini *et al*. [15]. Many works (few to mention [35], [36], [37]) subsequently improved the proposed worst-case latency bound of ROS 2 workloads and also proposed modified executor schedulers [38], [39], [40]. However, to our knowledge, there is no exact analysis for finding worst-case latency ROS 2 workloads. Formal methods have the potential to find exact worst-case timing bound (regardless of scalability issues) and were used in earlier works for (exact) schedulability analysis for the standard workload and resource models, *e.g*., exact worst-case response time computing for DAG tasks [41], exact scheduling test for non-preemptive self-suspending tasks [42], etc.

## IX. Conclusions and Future Work

We present a mixed-criticality model that uses environmental feedback to change the scheduling parameters of tasks in the system with the purpose of meeting the physical and environmentally required event reaction times. We use formal modeling as a process to determine the required reaction times and validate that the task parameters and system scheduler can meet these reaction times. We use system and controller models to verify the complete system behavior, showing that using environmental feedback in scheduling decisions can keep the system safe. Finally, we show an implementation of this system on an F1Tenth car and ROS 2, demonstrating the feasibility of physics-informed mixed-criticality scheduling on a realistic system. In future work, we aim to develop rigorous correctness proofs analytically for the verification framework and mixed-criticality scheduling policies to ensure both safety and efficiency in real-world applications.

## Acknowledgments

We are grateful to the anonymous reviewers for their valuable feedback and the anonymous shepherd for guiding us in improving the paper. This work was supported in part by the National Science Foundation under Grants CCF 2124205, CMMI 2246672, and CNS 2045539.

## References

[1] S. Chakraborty, M. A. Al Faruque, W. Chang, D. Goswami, M. Wolf, and Q. Zhu, "Automotive cyber–physical systems: A tutorial introduction," *IEEE Design & Test*, vol. 33, no. 4, pp. 92–108, 2016.

[2] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE international real-time systems symposium (RTSS 2007)*. IEEE, 2007, pp. 239–243.

[3] D. Seto, B. Krogh, L. Sha, and A. Chutinan, "The Simplex Architecture for Safe Online Control System Upgrades," in *Proceedings of the 1998 American Control Conference. ACC (IEEE Cat. No.98CH36207)*, 1998, pp. 3504–3508 vol.6.

[4] Lui Sha, "Using Simplicity to Control Complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, jul 2001.

[5] M. O'Kelly, H. Zheng, D. Karthik, and R. Mangharam, "F1tenth: An open-source evaluation environment for continuous control and reinforcement learning," *Proceedings of Machine Learning Research*, vol. 123, 2020.

[6] A. Heilmeier, A. Wischnewski, L. Hermansdorfer, J. Betz, M. Lienkamp, and B. Lohmann, "Minimum curvature trajectory planning and control for an autonomous race car," *Vehicle System Dynamics*, vol. 58, pp. 1–31, 06 2019.

[7] Z. Zang, H. Zheng, J. Betz, and R. Mangharam, "Local_inn: Implicit map representation and localization with invertible neural networks," 2022. [Online]. Available: https://arxiv.org/abs/2209.11925

[8] T. Nagy, A. Amine, T. X. Nghiem, U. Rosolia, Z. Zang, and R. Mangharam, "Ensemble gaussian processes for adaptive autonomous driving on multi-friction surfaces," 2023. [Online]. Available: https://arxiv.org/abs/2303.13694

[9] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen, "Uppaal SMC tutorial," *International Journal on Software Tools for Technology Transfer*, vol. 17, no. 4, pp. 397–415, 2015.

[10] M. Zhang, Y. Teng, H. Kong, J. Baugh, Y. Su, J. Mi, and B. Du, "Automatic modelling and verification of Autosar architectures," *Journal of Systems and Software*, vol. 201, p. 111675, 2023.

[11] R. Banach and J. Baugh, "Formalisation, abstraction and refinement of bond graphs," in *Graph Transformation*, M. Fernández and C. M. Poskitt, Eds., 2023, pp. 145–162.

[12] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018. [Online]. Available: https://arxiv.org/abs/1801.01290



Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.

[13] J. Betz, H. Zheng, A. Liniger, U. Rosolia, P. Karle, M. Behl, V. Krovi, and R. Mangharam, "Autonomous vehicles on the edge: A survey on autonomous vehicle racing," *IEEE Open J. Intell. Transp. Syst.*, vol. 3, pp. 458–488, 2022.

[14] R. C. Coulter *et al.*, *Implementation of the pure pursuit path tracking algorithm*. Carnegie Mellon University, The Robotics Institute, 1992.

[15] D. Casini, T. Blaß, I. Lütkebohle, and B. Brandenburg, "Response-time analysis of ROS 2 processing chains under reservation-based scheduling," in *31st Euromicro Conference on Real-Time Systems*. Schloss Dagstuhl, 2019, pp. 1–23.

[16] M. Althoff, M. Koschi, and S. Manzinger, "Commonroad: Composable benchmarks for motion planning on roads," in *2017 IEEE Intelligent Vehicles Symposium (IV)*. IEEE, 2017, pp. 719–726.

[17] D. Genin, E. Dietrich, Y. Kouskoulas, A. Schmidt, M. Kobilarov, K. Katyal, S. Sefati, S. Mishra, and I. Papusha, "A safety fallback controller for improved collision avoidance," in *2023 IEEE International Conference on Assured Autonomy (ICAA)*, 2023, pp. 129–136.

[18] K. Wilson, A. Al Arafat, J. Baugh, R. Yu, and Z. Guo, "Physics-aware mixed-criticality systems design via end-to-end verification of cps," in *2024 22nd ACM-IEEE International Symposium on Formal Methods and Models for System Design (MEMOCODE)*. IEEE, 2024, pp. 98–102.

[19] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 6, pp. 1–37, 2017.

[20] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *ECRTS*. IEEE, 2012, pp. 145–154.

[21] P. Ekberg and W. Yi, "Bounding and shaping the demand of mixed-criticality sporadic tasks," in *2012 24th Euromicro Conference on Real-Time Systems*. IEEE, 2012, pp. 135–144.

[22] S. Baruah and A. Burns, "Implementing mixed criticality systems in ada," in *International Conference on Reliable Software Technologies*. Springer, 2011, pp. 174–188.

[23] H. Su and D. Zhu, "An elastic mixed-criticality task model and its scheduling algorithm," in *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2013, pp. 147–152.

[24] Z. Guo, K. Yang, S. Vaidhun, S. Arefin, S. K. Das, and H. Xiong, "Uniprocessor mixed-criticality scheduling with graceful degradation by completion rate," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 373–383.

[25] A. Al Arafat, S. Vaidhun, L. Liu, K. Yang, and Z. Guo, "Compositional mixed-criticality systems with multiple executions and resource-budgets model," in *2023 IEEE 29th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2023, pp. 67–79.

[26] J. M. Wing, "Trustworthy AI," *Communications of the ACM*, vol. 64, no. 10, pp. 64–71, 2021.

[27] A. Kopylov, S. Mitsch, A. Nogin, and M. Warren, "Formally verified safety net for waypoint navigation neural network controllers," in *Formal Methods: 24th International Symposium, FM 2021, Virtual Event, November 20–26, 2021, Proceedings 24*. Springer, 2021, pp. 122–141.

[28] S. Mitsch and A. Platzer, "ModelPlex: Verified runtime validation of verified cyber-physical system models," *Formal Methods in System Design*, vol. 49, pp. 33–74, 2016.

[29] Q. Lin, S. Mitsch, A. Platzer, and J. M. Dolan, "Safe and resilient practical waypoint-following for autonomous vehicles," *IEEE Control Systems Letters*, vol. 6, pp. 1574–1579, 2021.

[30] X. Chen, E. Ábrahám, and S. Sankaranarayanan, "Flow*: An analyzer for non-linear hybrid systems," in *Computer Aided Verification: 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings 25*. Springer, 2013, pp. 258–263.

[31] R. Ivanov, T. J. Carpenter, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Case study: verifying the safety of an autonomous racing car with a neural network controller," in *Proceedings of the 23rd International Conference on Hybrid Systems: Computation and Control*, 2020, pp. 1–7.

[32] R. Ivanov, J. Weimer, R. Alur, G. J. Pappas, and I. Lee, "Verisig: verifying safety properties of hybrid systems with neural network controllers," in *Proceedings of the 22nd ACM International Conference on Hybrid Systems: Computation and Control*, 2019, pp. 169–178.

[33] N. Rajabli, F. Flammini, R. Nardone, and V. Vittorini, "Software verification and validation of safe autonomous cars: A systematic literature review," *IEEE Access*, vol. 9, pp. 4797–4819, 2020.

[34] K. Wilson, A. A. Arafat, J. Baugh, R. Yu, X. Liu, and Z. Guo, "Soteria: A formal digital-twin-enabled framework for safety-assurance of latency-aware cyber-physical systems," in *Proceedings of the 28th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '25)*. Irvine, CA, USA: ACM, May 2025, p. 11.

[35] Y. Tang, Z. Feng, N. Guan, X. Jiang, M. Lv, Q. Deng, and W. Yi, "Response time analysis and priority assignment of processing chains on ROS2 executors," in *2020 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2020, pp. 231–243.

[36] T. Blaß, D. Casini, S. Bozhko, and B. B. Brandenburg, "A ros 2 response-time analysis exploiting starvation freedom and execution-time variance," in *2021 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2021, pp. 41–53.

[37] H. Teper, M. Günzel, N. Ueter, G. von der Brüggen, and J.-J. Chen, "End-to-end timing analysis in ros2," in *2022 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2022, pp. 53–65.

[38] H. Choi, Y. Xiang, and H. Kim, "Picas: New design of priority-driven chain-aware scheduling for ros2," in *2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2021, pp. 251–263.

[39] A. A. Arafat, S. Vaidhun, K. M. Wilson, J. Sun, and Z. Guo, "Response time analysis for dynamic priority scheduling in ros2," in *Proceedings of the 59th ACM/IEEE Design Automation Conference*, 2022, pp. 301–306.

[40] A. Al Arafat, K. Wilson, K. Yang, and Z. Guo, "Dynamic priority scheduling of multithreaded ros 2 executor with shared resources," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 43, no. 11, pp. 3732–3743, 2024.

[41] J. Sun, F. Li, N. Guan, W. Zhu, M. Xiang, Z. Guo, and W. Yi, "On computing exact wcrt for dag tasks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 2020, pp. 1–6.

[42] B. Yalcinkaya, M. Nasri, and B. B. Brandenburg, "An exact schedulability test for non-preemptive self-suspending real-time tasks," in *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2019, pp. 1228–1233.




Authorized licensed use limited to: N.C. State University Libraries - Acquisitions & Discovery S. Downloaded on May 26,2026 at 23:33:14 UTC from IEEE Xplore. Restrictions apply.