SOTERIA: A Formal Digital-Twin-Enabled Framework for Safety-Assurance of Latency-Aware Cyber-Physical Systems

Kurt Wilson*
kwilso24@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Abdullah Al Arafat*
aalaraf@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

John Baugh
jwb@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Ruozhou Yu
ryu5@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

Xue Liu
xueliu@cs.mcgill.ca
McGill University
Montréal, Canada

Zhishan Guo
zguo32@ncsu.edu
North Carolina State University
Raleigh, North Carolina, USA

ABSTRACT

Verifying the safety of latency-aware cyber-physical systems is both critical and challenging due to the interaction between continuous physical dynamics and discrete computational constraints. This paper introduces SOTERIA, a formal framework that integrates digital twins for ensuring safety in these systems. SOTERIA models both the physical dynamics and computational behavior, enabling integrated verification within a specific operating environment. This approach goes beyond conventional methods that either treat physical and computational aspects separately or rely on overly conservative worst-case analyses. By modeling hybrid dynamics alongside computational models and operating environments, SOTERIA verifies both functional and timing correctness. Leveraging established verification tools, SOTERIA determines whether end-to-end latencies meet formal specifications, bridging the gap between computational and physical requirements. We first introduce a simple example of a 1D adaptive cruise control system to illustrate its effectiveness. We then present findings from a case study using the F1Tenth racing car platform and the UPPAAL tool to demonstrate SOTERIA's effectiveness in realistic scenarios, enabling safety verification that was previously infeasible with conventional schedulability analyses. This work underscores the importance of an integrated verification approach for enhancing safety and reliability in autonomous systems.

CCS CONCEPTS

• **Computer Systems Organization** → Cyber-Physical Systems; • **Computing Methodologies** → Formal Modeling and Verification.

*Both authors contributed equally to this work.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

HSCC '25, May 6–9, 2025, Irvine, CA, USA
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1504-4/2025/05…$15.00
https://doi.org/10.1145/3716863.3718028

KEYWORDS

Hybrid Systems and Models, Temporal Verification, Formal Methods, Real-Time Cyber-Physical Systems

ACM Reference Format:
Kurt Wilson, Abdullah Al Arafat, John Baugh, Ruozhou Yu, Xue Liu, and Zhishan Guo. 2025. SOTERIA: A Formal Digital-Twin-Enabled Framework for Safety-Assurance of Latency-Aware Cyber-Physical Systems. In *28th ACM International Conference on Hybrid Systems: Computation and Control (HSCC '25), May 6–9, 2025, Irvine, CA, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3716863.3718028

1 INTRODUCTION

Modern hybrid computing systems, such as mobile robots, self-driving cars, delivery drones, etc., autonomously and seamlessly interact with the physical world to ensure proper behavior. As a result of these interactions, many autonomous systems are safety-critical in the sense that failure can lead to catastrophic effects on human lives and physical well-being. Hence, safety assurance through verifying the temporal and functional correctness of these systems before deployment is imperative.

Traditional safety analysis of such systems typically decouples the timing requirements from the functional correctness in the physical world as two independent steps [11]: (i) verify physical correctness via formal methods and derive all safe timing bounds from control theory, and then (ii) design and verify the computing systems for the timing bounds independently. While such approaches are common practice for verifying these complex systems, they tend to be overly pessimistic because all timing bounds are derived from the worst-case scenarios, including the system's operating environment. Moreover, verifying computing systems for the timing bounds using standard schedulability analysis techniques is challenging and often pessimistic due to the complexity of the middleware (*e.g.*, ROS 2 [1], AUTOSAR [2], etc.) used in these systems. As a consequence of such pessimism, existing approaches may lead to very strict timing bounds that are difficult to satisfy (given certain hardware or energy limitations) or that only very simple computation modules may be used in order to satisfy the strict bounds. Strictly satisfying these timing bounds (such as performing strict but pessimistic schedulability analysis) may not always be necessary for ensuring safety in real-world scenarios. In other words, the widely applied schedulability analysis can be precise and effective in determining whether the latency (*i.e.*, response time) of

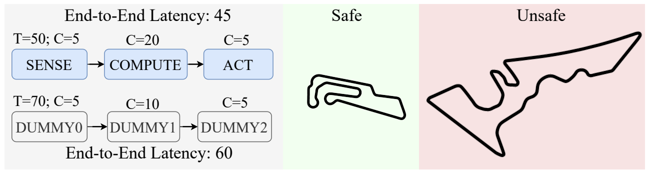

Figure 1: Experiment on F1Tenth racing car running on two different real racing tracks [7]. Details on the experimental setup are presented in Sec. 4. All units are in milliseconds. [Left] The blue task chain controls the behavior of the vehicle, and the gray chain is another task on the system. Due to blocking caused by other tasks on the system, the driving chain experiences extra end-to-end latency in addition to the execution time. Depending on the physical environment, this latency may or may not be acceptable. [Middle] An environment where the physical system can safely operate with worst-case driving chain latency, and [right] an unsafe environment for the system with the same worst-case latency.

a task set under certain system settings and scheduling algorithms exceeds a fixed deadline value, but is often too pessimistic in telling whether the latency exactly fits the system's physical requirements.

This leads to a key research question: *is it possible to verify the correctness of a real-time hybrid computing system without involving (too much) the pessimism of schedulability analysis and general environment models?* The answer is yes. By designing a digital twin—a model-based representation that mirrors the real system's physical and computational behaviors and its operational environment—we demonstrate that it is not necessary to (1) satisfy strict timing bounds or (2) consider an overly conservative environment model to ensure the actual safety of the systems in the physical world. Instead, it is possible to verify whether a system will work correctly in a physical environment using formal models of a system's scheduling, computation, and physical properties. To do so, one needs *model twins* such as formal models that accurately describe the necessary behaviors of the system.[1] Once the worst-case latencies of a workload are given, one can determine whether a controller is safe to use in a specific environment by simulating the workload under latency in a physical environment. If not, the system models can be used to test whether modifications to the system can ensure correct behavior. Example 1 further supports our argument that system correctness highly depends on its operational environment, and verification results may differ even if physical and computational models remain fixed. As a result, there is a need for an integrative safety assurance framework that takes the model twins of the system's physics, controller, and timing properties as inputs and finds whether the end-to-end (worst-case) latencies are feasible in specific operating environments/scenarios.

Example 1. (Illustrative Example) *Fig. 1 shows an example workload of a timer-driven driving chain, and a dummy chain (representing some non-driving work also running on the system) running on a ROS 2 system. We focus on modeling individual environments and not verifying against a worst-case scenario. This is important as verifying against a worst-case scenario may prevent the controller's usage in environments where it is 'good enough.' For instance, in Fig. 1, for the same end-to-end latency of driving chain under a fixed physical and computational model, an F1Tenth car can safely run in the middle track without crashing, but not in the right track.*

[1]We assume all provided models for different system components are validated correctly. The extent to which physical behavior deviates from this mathematical description is known as the 'sim2real gap,' which should ideally be minimized or, when necessary, over-approximated through careful choice of parameters and models.

Example 1 implies that verification results can change drastically even if physical and computational models remain fixed, depending on the operational environment. Instead of verifying the worst-case environment, we are aiming to verify the system for the specific environment in which the system will operate. Since it may not always be feasible to verify whether the models accurately describe the entire system, we consider the scheduler model separately from the computation and physics models. By symbolically modeling the scheduling behavior of the system, the controller workload, and other auxiliary tasks, we can find the guaranteed worst-case latency experienced by the controller.
Contribution. We propose a novel formal safety assurance framework, SOTERIA, which uses the model twins of the system's physics, controller, and timing properties to find whether the end-to-end (worst-case) latencies are feasible in specific operating environments/scenarios. Such safety may be guaranteed even if the latency is not feasible under *all* cases. Specifically, once the worst-case latency value for the workload is found, SOTERIA uses the physics, environment, and controller model twins to determine whether it is feasible to drive in that environment with this computation workload. To better illustrate how SOTERIA works, we present a rigorous formal model of ROS 2 scheduler, and then demonstrate an extensive case study of the proposed framework using F1Tenth racing car running on several real-world racing tracks.

2 SOTERIA: PROPOSED SAFETY-ASSURANCE FRAMEWORK

In this section, we introduce SOTERIA,[2] our proposed formal safety assurance framework. Fig. 2 provides an overview of the framework. We first describe the system components of SOTERIA, followed by the design flow for end-to-end verification.

2.1 System Model

Our system model $\mathcal{M} := \{\mathcal{M}_W, \mathcal{M}_S, \mathcal{M}_C, \mathcal{M}_P\}$ is the composition of workload model $\mathcal{M}_W$, scheduler model $\mathcal{M}_S$, controller model $\mathcal{M}_C$, and physical model $\mathcal{M}_P$. The relationship among these models is depicted in Fig. 2. To ensure safety, we show that a model $\mathcal{M}$ in environment $\mathbb{E}$ satisfies ($\vDash$) a property P, or $\boldsymbol{\mathcal{M} \,\|\, \mathbb{E} \vDash P}$, where $\mathcal{M}$ and $\mathbb{E}$ are composed in parallel, and P can be viewed as a system-level property. For instance, $\mathcal{M}$ may be an F1Tenth racing car, $\mathbb{E}$ a representation of obstacles and other geometric features such as racing track, which may be time-varying, and P a safety property, *e.g.*, collision avoidance.
Workload Model ($\mathcal{M}_W$) consists of a set of n tasks $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Depending on the inter and intra-dependencies of the tasks, the task set Γ can be independent, dependent as processing chains or directed acyclic graphs, etc. In addition, depending on the release pattern of the tasks, the tasks could be periodic, aperiodic,

[2]Soteria was the Greek goddess of safety, deliverance, and preservation from harm.

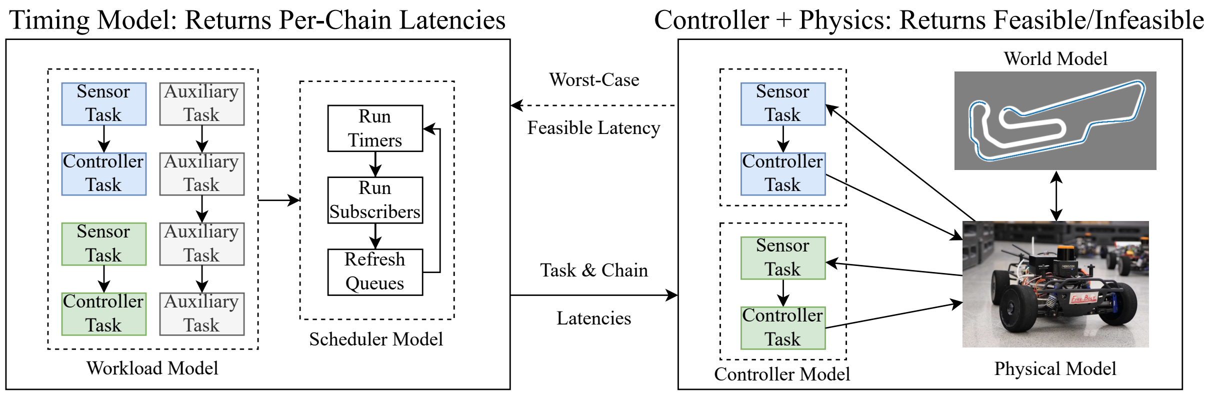

Figure 2: Overview of relationship among models in SOTERIA. [Left] The timing model provides a high-fidelity simulation of the scheduler used in the system. Queries for the timing model answer the question "Given a workload, what is the maximum latency experienced by each task or task chain?". Note that getting maximum latencies from the timing model using queries serves a purpose similar to that of standard scheduling analysis-based ones. [Right] The Physics and World (i.e., environment) model simulates the user's control code. The computation happens in one virtual task per chain (using the chain latency from the scheduler). The full scheduler is not simulated, and only tasks that affect the system's physical behavior are required (latency due to chain interference is already encoded in the latencies found in the scheduler model). Queries for the Physics and World model answer the question "Given the latency the workload may experience, can the car crash?"

or sporadic. In order for the scheduler model $\mathcal{M}_S$ to determine an accurate worst-case latency, the tasks in Γ should include all the tasks in the system that could affect the timings of the controller execution, even if they do not directly contribute to the computed values of the controller.

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 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.

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 model experiences some latency between reading the environment state and writing changes to the physics model due to the execution time of the controller code, and latency caused by other tasks in the system. 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 some of the tasks in Γ represent the controller model. The user must provide a worst-case execution time (not latency) for the controller.

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 $\mathcal{M}_C$. Note that most physical systems are hybrid dynamical systems, and precisely modeling is often hard. The objective is to use a model twin for a physical system with a minimal sim2real gap.

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. It 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. Multiple versions of the environment model may be implemented and tested with a single physics model.

2.2 End-to-End Verification

Before presenting the end-to-end verification steps, we first define the latency for tasks.

DEFINITION 1. (Task Latency) *The task 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).*

Tasks can be arranged in a chain layout, where the first task is triggered by some external event (or a timer) and releases some other task. The released task may cause other tasks to be released, until some final task is completed, ending the chain. This structure appears in systems as a way to read/receive sensor data, perform some processing, and perform some actuation in the physical system. Multiple chains can be in a system to perform different tasks.

DEFINITION 2. (Worst-Case Chain Latency) *The chain latency is the amount of time between the release of the first task in a chain and the completion of the last task in the chain. This is also referred to as* end-to-end *latency. The worst-case latency is the maximum latency to process a sensor input to actuation output in any system state.*

Using formal modeling methods, we can determine a worst-case latency for some control system, and determine whether the latency is acceptable to operate in a specific environment. Following are the design steps to implement SOTERIA:

- **Step 1:** Derive a timing model $\mathcal{M}_T := \{\mathcal{M}_W, \mathcal{M}_S\}$ that represents the timing properties of the system. Specifically, $\mathcal{M}_T$ should represent how decisions made by the scheduler affect the tasks that directly control the physical system. $\mathcal{M}_S$ must be validated for correctness. $\mathcal{M}_T$ determines the worst-case latency of the controlling tasks, which is used in $\mathcal{M}_C$. $\mathcal{M}_T$ should not include implementation details of the controller, only the layout of the controller and auxiliary tasks (ref. left box of Fig. 2).
- **Step 2:** Derive a model of the environment, $\mathbb{E}$ where the physical system will operate, a control algorithm $\mathcal{M}_C$, and a physical system $\mathcal{M}_P$ that is driven by the controller. For instance, we can derive the dynamics of a physical system (*e.g.*, differential equations for an F1Tenth car) and the inputs used to control it. $\mathbb{E}$ should represent the system's operating environment, and expose values that can be read by the controller model as sensor inputs. $\mathcal{M}_C$ should observe the state of the $\mathbb{E}$ and $\mathcal{M}_P$, and write values to $\mathcal{M}_P$ to influence its behavior. The right box of Fig. 2 shows the interactions between the environment, control, and physics models.
- **Step 3:** Separately verify the timing model $\mathcal{M}_T$ and physics model $\mathcal{M}_P$ for the controller $\mathcal{M}_C$ and latency. $\mathcal{M}_T$ uses the workload and scheduler to determine the maximum response time of the chains that directly affect the behavior of the physical system. Once the end-to-end latency of the controller tasks is determined, we can use $\mathcal{M}_C$, $\mathcal{M}_P$, and $\mathbb{E}$ to determine whether the end-to-end latency results in acceptable behavior. If the worst-case latency is inadequate, the environment or controller can be adjusted, or the workload can be modified to find a better worst-case latency.

3 APPLICATION EXAMPLE

3.1 Background

This section presents the necessary background for a short example of an application for SOTERIA. We use Uppaal [15] to verify the end-to-end safety assurance of the model twins of the computation workload, physical system, and environment. We leave the full validation and verification workflow to the Case Study in Sec. 4.

Uppaal. Uppaal provides tools to represent and verify timed hybrid automata. Timed automata use graphs of locations and state transitions to represent a system. A timed automaton [5] is a tuple $A = (L, \ell_0, X, \Sigma, E, I)$, where L is a set of locations representing some state in the system. $\ell_0 \in L$ is the initial state of the automaton. X is a set of clocks that evolve over time and $\Phi(X)$ represents clock constraints, *e.g.*, clock comparisons ($x <= 5 | x \in X$). Locations can have clock invariants $I : L \longrightarrow \Phi(X)$ attached, where the invariants must be met while the system is in the location. Moreover, Uppaal supports location invariant I containing differential equations that govern the clock rate, *e.g.*, $I := (v' = -9.8) \wedge (y' = v)$, where $\{y, v\} \in X$. Locations are connected via edges $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$. Edges can have guards/actions Σ attached (in Uppaal, actions are also denoted as 'updates'), where the associated guards must be true for the edge to be taken. Taking an edge to another location changes the state of the system to the attached location and can trigger actions, such as setting a clock or other variable values.

Speed	30mph	40mph	50mph
Freq			
1Hz	0.00	4.39	14.01
2Hz	0.00	0.00	0.221
5Hz	0.00	0.00	0.00

A

Brake Scale	0.0	1.0	10.0
Freq			
1Hz	0.00	1.73	5.84
2Hz	0.00	0.00	0.83
5Hz	0.00	0.00	0.40
10Hz	0.00	0.00	0.00

B

Table 1: Each table shows the minimum crash chance (with a 95% CI) for each experiment. Table A shows the crash chances of different controller frequencies with different speed limits for the leader car. Table B shows the crash chances when applying different scales to the leader car's hard brake option.

3.2 Cruise Control System

To illustrate a simple use case of SOTERIA, we use a 1D automatic cruise control system where a leader car can accelerate, decelerate, or maintain its speed, and a follower car uses a PI controller to maintain a constant time gap between it and the leader car.

Workload and Scheduler Models ($\mathcal{M}_T$). The system uses a simple Fixed Priority without preemption. The cruise control task has an adjustable period, and has an execution time of 10 ms. Another task runs in the system with a period of 100 ms and an execution time of 5 ms, and has a higher priority than the cruise control task. We can implement this using the SchedulerFramework model built into Uppaal. The model shows that the cruise control system has a latency of 15 ms for all the control periods used here.

Physics Model ($\mathcal{M}_P$). The follower car's motions are governed by differential equations, where $vF' = aF$, $xF' = vF$. aF is controlled by the follower vehicle controller.

Controller Model ($\mathcal{M}_C$). The controller is a PI controller that measures the velocities and relative positions of the leader and follower cars, and adjusts aF to try to keep a 1-second gap in between them.

Environment Model ($\mathbb{E}$). The leader car is controlled in a similar way to the follower car; $vL' = aL$, $xL' = vL$. aL, the acceleration of the leader car, is controlled by a random selector - at any given moment, it may decide to speed up (1.5 m/s^2), slow down (-1.5 m/s^2), or maintain the same speed. Depending on the model configuration, there is some chance for a hard brake (by default, -6 m/s^2).

Results. For each configuration, we can query the probability of whether the follower car can crash into the leader car:

$$\texttt{Pr[}\leq \texttt{MAX\_TIME]} \diamond xL - xF \leq 0$$

Uppaal outputs a probability window of the condition in the query being true, with a configurable confidence value.

We ran two tests. First, we varied the maximum allowed speed of the leader car. We placed the follower car 1 second behind the first. Both started at half of the leader's maximum speed. We tested 3 maximum speeds along with 3 possible control frequencies. This demonstrates that as long as the first car holds to a speed limit of 30 mph, a 1 Hz control loop for the follower's cruise control is enough to prevent a collision. Once the speed limit is set to 40 mph, a 2 Hz control loop is required to prevent a crash. Finally, at 50 mph,

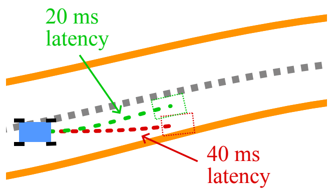

Figure 3: The effects of latency on path safety, where the green trajectory with less latency (20 ms) is a safe trajectory, and the red one (with 40 ms latency) is unsafe.

the 1 Hz and 2 Hz control loops were not enough - only the 5 Hz controller could drive without crashing.

Second, we varied the maximum braking force for the leader's hard-brake action. With a brake scale of 0, the leader car never performs a hard brake. With a scale of 1, the leader car's hard-brake action applies -6 m/s^2 of deceleration to the leader car. The scaling of 10.0 represents a hard crash into a solid object, applying -60 m/s^2 to the leader car. In the presence of any sudden braking, the 1Hz controller could cause crashes. The 2 Hz could handle the "normal" -6 m/s^2 sudden braking, and the 5 Hz model prevented the follower car from crashing in all three scenarios.

These two tests demonstrate that SOTERIA can be used to select an appropriate controller configuration depending on the expected environment (speed limit, possibility of crashes) in which the system will operate.

4 CASE STUDY WITH EXPERIMENTS

Here, we verify the safety of an F1Tenth [22] vehicle running on a racing track using a Pure Pursuit [14] path follower. The vehicle uses localization and a predefined path to drive along racetrack without crashing. The vehicle uses the Robot Operating System (ROS 2) to support communication between different software packages and sensors, so the driving algorithm is subject to decisions from the scheduling algorithm used in the ROS 2 executor. The scheduler model, $\mathcal{M}_S$, implements the task selection process used by the ROS 2 executor, and the workload model, $\mathcal{M}_W$, represents tasks running under the executor. The physics model, $\mathcal{M}_P$, simulates the motion of the vehicle on a 2-dimensional plane. The environment is abstracted into just the centerline of the track, and success is measured by whether the vehicle ever moves far enough away from the centerline (such that it may collide with the track borders). We show an example scenario in Fig. 3.

The tasks used to control the vehicle involve reading sensor data, making driving decisions, and sending control data to motors. We abstract these tasks into three tasks: a sensing task, a compute task, and an actuation task, which form a processing chain. We also add other randomly generated tasks to $\mathcal{M}_W$ to simulate the effects of other tasks on the system.

We use $\mathcal{M}_S$ and $\mathcal{M}_W$ to calculate the worst-case end-to-end latency of the controlling chain caused by executor decisions and blocking time due to other tasks.

We perform three tests: a schedulability test, where we measure the effectiveness of $\mathcal{M}_S$ and $\mathcal{M}_W$, a sustainability test, where we show that it is sufficient to test with only the worst-case response time, and an end-to-end test, where we demonstrate the use of the SOTERIA framework.

4.1 ROS 2 Background

ROS 2 is a collection of software tools and libraries supporting robotics application development. Tasks in ROS 2 are written in *callbacks*, which are called when certain conditions are met. For example, timer callbacks run on a set period, and subscriber callbacks run when a message is published to a specific topic.

ROS 2 callbacks run under executors, which maintain separate queues for each type of callback. At runtime, executors refresh each queue, collecting incoming messages and events, and queues up to one job of each callback type. The executor then runs the queued callbacks in order of type - timers run first, then subscribers, then services, etc. Only once each queue is depleted does it move to the next. Once all queues are completely empty, it begins again and refreshes the queues.

4.2 Workload Model ($\mathcal{M}_W$)

We consider the workload as a set of n processing chains $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$. Each processing chain (in short, *chain*) consists of a sequence of callbacks. For instance, i^{th} chain $\tau_i = \langle \tau_{i,1}, \ldots, \tau_{i,|\tau_i|} \rangle$ is a sequence of $|\tau_i|$ callbacks. The first callback $\tau_{i,1}$ of any chain τ_i is a timer callback. The timer callback is characterized as $(c_{i,1}, T_i)$ where $c_{i,j}$ is the worst-case execution time (WECT) and T_i is the period. The other callbacks, known as subscriber callbacks, are only characterized by their WECT $c_{i,j}$ and triggered to execution by their subscribed callback. Besides the sequence of callbacks, a chain is further characterized as a tuple (C_i, T_i), where $C_i = \sum_{\forall j} c_{i,j}$ is the WECT of the chain and T_i is the period (same as the timer callback's period of the chain). Without loss of generality, we consider integer time instances only, aligned with the granularity of the clock tick supported by UPPAAL.

4.3 Scheduler Model ($\mathcal{M}_S$)

We implement ROS 2 executor behavior in UPPAAL for modeling the scheduling policy of ROS 2. The executor model takes a list of timer and subscriber callbacks, including their layout and execution times, and determines the maximum possible callback and chain latencies due to the scheduling policy and task interferences. Since the scheduler and callback models are implemented as timed automata, their properties can be found with symbolic queries. The models keep track of the response times of chain instances, so a symbolic query for the supremum of each chain's response time will determine the worst-case responses of the system. We describe how the callback and executor models function in detail below.

Callback Model: Fig. 4 presents the UPPAAL time automaton template of ROS 2 callback. This model handles features for both timer and subscriber callbacks. UPPAAL instantiates one copy of the callback template for each callback in the system. Each callback starts in the `postJobComplete` state and, depending on the callback type, waits for some condition. If the callback is a timer, it checks if enough time has passed on the per-callback timer. If so, it proceeds to `handleOverload`, and if not, moves to `timerSleeping` until enough time has passed. `handleOverload` adjusts the release time of the callback if the next release time is more than one period in the past. If the callback is a subscriber, it checks it has received at least one message by calling `hasMessage()` and moves to `subSleeping` or `taskReleasing` depending on the result. Upon

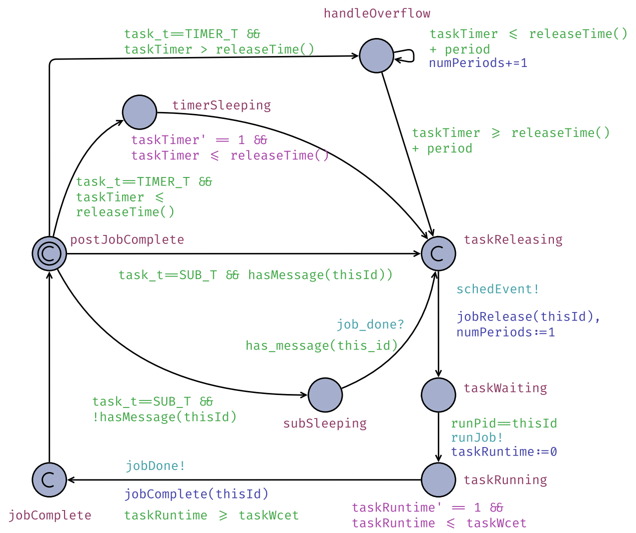

Figure 4: ROS 2 Callback template. Color code: State labels (L), State invariants (I), Guards and Actions (Σ), Synchronizations. The initial state, ℓ_0, is denoted with an extra circle.

entering taskReleasing, the callback emits a schedEvent! event, waking up the executor if it is sleeping state, and adds itself to the appropriate wait set with jobRelease(). jobRelease() does not add the task to the executor's wait set immediately — the executor will only recognize the release after the executor has called refresh(). The callback waits until the executor selects it using the global runId and receives a runJob! event. Once both conditions are satisfied, the callback resets its runtime timer and sleeps until its WCET has passed before notifying the scheduler over the jobDone! event. It also calls jobComplete(thisId), which sends a message to the next callback in the chain, if any exists. If the task is a timer, jobComplete(thisId) adjusts the taskTimer clock for the next release. In case a timer callback is blocked for more than one period, the handleOverflow state moves the next release time of the timer to the next period in the future.

Executor Model ($\mathcal{E}$): Uppaal time automaton template of ROS 2 executor model is shown in Fig. 6. The executor maintains ready sets for each callback type, and depletes each set in order of type. It runs all timers in the ready timers set, in order of registration (when the timer was declared and added to the executor). Once the ready timers set is empty, it runs all the subscribers in the ready subscribers set. Once the subscribers set is empty, it calls refresh(), which updates the ready timers and subscribers with pending jobs. As a result, refresh() can only happen once both the ready timer and subscriber sets are empty. Only one job of each individual callback is considered, so if there are two queued messages for a subscriber, only one job for that callback will be released. If no jobs are added to the sets during a refresh, the scheduler sleeps until a new job is released. When the scheduler runs a job, it sets a globally-shared runId value to the id of the selected job and sends a runJob! event. When a runJob! event fires, each task compares the shared runId value with its own ID, and if they match, it has been selected to run. The task waits for its execution time, releases any additional tasks, and sends a jobDone! event, which allows the

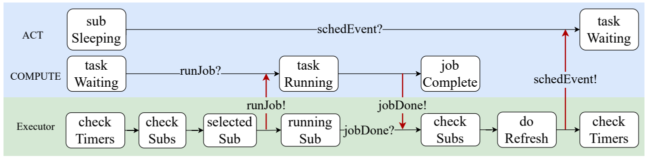

Figure 5: Synchronized interactions between the executor and scheduler models. The executor template is colored green, and the ACT and COMPUTE templates, instantiated from the callback template, are colored blue. During callback selection, tasks are notified of their turn by the scheduler's runJob! channel. Once the scheduler sends runJob!, it waits on the jobDone? channel, which the job broadcasts to return control back to the scheduler. The scheduler and callback templates use schedEvent! to notify each other of newly released tasks and queue refreshes.

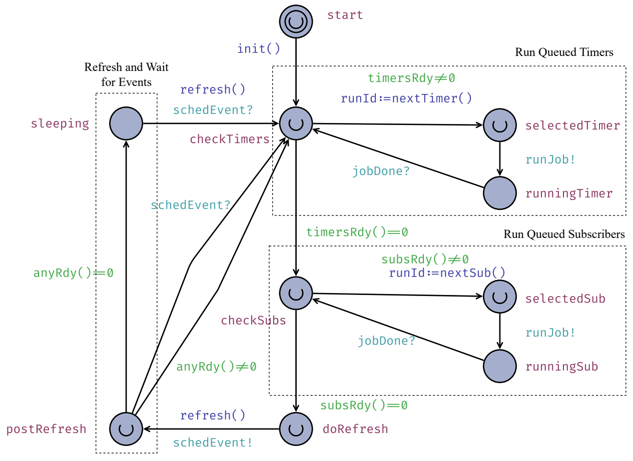

Figure 6: ROS 2 Executor model. Color code: State labels (L), State invariants (I), Guards and Actions (Σ), Synchronizations. The initial state, ℓ_0, is denoted with an extra circle.

executor to move on to the next job. For simplicity, the executor model only implements timer and subscriber callbacks, the most common types of callbacks used in ROS 2.

Synchronization between Callback and Executor Model. The schedEvent! synchronization is used to ensure tasks and the scheduler are kept aware of state changes. When the scheduler performs a queue refresh and finds that there are no jobs to run, it waits on schedEvent? in the sleeping state. When a timer releases a job, it wakes the scheduler by broadcasting schedEvent!. Likewise, released callbacks wait for queue refreshes on schedEvent?, which is sent by the scheduler whenever a queue refresh is performed. Since subscriber callbacks can only be released once a calling callback completes execution, subscribers wait for other jobs to complete on jobDone?, where it proceeds to taskReleasing once a message has been received. An example of a task and scheduler interaction is shown in Fig. 5.

Computation of chain latency. To compute chain latency, we utilize the following properties of ROS 2 executor scheduling policy:

Property 1. At most, one instance of a callback executes in a processing window [9, 26].
Property 2. The callback instances of a chain instance execute in consecutive processing windows one by one [9, 26].

Corollary 1. *Among multiple concurrently active chain instances in ROS 2, the earlier-released chain instance will always be completed before later-released instances if the instances are from the same chain.*

Proof. The Corollary is a direct consequence of the highlighted properties of ROS 2 executor scheduling policy. □

The model supports a `MAX_CHAIN_INSTANCES` value which controls the maximum amount of instances of a single chain that can be running at a time. To measure the end-to-end latency of callback chains, each chain has an array of timers—one for each possible concurrent chain instance—called `chainLatencies` to record the time between chain releases and completions. Each chain's `chainLatencies` array serves as a queue to record for how long each currently running chain instance has been released. Once a chain timer releases, it sets the first free timer in `chainLatencies` to 0, and allows it to count up. Due to the queue refreshing behavior, an earlier-released chain instance will always be completed before a later-released instance if the instances are from the same chain (Corollary 1). When the last callback in a chain calls `jobComplete`, it clears the *first* value in `chainLatencies` and moves any remaining values one position closer to the start of the array. As a result, the latency of the oldest currently running instance of a chain will always be stored in the first value of that chain's `chainLatencies` array. To determine the maximum possible latency of any chain, it is enough to query for the supremum of the first value in `chainLatencies`. If a timer callback tries to start a chain instance and `MAX_CHAIN_INSTANCES` chain instances are already running, an overloaded flag is set, and the verification fails.
Validation. We can validate that the executor and callback models represent the described behavior of the ROS2 executor using queries:

$$A\square\ !\texttt{duplicateWaitingJobs()}$$
$$A\square\ \texttt{waitsetsOrdered()}$$
$$A\square\ \mathcal{E}.\texttt{checkSubs imply nTimerRdy() == 0}$$
$$A\square\ \mathcal{E}.\texttt{doRefresh imply nTimerRdy() == 0}$$
$$A\square\ \mathcal{E}.\texttt{doRefresh imply nSubsRdy() == 0}$$

`duplicateWaitingJobs()` checks if there is ever more than one job of each callback in the waitsets at any time. During the refresh process, the executor only selects one job of each individual callback, even if there are multiple messages queued. `waitsetsOrdered()` checks whether the jobs in the waitset are sorted in the callback registration order. For each callback type, the ROS 2 executor selects the callback in order of their declaration. In the `checkSubs` state, all timer jobs in the waitset must have been run and removed. Similarly, in the `doRefresh` state, all timer and subscriber jobs in the waitset must have been removed, sincewaitset the executor only refreshes the waitset once they are all empty.

Evaluation. To evaluate the scheduler model, we first present a numerical example to illustrate the computation of worst-case latency using $\mathcal{M}_S$. Then, we present a large-scale comparison with a state-of-the-art schedulability test [9].

Table 2: The taskset used to represent the driving setup. These timings are arbitrary and represent a simplified workload.

Label	Name	Callback Type	Period	WCET	Calls
Task(0)	SENSE	Timer	50 ms	5 ms	DRIVE
Task(1)	DRIVE	Subscriber		20 ms	ACTUATE
Task(2)	ACTUATE	Subscriber		5 ms	
Task(3)	DUMMY0	Timer	70 ms	5 ms	DUMMY1
Task(4)	DUMMY1	Subscriber		10 ms	DUMMY2
Task(5)	DUMMY2	Subscriber		5 ms	

Example 2. *If we pass the taskset described in Table 2 to the timing model in Uppaal, it will instantiate the callback templates with the provided properties. Queries ran against the model will now reflect how the given taskset behaves under the ROS 2 executor, so we can use queries to determine the maximum latency each callback can experience. Note that chain latencies are tracked in the* `chainResults` *array, so querying for the supremum value while in a chain-ending state allows us to determine the maximum latency experienced by a chain.*

$$\texttt{sup\{Task(lastInChain).jobComplete\}}$$
$$:\texttt{chainResults[chainIndex]}$$

This query should be called on the ACTUATE callback, which is the last task of the driving chain. Uppaal returns a supremum for the clock given in the query. The `chainResults` *array is updated whenever a chain completes. The values in* `chainResults` *are clocks and increase over time, so the value must be measured in the same instant that it is written. The query checks for the supremum only when the callback is in its* `jobComplete` *state. If run for the last callback in the first chain, the query returns the worst-case latency of the chain, 50.*

Because the sup query in Uppaal is a symbolic query, where the entire possible state space is examined, it is guaranteed that the results will always hold for a taskset and that the latency supremum for each callback will never be exceeded.

Comparison with schedulability test. Previous works (e.g., [9, 10, 26], etc.) use analytical schedulability tests to determine the maximum latencies of ROS 2 callbacks and execution chains. Blaß *et al.* [9] provide a worst-case latency bound for ROS 2 processing chains by taking advantage of the fact that only one instance of each callback can be run between executor refreshes.

To compare our model-based approach with [9], we use randomly generated tasksets from 8 different utilizations, and compare the latency results from both methods. We chose 8 utilizations from [0.1, 0.8]. For each utilization value, we generate 500 random workloads using UUniFastDiscard [8]. Each workload has up to 4 callback chains, and each chain may have up to 4 callbacks. Each chain begins with a timer, where each timer has a period between 20 and 100 time units. To reduce the search space for model-based verifications, we limit the timer periods to be divisible by 10. We use UUniFastDiscard to distribute the utilization between the generated chains, and again to distribute each chain utilization to the callbacks. For each generated workload, we calculate the end-to-end latencies

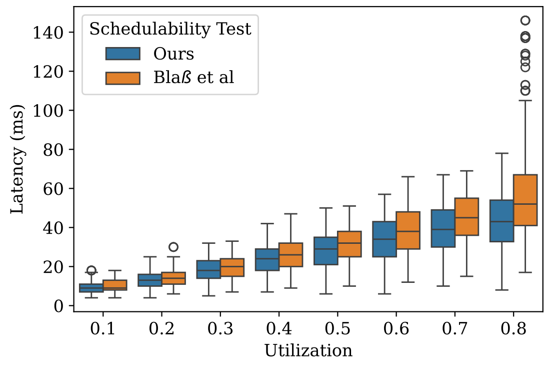

Figure 7: Worst-case latency bound using our model and Blaß *et al.* [9]

using the schedulability test from [9] and our model-based method. For our model, we set `MAX_CHAIN_INSTANCES` to 2. The method from Blasß *et al.* has an additional per-callback liveliness parameter, which we set to two periods. Due to this difference, and for a fair comparison, we only show workloads that were considered schedulable by both tests.

A summary of the worst-case latency comparison is presented in Fig. 7. As expected, the model-based test always performs better than, or at least equal to, the schedulability test. This is expected since the model-based method performs a symbolic search of all possible state space and returns exact worst-case latency, whereas the schedulability test [9] only returns a safe upper bound of worst-case latency. Accurate worst-case latency estimates are crucial. Overestimating may cause simulations to incorrectly label a sufficient controller as inadequate.

4.4 Physics Model ($\mathcal{M}_P$)

To represent the F1Tenth car, we use a two-wheeled bicycle model [4, 24] which is commonly used as an abstraction for four-wheeled Ackermann-steering vehicles. The bicycle model combines the two front wheels into a single wheel, and the two rear wheels into a single wheel, and while turning, the car rotates about the center of its rear axle. Tire slip is not considered.

In the two-wheeled bicycle model, the car's state is described by five variables, representing the vehicle's 2-D position (s_x, s_y), velocity v, yaw δ, and steering angle Ψ. The state change rates are computed as, $\dot{s}_x = v\cos(\Psi)$, $\dot{s}_y = v\sin(\Psi)$, $\dot{\delta} = \text{steerLimit}(\text{steeringIn})$, $\dot{v} = \text{accLimit}(\text{accIn})$, and $\dot{\Psi} = \frac{v}{l_{wb}}\tan(\delta)$, where l_{wb} is the vehicle's wheelbase length or the distance between the front and rear tire. This affects the turning radius of the vehicle. The variables `steerRate` and `limitAcc` limit the steering and acceleration inputs to the physical limits of the vehicle, and `steeringIn` and `accIn` are control inputs, which can be set by the computation model.

By manipulating the control inputs over time, the differential equations simulate the vehicle's motion and update state variables.

4.5 Controller Model ($\mathcal{M}_C$)

The controller generating vehicle control inputs is a path-following Pure Pursuit [14] controller. The path is given as a set of waypoints,

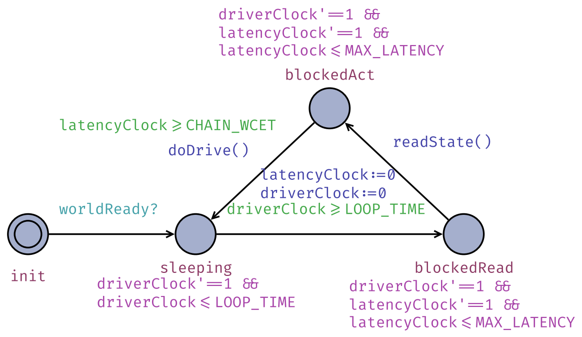

Figure 8: The model used to represent the driving callbacks when used alongside the physics model.

and the controller uses the car's current location along with a lookahead value to select a target point along the line. It uses the target point, along with the car's current state, to select a steering and throttle value. The path follows the center of the track, and the vehicle is considered 'crashed' if the vehicle's distance from the line exceeds half of the track width, which would cause the vehicle to collide with a wall. Obstacles are not considered.

Controller model implementation. Callback chains that directly influence the vehicle are abstracted in the controller model as a single task. Fig. 8 shows the model that represents the controller chain. The model performs the actions of the driving chain: `readState()` stores the current state of the vehicle, and `doDrive()` performs the driving calculations, updating `steeringIn` and `speedIn`. The model uses two clocks: `driverClock` to represent the chain's timer, and `latencyClock` to represent latency experienced by the chain. The compute model waits in `sleeping` until the controller period passes, and moves to `blockedRead`, resetting both the `driverClock` and `latencyClock`. The model can wait in `blockedRead` for some time, for at most `MAX_LATENCY` (calculated by the scheduling model above) time units. The next transition to `blockedAct` records the state of the vehicle. The model must remain in `blockedAct` until the `latencyClock` reaches `CHAIN_WCET`, which represents the execution (and therefore minimum) time that the driving chain takes. Once that time has passed, the model can remain in `blockedAct` until the `latencyClock` reaches `MAX_LATENCY`. Overall, the model represents blocking time between each callback in the chain, as well as the execution times of each callback. Note that this does not fully represent the callback behaviors, just the possible latency experienced by the chain. The physics and controller models run together in the same file. It is not necessary for the controller model to simulate all tasks in the system—it only needs to simulate the tasks relevant to controlling the car, nor does it simulate the scheduler. This is possible because the latencies calculated separately in the scheduling model already consider the scheduling decisions and possible interferences from all tasks in the system.

The environment model ($\mathbb{E}$) runs periodically, each time calling `updateWorld()`, which compares the vehicle state with the environment representation and updates the `trackDist` variable to check for safety in the verification queries.

The implementation of `runDrive()` and `updateWorld()` are done in an external shared library, which can be called from Uppaal. This is done to ensure that the driving logic matches the real-world controller as closely as possible, allows for code reuse, and provides performance improvements over implementing it in UPPAAL.

Safety verification. Since the physics and controller models use differential equations to describe state evolution, they cannot be queried using symbolic queries, and an exhaustive search is not possible. Therefore, we use statistical queries, where UPPAAL performs a large number of simulations and makes empirical measurements of the model properties.

$$\texttt{Pr[<= MAX\_TIME]} \diamond |\texttt{line\_dist}| > \texttt{TRACK\_WIDTH/2}$$

The above query returns an interval for the probability (*e.g.*, either $[0, 0.001]$ or $[0.9, 0.999]$) with a confidence level (*e.g.*, 95%) that the given constraint is true for a given confidence value. If the confidence value meets the application requirements, then the constraint is likely to hold.

To produce the results in Fig. 1, we set maxLatency of each callback to the latencies found in the scheduler model queries and the WCET values from the taskset. The safety query succeeds on the safe map but fails on the unsafe map. By adjusting the times of the dummy chain (or the driving chain) to reduce the driving chain latency, the controller can be made safe on both maps.

The time horizon and the number of simulations can be adjusted using `MAX_TIME` and UPPAAL's Statistical Parameters settings. To obtain reasonable confidence, a large number of simulations is required. The number of simulations run is decided by UPPAAL for a given confidence value, but can also be specified by the user

4.6 End-to-End Verification

Once the end-to-end response time is calculated using the scheduler model, we need to evaluate whether it is feasible for the controller to drive on a track while running with latency. The times at which the controller model reads the vehicle state and outputs new control values are affected by the WCET and latency values from the taskset definition and the executor model. Increasing the time between sensing the environment at the beginning of the chain and causing a physical actuation at the end means that the system responds slower to changes in the environment and system, and may take actions that lead to unsafe states.

The vehicle may perform differently depending on the environment configuration, since individual environments may require lower response times than other environments. Therefore, the controller should be tested in the environment in which it is expected to be used. To test the controller in an environment, it needs to be simulated across a large variety of states and checked for safety. We do this using UPPAAL's statistical queries. The vehicle starts at a random position on the centerline, and the vehicle and controller are simulated up until some time horizon while being monitored for safety. If any run violates a safety constraint, then the environment is infeasible for this combination of models.

Environment Model. The environment is a racetrack stored as a series of waypoints forming a centerline. The centerline is loaded in an external library, which contains functions accessible to UPPAAL via its *Foreign Function Interface.* Both the driver and environment model call functions from this library to make read waypoints, make driving decisions, and update safety properties, which are used by queries to determine whether the vehicle is in a safe state. In UPPAAL, the Environment template polls the vehicle's position and

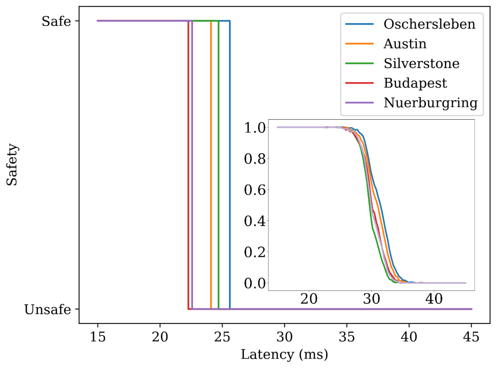

Figure 9: The outer plot shows the maximum possible latency for safe driving on a selection of 5 maps. The inner plot shows the success rates of running the driver with different latency values on a selection of possible maps.

calculates its proximity to the nearest centerline segment, storing the value in `line_dist`. If `line_dist` ever exceeds the width of the track, then the vehicle is in an unsafe state.

Evaluation. To evaluate the proposed method, we randomly generate workloads composed of non-driving tasks that can add latency to the driving chain and a driving chain with randomly selected execution times to represent different possible system implementations. The generated workloads always contain the callbacks from Fig. 1 but with randomly selected execution times. The workloads also contain some randomly generated interfering tasks, which to not directly affect the vehicle's behavior, but may add latency. For example, the SENSE callback represents the process of reading sensor input and determining the vehicle's location on the racetrack. Different sensors and algorithms can be used for this step and can take varying amounts of time. The same goes for the DRIVE and ACTUATE steps. We use the ROS 2 executor model to estimate the maximum latency of the callbacks in each workload's driving chain and use a schedulability test [9] for comparison. For both sets of calculated latencies, we test each configuration with multiple racetracks. We use 10 racetracks from Betz *et al.* [7], which are traced from **real-world F1 racetracks** and scaled down for F1Tenth cars.

Sustainability Analysis. We verify the sustainability of the verification results to ensure that if a system is identified as safe in the worst-case latency, the system will remain safe even if it experiences less latency than in the worst-case during runtime. Fig. 9 shows the sustainability results across different racing tracks. Notice that the latency at the transition of step function for each track is the latency safety upper bound; any latency less than that would allow the system to remain safe and vice versa. Since this evaluation is based on a statistical query, such sustainability results hold with high confidence but are not guaranteed.

Evaluation on worst-case latency. To demonstrate the end-to-end usage of SOTERIA, we generate 100 random tasksets, where the utilization of each taskset is 50%. The first chain is set to have 3 callbacks, has a period of 40 ms, and serves as the driving chain for the vehicle. We add up to 3 more chains, each of which can have up to 4 callbacks and have random periods from 20 to 100 ms. We use UUniFastDiscard to assign execution times for all callbacks. We then find the maximum response time of the driving chain using

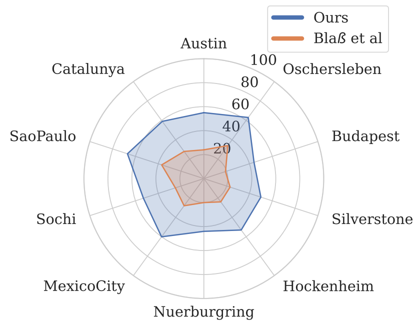

Figure 10: The number of tasksets that Uppaal determines can safely drive, for each map, using latencies from our method and Blaß *et al.* [9].

Table 3: A selection of results from the end-to-end tests using randomly generated workloads. Green checkmarks denote latencies where Uppaal determined that the controller can run safely at a high confidence.

Latency		Aus		Osc		Bud		Sil		Hoc		Nue		Mex		Soc		Sao		Cat	
M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A	M	A
56	58	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
60	68	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
52	58	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×	×
60	62	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	×	×	×
52	58	×	×	×	×	×	×	×	×	×	×	×	×	×	×	✓	×	×	×	×	×
52	70	×	×	×	×	✓	×	×	×	×	×	×	×	×	×	✓	×	×	×	×	×
34	64	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×	✓	×
48	50	✓	×	✓	×	✓	✓	✓	✓	✓	×	×	×	✓	✓	✓	✓	✓	×	✓	✓
34	36	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
44	46	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

our model-based method, and the analytical method from Blaß *et al.*. For both latency values, we test each workload on 10 maps, where we use an Uppaal statistical query to determine whether the model can perform on each map for the given response times.

Fig. 10 shows the results from this test. The number of successful tasksets for each map is shown on the radius axis and the 10 maps on the angle axis. As the model-based method always finds a lower or equal response time bound compared to the analytical method, more taskset configurations can be used successfully on the tracks.

Table 3 shows whether the controller could safely drive the vehicle across a selection of 10 tracks. We selected 10 random tasksets from the case study and showed the computed latency values, as well as whether each latency value is acceptable for each track. These results validate our hypothesis related to SOTERIA with a specific environment model. For instance, the first eight tasksets failed in the worst-case environment model, but there are several tracks that are safe for these tasksets.

5 RELATED WORK

Formal Methods for CPS. In a recent survey of autonomous systems and the challenges they pose, Wing [30] 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.* [19] verify a "safety net" for a waypoint navigation controller using ModelPlex [21] to synthesize a monitor using theorem proving. Lin *et al.* [20] extend the work by combining theorem proving and reachability analysis with Flow* [12] for synthesizing switching monitors. In the context of F1Tenth vehicles, Ivanov *et al.* [17] verify the safety of a neural network controller using their Verisig tool [18]. Vehicles operate at constant throttle in a structured environment, and the effect of missing LiDAR rays due to reflections is evaluated. [23] presented a systematic literature review on verification and validation for safe autonomous cars. Our concurrent works [28, 29] utilized environment feedback to design mixed-criticality scheduling with end-to-end verification.

ROS 2 and Schedulability Analysis. ROS 2 recently received significant attention from the real-time systems community after the pioneering work by Casini *et al.* [10]. Many works (few to mention [9, 26, 27]) subsequently improved the proposed worst-case latency bound of ROS 2 workloads and also proposed modified executor schedulers [3, 6, 13]. 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 [25], exact scheduling test for non-preemptive self-suspending tasks [31], etc.

Besides standard scheduling problems, formal methods are also used for timing analysis of ROS 2 [16] and AUTOSAR [32]. The model in [16] does not consider communication between callbacks, and instead requires that the release times of callbacks be predetermined and provided to the model as inputs. By simulating the chain relationship between callbacks, our model requires only the layout and execution times as input, and calculates the release times using the behavior of the ROS 2 executor.

6 CONCLUSION AND FUTURE WORK

We presented SOTERIA, an end-to-end safety assurance framework that uses model twins of a system's computational and physical components operating in a specific environment. Our framework prioritizes safety assurance within the intended operating environment rather than pessimistically verifying against worst-case conditions. In addition, we conducted a detailed case study of SOTERIA on a ROS 2-based F1Tenth racing car. Our rigorous formal model twin of ROS 2's functional behavior enabled us to determine the exact worst-case latency for verifying the car's safety across ten different real-world racing tracks. Extensive empirical evaluation confirms that our approach outperforms state-of-the-art methods in safety assurance. However, one of the key limitations of model-based safety assurance is that it is more computationally expensive than analytical methods, as symbolic verification explores the entire state space. A well-informed design can help reduce the search space, making model-based methods feasible for more applications. Additionally, developing accurate models of hybrid dynamical systems is challenging, particularly for compplex physical systems. Our case study focuses on a static environment, but future work could explore applying SOTERIA in dynamic environments. Building model twins for dynamic environments would require two-way verification, where the digital twin's verified model informs the physical system's design, while real-world feedback is used to refine the model twin.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful feedback. This work was supported in part by the National Science Foundation under Grants CMMI 2246672 and CCF 2124205.

REFERENCES

[1] ROS 2 Documentation. https://docs.ros.org/en/foxy/index.html.

[2] Autosar adaptive platform. https://www.autosar.org/standards/adaptive-platform/, 2022. [Online; accessed 13-October-2022].

[3] 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*, 43(11):3732–3743, 2024.

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

[5] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235, 1994.

[6] 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*, pages 301–306, 2022.

[7] 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.*, 3:458–488, 2022.

[8] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[9] 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)*, pages 41–53. IEEE, 2021.

[10] 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*, pages 1–23. Schloss Dagstuhl, 2019.

[11] 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*, 33(4):92–108, 2016.

[12] 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*, pages 258–263. Springer, 2013.

[13] 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)*, pages 251–263. IEEE, 2021.

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

[15] A. David, K. G. Larsen, A. Legay, M. Mikučionis, and D. B. Poulsen. Uppaal SMC tutorial. *International Journal on Software Tools for Technology Transfer*, 17(4):397–415, 2015.

[16] L. Dust, R. Gu, C. Seceleanu, M. Ekström, and S. Mubeen. Pattern-based verification of ros 2 nodes using uppaal. In *International Conference on Formal Methods for Industrial Critical Systems*, pages 57–75. Springer, 2023.

[17] 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*, pages 1–7, 2020.

[18] 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*, pages 169–178, 2019.

[19] 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*, pages 122–141. Springer, 2021.

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

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

[22] 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*, 123, 2020.

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

[24] P. Riekert and T.-E. Schunck. Zur fahrmechanik des gummibereiften kraftfahrzeugs. *Ingenieur-Archiv*, 11:210–224, 1940.

[25] 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)*, pages 1–6. IEEE, 2020.

[26] 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)*, pages 231–243. IEEE, 2020.

[27] 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)*, pages 53–65. IEEE, 2022.

[28] 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)*, pages 98–102. IEEE, 2024.

[29] K. Wilson, A. Al Arafat, J. Baugh, R. Yu, and Z. Guo. Physics-informed mixed-criticality scheduling for f1tenth cars with preemptable ros 2 executors. In *2025 31st Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 2025.

[30] J. M. Wing. Trustworthy AI. *Communications of the ACM*, 64(10):64–71, 2021.

[31] 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)*, pages 1228–1233. IEEE, 2019.

[32] 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*, 201:111675, 2023.