Physics-Aware Mixed-Criticality Systems Design via End-to-End Verification of CPS

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

***Abstract*—Autonomous systems are heavily used in many safety-critical systems, such as industrial automation, autonomous cars, Industrial Internet of Things (I-IoT), etc. Verification of the functional and temporal correctness of such systems is necessary before deployment to ensure their safety. However, due to the presence of physical systems in the continuous-time domain and computational models in the discrete-time domain, end-to-end verification of these systems is highly challenging. Existing formal methods focus on verifying physical models assuming static or simplified computation models. In contrast, existing real-time systems focus on satisfying strict timing bounds but do not care how those bounds are obtained and how they relate to physical safety. Our approach bridges these two domains, and constitutes an end-to-end verification framework for arbitrary physical models and computational models incorporated within a cyber-physical automated system. By allowing the interaction between the computational and physical models, our verification framework enables a fine-grained scheme that verifies against the local environment instead of verifying against global worst-case assumptions. Moreover, to support locally varying worst-case scenarios, a mixed-criticality system is proposed where the system supports several critical models and switches among the modes based on environmental uncertainty. Finally, a proof-of-concept evaluation of the proposed framework is reported.**

1. Introduction

Cyber-physical systems such as industrial control systems, autonomous vehicles, I-IoT, *etc.*, 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. Hence, 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 [1]: (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 systems, they tend to be overly pessimistic as all timing bounds are derived for worst-case scenarios, including the system's operating environment.

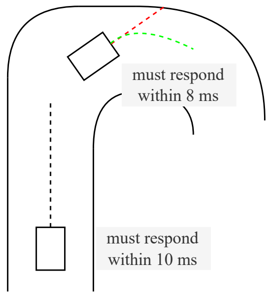

Figure 1. An autonomous vehicle in two positions on a racetrack. For some states, a tighter latency bound is required for safety. While driving on a straight section of track, the controller can operate safely with a 10 ms latency, but while on a curved section, must respond within 8 ms. Standard verification methods would report that a 10 ms latency is required for safe operation on this track, but using mixed-criticality scheduling, it may be possible to use a dual-critical system where the car would run in straight segments of the track in one system mode and the curved segments in another system mode.

Moreover, when verifying a system and environment with a *single* timing bound that applies in all cases, the situation in Fig. 1 may result in one finding that the environment cannot be safely navigated.

Contribution. We propose an end-to-end verification framework that collaboratively verifies computational and physical models. By allowing bi-directional communication of maximum observable latency from the computational model and maximum allowable latency by the physical model for safe operation, we propose a fine-grained verification framework that only verifies locally (while remaining safe eventually[1]) instead of the worst-case global scenario.

Furthermore, we propose a mixed-criticality system[2] that supports several critical modes and switches among the modes based on environmental variations to support locally varying worst-case allowable latencies. By considering the possibility of a mode switch during verification, the proposed method could decide that the environment is actually feasible, making it a worthwhile improvement.

Related Work. Our proposed framework can be compared with the Simplex architecture [4], [5]. In the Simplex architecture, an unverified High-Performance Controller (HPC) is accompanied by a safe High-Assurance Controller (HAC)

1. This is based on our hypothesis that locally safe operations eventually keep the system safe globally–formal proofs are left for future work.

2. Please see Burns and Davis [2] for a detailed introduction and up-to-date survey of the research related to mixed-criticality systems, and Arafat et al. [3] for recent work on generalized mixed-criticality systems.

with a (verified) safe Monitoring and Decision Logic (MDL). When the HPC's control action could lead the system to an *unrecoverable state* where the HPC can no longer guarantee safety, the MDL automatically switches to the HAC to ensure safety. The MDL is usually achieved by runtime reachability analysis or statistical simulations.

2. Proposed Verification Framework

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 $\boldsymbol{\mathcal{M}} \,||\, \mathbb{E} \vDash \boldsymbol{P}$, 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, 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 [6] 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 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 $\mathcal{M}_C$.

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*.

2.2. End-to-End Verification

We first define the latency for tasks and the system modes for mixed-criticality scheduling.

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 (a timer or some signal) and releases some other task. The released task may cause other tasks to release, until some final task completes, 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.*

Definition 3. (System Modes) *The scheduler must expose configuration parameters that change the achievable deadline for a specific set of tasks or task chain. A set of parameters is referred to as a mode. For a dual-critical system, the scheduler may have the option only to run the driving task chain and prevent all other tasks from running. We could refer to this as a* high *mode. We can refer to the scheduler's normal behavior as a* low *mode.*

Using formal methods, we can determine the control system's worst-case latency and the proper system mode for acceptable latency 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 response times of the control chain. The environment model describes where the

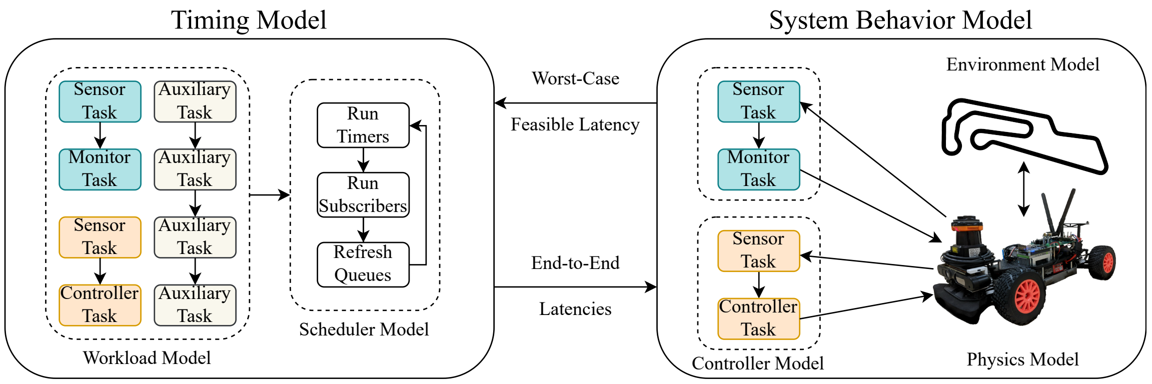

Figure 2. End-to-end verification framework. The timing model (right side) uses the system workload, scheduler policy, and outputs from the monitoring task to determine the end-to-end latency of the system task chains, given the current scheduler mode. The controller model chains run according to the latencies calculated by the scheduler model, and read values from the physics model and environment value. The controller tasks provide control inputs by writing to the physics model, moving it through the environment. The monitoring task reads the physics and environment state, sending latency requirements to the scheduling model. The scheduling model uses the latency requirements to make mode switches, which influence the determined end-to-end latencies.

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. The physical system model should be derived from system dynamics [7].

- **Step 2:** Divide the environment states into groups of similar states. For each state, use the controller model to determine the maximum allowable latency for the controller to succeed in each state.
- **Step 3:** The user must determine what strategies are available to the scheduler to reduce the controller latencies, and define a set of modes with the goal of reducing controller latency. The user should also provide possible overhead times for transitions between modes, if necessary.
- **Step 4 :** In the controller model, create a new task which we call the monitoring task, that uses the environment and physics state to make mode switching decisions. The monitoring task should use a sliding window method to compute actions and states that the physics model will reach in future states. The task checks whether the frontier state enters a group with a different maximum-required latency - if so, it triggers a mode switch. The monitoring task selects the scheduler mode with the highest controller latency that still satisfies the environment requirements. The sliding window ensures that by the time the physical system enters a state with a lower latency requirement, the mode switch has occurred in time.
- **Step 5:** Verify all of the models together for safety. During verification, the physics and environment models are simulated over time, while the scheduler model selects tasks to run from the workload, including the controller and monitoring tasks. The controller causes the physics model to move through the environment, and the monitoring task influences the scheduler behavior through mode switches. If the verifier determines that the physical system never enters a failure state, then the controller, scheduler, and workload is safe to use.

3. Discussion

This section discusses the design choices for the proposed framework and the trade-offs between them.

Offline vs. Online Verification. Depending on the availability of the environment model, the latency requirement of different environment states can be determined using online or offline methods. If the full environment is known ahead of time, possible environment states can be sampled and tested for the required response time for safe behavior. For example, suppose the environment is a race track, and the physical system is an autonomous vehicle. In that case, the track can be split into zones, and the required response time of each zone can be evaluated offline. If the environment is unknown, then the required response time can be determined by the system at runtime. For example, a vehicle can determine a required response time using speed measurements and sensor data of nearby obstacles. For both online and offline response time decisions, some work has to be done online to either look up or calculate a response time.

Lookahead Window for Environment Monitoring. To determine a required response time for safe behavior, the system may perform some ‘look-ahead’ to determine how and where the system will be in the future, depending on the current system latency. The system will use a sliding window, where the system is simulated up to a time horizon. At each time step, the first control input from the sliding window is used in the physical system, the sliding window is incremented forwards in time, and the control input for the new frontier is calculated. If a crash is detected at the frontier, then a mode switch is initiated.

For an offline method, the required response times for environment states is pre-computed. If the frontier state in the sliding window enters a state with a different response time requirement, a mode switch is initiated.

The length of the lookahead window must be determined using the time required to make a mode switch and the current system response time. The mode switch must complete before the physical system enters the environment state with the new latency requirement.

Latency Controlling Mechanisms. The response time of the control system is determined by multiple factors, including the scheduler policy, control task priorities and the presence of other tasks in the system. A mode switch may reduce the system response time by changing priorities, dropping other tasks, or increasing the clock speeds of the control hardware.

Jobs may have priorities that are used to determine, out of all pending jobs, which are run first by the scheduler. If the priorities are dynamically determined from deadlines, then the deadlines of the controller chain could be reduced to increase their relative priority. It may also be possible to temporarily prevent jobs of some tasks from running at all, which may reduce the latency of the remaining tasks.

Depending on the scheduler policy, multiple instances of a chain may be running at the same time, allowing a prior instance of a chain to block execution of a subsequent instance. If the scheduler supports a limit for maximum concurrent chain instances, lowering the limit during a mode switch may reduce the latency of the controller chain.

Processors may run at a speed lower than their maximum possible speed to reduce power consumption and thermal load. During periods where a lower response time is required, the processor speed could be temporarily increased to meet the latency objective.

Mode Switch Overhead. Making changes within the scheduler during a mode switch to meet a latency requirement is additional overhead that must be considered when determining the need for a mode switch.

4. Proof of Concept Evaluation

UPPAAL [8] is a model-checking tool that can *symbolically* verify properties of timed autonoma and *statistically* check properties of hybrid systems. We use UPPAAL to model, as a proof of concept, a single instance of offline latency determination for a racetrack, vehicle, and controller. We take the single-track vehicle model from [9] and specify it using the support for ordinary differential equations provided by UPPAAL's SMC extension [10]. The vehicle is driven using a pure-pursit line-following controller [11]. The controller has some execution time and latency, which contribute to its response time, and is applied to sensing and control inputs. We find that for each track, there is a response time such that, if the controller always responds within that limit, the vehicle will not crash. We consider this the track's *maximum allowable* response time. If the controller is run with a response time just above the maximum-allowable response time, the vehicle consistently crashes at specific points. We consider areas around these points to be areas that require the maximum-allowable response time.

Whenever the vehicle moves too far from the centerline, indicating that a crash is likely, we reduce the response

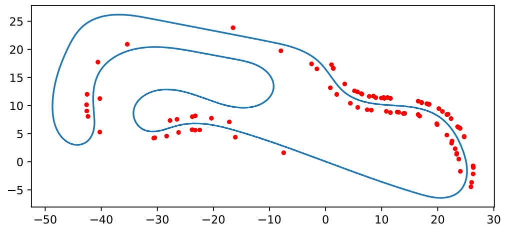

Figure 3. Red dots are crash locations on one map, without any mode switches. These are locations where using a lower latency scheduling mode may allow the vehicle to safely navigate the track.

time to the maximum-allowable response time, allowing the car to successfully navigate the track. This demonstrates that it is possible to use a greater response time than what would normally be required to complete the track, and that temporarily reducing the response time to the track's worst-case response requirement is sufficient.

For the map in Fig. 3, the controller chain has a period of 50 ms and an execution time of 40 ms. Due to blocking, the controller chain has a maximum response time of 130 ms, which causes the vehicle to crash when at high speeds. To prevent crashing, we reduce the response time to 91 ms whenever the vehicle strays too far from the centerline and keep it reduced for at least 0.25 ms until the vehicle returns to a safe distance. The vehicle does not need to spend a long time in the reduced-latency state: on average, it only spends 1.5 s in the reduced latency state in each 100 s run.

This proof-of-concept shows a system with high controller latency that still performs safely in difficult situations by performing a scheduler mode switch to temporarily reduce the controller latency. The problem of determining the latency requirement for environment states remains, but we have preliminary work that determines the maximum allowable latency for a whole environment. We expect to improve this to determine latency requirements for different states within an environment, and to use these results during the verification process.

The goal of this research is to develop a verification approach that can automatically determine the maximum latency requirement for different states within an environment, and, using the execution times of the controller and system tasks, scheduler, and physics models, verify that switching to the required execution times during controller executions is sufficient for the controller to work safely in an environment. This would allow a more flexible software controller design.

5. Conclusions and Future Work

We propose an end-to-end verification framework that collaboratively verifies computational and physical models. To improve system performance, we propose the design of a mixed-critical system where the source of criticality is the variation of environmental states. We provide a proof-of-concept of a system where end-to-end latencies can change in response to the system state. We plan to implement end-to-end verification as a case study on a similar system. In future work, we will develop rigorous correctness proofs for verification and mixed-criticality scheduling policies.

Acknowledgments

This work was supported in part by the National Science Foundation under Grants CCF 2124205, CMMI 224667, 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] 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.

[3] 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.

[4] 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.

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

[6] 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.

[7] 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.

[8] G. Behrmann, A. David, and K. G. Larsen, "A tutorial on Uppaal," *Formal Methods for the Design of Real-Time Systems*, pp. 200–236, 2004.

[9] 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.

[10] 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.

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