Plan B: Design Methodology for Cyber-Physical Systems Robust to Timing Failures

MOHAMMAD KHAYATIAN, San Jose State University
MOHAMMADREZA MEHRABIAN, University of the Pacific
EDWARD ANDERT, Arizona State University
REESE GRIMSLEY, KYLE LIANG, YI HU, IAN MCCORMACK, CARLEE JOE-WONG, and JONATHAN ALDRICH, Carnegie Mellon University
BOB IANNUCCI, Google Inc.
AVIRAL SHRIVASTAVA, Arizona State University

Many Cyber-Physical Systems (CPS) have timing constraints that must be met by the cyber components (software and the network) to ensure safety. It is a tedious job to check if a CPS meets its timing requirement especially when it is distributed and the software and/or the underlying computing platforms are complex. Furthermore, the system design is brittle since a timing failure can still happen (e.g., network failure, soft error bit flip). In this article, we propose a new design methodology called *Plan B* where timing constraints of the CPS are monitored at runtime, and a proper backup routine is executed when a timing failure happens to ensure safety. We provide a model on how to express the desired timing behavior using a set of timing constructs in a C/C++ code and how to efficiently monitor them at the runtime. We showcase the effectiveness of our approach by conducting experiments on three case studies: (1) the full software stack for autonomous driving (Apollo), (2) a multi-agent system with 1/10th-scale model robots, and (3) a quadrotor for search and rescue application. We show that the system remains safe and stable even when intentional faults are injected to cause a timing failure. We also demonstrate that the system can achieve graceful degradation when a less extreme timing failure happens.

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems;**

Additional Key Words and Phrases: Cyber-physical systems, time-sensitive systems, worst-case execution time

This work was partially supported by funding from NIST Award 70NANB19H144, and by National Science Foundation grants CNS 1525855, CPS 1645578, and CPS 1646235.

Authors' addresses: M. Khayatian, San Jose State University, San Jose, 660 S Mill Ave, Tempe, AZ, 85281; email: Mohammad.Khayatian@sjsu.edu; M. Mehrabian, University of the Pacific, Stockton, CA; email: mmehrabian@pacific.edu; E. Andert and A. Shrivastava, Arizona State University, Tempe, AZ; emails: {eandert, aviral.shrivastava}@asu.edu; R. Grimsley, K. Liang, Y. Hu, I. McCormack, C. Joe-Wong, and J. Aldrich, Carnegie Mellon University, Pittsburgh, PA; emails: reese.grimsley@sv.cmu.edu, {kmliang, yihu, icmccorm, cjoewong}@andrew.cmu.edu, jonathan.aldrich@cs.cmu.edu; B. Iannucci, Google Inc., Mountain View, CA; email: bob.iannucci@west.cmu.edu.

2378-962X/2022/09-ART21 $15.00
https://doi.org/10.1145/3516449

ACM Reference format:
Mohammad Khayatian, Mohammadreza Mehrabian, Edward Andert, Reese Grimsley, Kyle Liang, Yi Hu, Ian McCormack, Carlee Joe-Wong, Jonathan Aldrich, Bob Iannucci, and Aviral Shrivastava. 2022. Plan B: Design Methodology for Cyber-Physical Systems Robust to Timing Failures. *ACM Trans. Cyber-Phys. Syst.* 6, 3, Article 21 (September 2022), 39 pages.
https://doi.org/10.1145/3516449

1 INTRODUCTION

Cyber-Physical Systems (CPS) are commonly referred to as the integration of software components (cyber) interacting with physical processes [1]. Pacemakers, drones, **Autonomous Vehicles (AVs)**, and smart cities are a few examples of CPS ranging from small to very large. There are many CPS that are time sensitive, where it is important for the software to generate not only the "right values" but also "at the right time" since a late response from the software may be as fatal as a wrong output [2].

To simplify the design of time-sensitive CPS, one of the earliest steps is to analyze the behavior of the whole system. To do so, a model is considered for the physical process (e.g., differential equations) and the software components (e.g., finite state machine) of the CPS, then a set of timing constraints are determined that must be satisfied by the implementation [3]. For example, from early analysis of a vehicle's kinematics, the delay from detecting an obstacle to applying the brake should be no more than a deadline, say 500 ms, to operate an AV safely at a certain speed. Once timing constraints are set, it is then the software engineers' job to select a proper platform and develop the software such that the timing constraints are always met. This assumption—that the timing constraint will be always met—helps the CPS developers, as they only need to think about system functionality. They do not have to think about what happens when a timing constraint is not met. A few unexpected reasons for timing failure include network delay, a bug in the code, soft error [4], and aging. This article takes the position that such a hard abstraction (assuming that all timing constraints will be met) is not effective anymore and leads to inflexible designs.

A number of CPS design methodologies like PTIDES [5] and Giotto [6] exist that take the CPS system specification and design the CPS that is guaranteed to meet its timing constraints. Such approaches, however, require the calculation and use of **Worst-Case Execution Time (WCET)** of program routines/tasks [7]. Although it was possible to estimate the WCET for small pieces of software that are executed on simple hardware, today both the size of the CPS software and the complexity of CPS hardware have increased dramatically. For instance, the Waymo AV's software comprises more than 100 million lines of code [8], and the platform consists of a multi-core CPU and a GPU with multiple cache levels. The WCET estimate for such large applications that run on such complex hardware will either be impractically pessimistic or unsafe [9]. In addition, any modification to the software/hardware invalidates previous timing analysis, which makes the system design reiteration a tedious job.

To address this issue, we take the stance that timing constraints may fail, and a scalable way to design time-sensitive CPS is to urge programmers to not only specify what happens when a timing constraint is met but also what happens when it is not met.

In this article, we make three contributions:

- We introduce a design methodology called *Plan B* to develop time-sensitive CPS resilient to timing failures. With proper backup routines, a safe design can be achieved by just knowing the WCET of backup routines instead of estimating the WCET for the whole software. In addition, more flexible designs can be developed that meet their timing requirements "most of the time,"and when a timing requirement fails—which happens rarely—the system remains

safe through the execution of a fail-safe backup routine. In addition, the developed system can gracefully degrade and operate at lower rates instead of completely shutting down the system when timing violations are tolerable. This feature enables designers to renegotiate timing contracts at runtime to tune the performance of the system.

- We propose a set of timing APIs for C++ language that can be used by programmers to express the desired timing behavior of the application in the code. Using our API, we urge programmers to envision fail-safe backup routines as a part of the program, which will be executed when timing failures happen. The proposed API also includes specifying timing constraints among nodes of a distributed system.
- We also propose an efficient monitoring mechanism to check the timing requirements of the system at the runtime. The proposed "virtual timer" can be employed when the selected platform has a limited number of hardware timers. Furthermore, we study the implementation of distributed timing constraints and practical considerations like time synchronization.

We apply Plan B's methodology to three case studies: (1) a complex application, the software stack of Apollo [10], which is an open source platform for autonomous driving (tested in a co-simulation with the LGSVL simulator); (2) a distributed system, an automated intersection system with 1/10th-scale model miniature AVs where a set of distributed timing constraints must be met to ensure the safety of AVs; and (3) a quadcopter that should enter a house through a window for disaster response (simulated in Matlab). We showcased that CPS can be designed more reliably where there is no need to accurately estimate the WCET, and our approach can achieve both safety and higher performance compared to static and measurement-based values. First and foremost, we showed that the pessimism in the design of safety-critical CPS can be reduced while safety guarantees are provided. For example, in one of the case studies, we were able to achieve 1.83× higher performance compared to the case where WCET is determined pessimistically, at the cost of infrequent execution of backup routines (0.1% of the time). Results from our experiments show the resiliency of our approach against timing failures with the help of backup routines. To show the resiliency of our approach to timing failure, we intentionally injected faults to cause a timing violation. Compared to the conventional approaches—where a backup routine is not envisioned—the plan B approach was able to avoid accident/instability. We also show the flexibility of our approach to achieve graceful degradation. Results show that the system can operate at a lower performance when an intermittent timing failure happens instead of completely shutting down the system.

2 RELATED WORKS

Researchers have broadly studied WCET estimation in the literature [7, 11–15]. There are two main approaches to estimate the WCET of the software: static timing analysis and measurement-based approaches. Static methods estimate the WCET based on the structural information of the software. In a search process, the path in the software that corresponds to the longest execution time is sought [16] and the WCET is calculated accordingly. Since caches have the most contribution in WCET computation, a cache model is considered to have more accurate estimation [17]. However, due to the complexity of the actual hardware (many levels of caching, out-of-order execution, etc.) and software (many lines of code, OS calls, etc.), timing models are simplified, which result in inaccurate WCET estimation. Although static methods may be able to find a safe bound on the execution time of a program, they usually overestimate the WCET and therefore provide pessimistic WCET values.

In measurement-based WCET approaches [12, 18], the program is executed for different sets of input and its execution time is measured. Then, the longest execution time is considered as the WCET of the program. To be confident, a 20% safety margin is added to the longest observed value to account for cases that are not covered. Researchers have also developed probabilistic WCET

(pWCET) approaches to estimate the WCET [19–21] where a probability distribution function is fit to the measured execution times and the WCET is provided with a confidence value (e.g., 99.9%). Measurement-based techniques and the probabilistic approach, however, underestimate the WCET, and the calculated WCET using these methods is not safe. This is because the code and the system cannot be tested for all possible inputs and states. To help researchers estimate the WCET, many WCET analysis tools are developed. aiT [22], Bound-T [23], and RapiTime [24] are a few examples of such tools. Uppaal [25] is a model checker tool that supports WCET analysis.

Assuming the WCET is known for a number of programs that run on the same platform, the **Worst-Case Response Time (WCRT)** [13] of a task is calculated by determining how many times it may be preempted by other high-priority tasks in the worst case. WCET and WCRT are used to perform a schedulability analysis. Based on the data dependency between tasks, end-to-end analysis is done to estimate the **Worst-Case End-to-End Delay (WCE2ED)** [26, 27].

Simplex architecture [28] uses a similar concept to design safety-critical systems. In the Simplex approach, a simpler subsystem is designed to take over when a fault (timing error, OS crash, no output, voltage fault, etc.) happens. Compared to Simplex, Plan B is mainly focused on timing violations and supports distributed systems.

3 BACKUP ROUTINE BASED EXECUTION

In this section, we first present the key idea for the design methodology of the Plan B approach and then present a motivating example to benefit from backup routines for graceful degradation.

3.1 Key Idea of Backup Routine Based Execution of Time-Sensitive Applications

We developed the Plan B framework based on the premise that a timing constraint may fail (despite WCET analysis) and the designer should envision what should happen when a timing violation occurs. In Plan B, a monitoring mechanism is developed to check if timing constraints are met at runtime and, if not, execute a backup routine. By timely execution of a proper backup routine, the safety of the system could be guaranteed. Since designing a proper backup routine depends on the model of the system, backup routines vary from one application to another.

In many time-sensitive applications, the software is very large and the hardware is heterogeneous and has a complex architecture, which makes WCET estimation a difficult process. Unlike existing methods that try to estimate the WCET using static and measurement-based methods, the Plan B approach does not require estimating the WCET because the execution of the software can be bounded by an upper bound. Typically, the distribution of the execution time is similar to the histogram depicted in Figure 1. The histogram shown later in Figure 10 belongs to an AV. An important timing constraint for an AV is that "the delay from sensing to actuation should be less than a threshold." The AV should be able to detect an obstacle and slow down when needed in a timely manner. Let us assume that the *actual* WCET of the whole software is 650 ms, and the estimated WCET by a measurement-based approach and a static approach is 450 ms and 1,800 ms, respectively. If the system is designed based on the static WCET (1,800 ms), the vehicle should drive very slowly to ensure it does not hit an obstacle, which is very conservative. If the system is designed based on the measurement-based WCET (450 ms), we cannot guarantee its safety since the AV's execution time can exceed 450 ms and it hits an obstacle.

As the first step in Plan B's design methodology, we need to find a reasonable upper bound for the execution time. A practical way that is used is to consider the measurement-based WCET plus 10% buffer as the upper bound. Let us assume the upper bound is considered to be 500 ms. The next step is to envision a backup routine such that if the backup routine is executed in a timely manner, the AV remains in a safe state. A simple backup routine for this application is to apply the full brake. We set the deadline for delivery of the backup routine to be 500 ms. Since this backup routine is

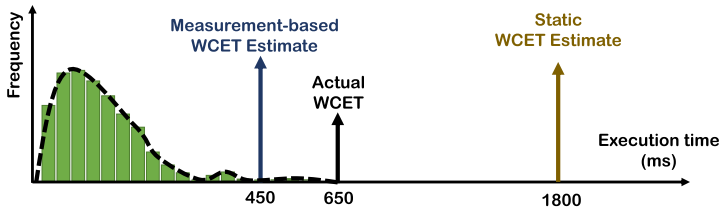

Fig. 1. Histogram of the execution time of the AV software and estimated WCET using static and measurement-based approaches.

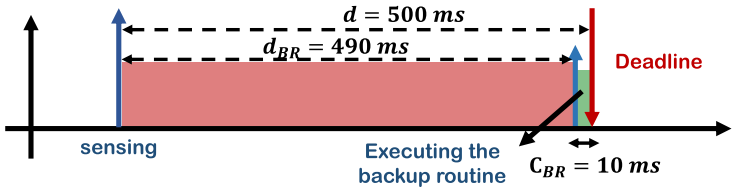

Fig. 2. By on-time execution of a safe backup routine, the vehicle remains in a safe state.

very simple and comprises only a few lines of code, it is easy to accurately estimate its WCET. For this example, we assume the WCET of the backup routine (C_{BR}) to be 10 ms. If the backup routine is executed within 500−10 = 490 ms of the sensing, even in the worst-case scenario, the brake signal is delivered within the set deadline (500 ms). Based on this design, it is guaranteed that the AV never hits its front vehicle and will stop safely even if a timing failure happens. However, the AV's behavior is more conservative since it stops whenever there is a timing failure regardless of the presence or absence of an obstacle. Figure 2 shows an overview of Plan B's execution model for our AV example.

3.2 Graceful Degradation Using Backup Routine Based Execution

The Plan B approach allows for bounding the execution time of a program by defining an alternative path (backup routine), which enables a number of capabilities. First and foremost, the system will be more resilient because even if a timing failure happens—due to an unexpected issue such as aging, soft error, or bug in the code—the system remains safe. Second, the design can become flexible to timing failure. Instead of having timing constraints with fixed deadlines, the deadlines can be adaptively set depending on the state of the system. Normally, the deadlines for a system are set while accounting for the worst-case scenario, such as the maximum braking distance of the vehicle, which corresponds to the maximum velocity of the vehicle. However, when driving at a lower velocity, the timing constraints can be relaxed. In a flexible design, when a timing constraint is not met, the system can be reconfigured to operate at a lower rate (e.g., drive at a slower speed) and gracefully degrade instead of a harsh action (e.g., applying the full brake or completely terminating the operation). Recalling the AV example, one can derive a relationship between the deadline for executing the backup routine and the maximum speed at which the AV should drive. We assume the AV is following the 2-second rule and its the dynamics are modeled using the following equations:

$$\begin{cases} \dot{p} = v \\ \dot{v} = a \\ a(t) = u(t-\rho) \\ u(t) = K(p_f(t) + 2 * v + c - p), \end{cases} , \tag{1}$$

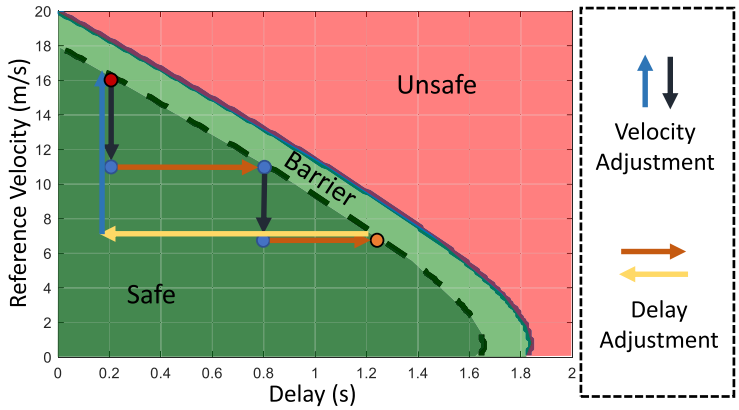

Fig. 3. Relationship between the velocity of the AV and response time of the software. When a fault happens, the backup routine is executed periodically (every 200 ms), and each time the reference velocity is reduced by 70%.

```
Backup_Routine(){
    if (v_prev > 4)
        v_r = v_prev * 0.7;
    else
        v_r = 0;
    TC.deadline = lookupTable(v_r);
}
```

Listing 1. A flexible backup routine.

where p is the longitudinal position of the AV, v is the velocity, c is a constant safety barrier, a is the acceleration, u is the control input (applied as throttle or brake), t is time, ρ is the response time of the software (from sensing to actuation), and p_f is the longitudinal position of the front vehicle. The control input is designed such that the AV maintains a distance equal to 2 seconds from its front vehicle plus some constant c. Since the acceleration rate and velocity of a vehicle are bounded, we consider a bound on the acceleration $a \in [a_{min}, a_{max}]$ and velocity $v \in [0, v_{max}]$. Considering the delay-free model ($\rho = 0$), the stop distance of the AV is calculated as $d_{stop} = \frac{v^2}{2|a_{min}|}$. If the response is delayed, the traveled distance will be $d_{stop} = \frac{v^2}{2|a_{min}|} + \rho v + 0.5 a_{max} \rho^2$ in the worst case as the AV may be accelerating while processing the sensed data. Substituting the maintained distance $2 * v + c$, we can find a relationship between ρ and v:

$$2v + c \geq \frac{v^2}{2|a_{min}|} + \rho v + 0.5 a_{max} \rho^2. \tag{2}$$

For this example, we use following parameters: $a_{min} = -4.5$, $a_{max} = 2.6$, and $c = 4.5$. Figure 3 shows the relationship between the AV's velocity and the response time of the autonomous driving software. The area highlighted as "unsafe" means the AV is in an unsafe state and may hit its front vehicle if the front vehicle stops suddenly, whereas the green area indicates that the AV can apply the brake in time and it is physically impossible to hit its front vehicle even if the front vehicle stops suddenly. Let us assume the AV is initially set to drive at 16 m/s (about 30 mph). A possible backup routine to achieve graceful degradation is to reduce the velocity by 30% and readjust the deadline (see Listing 1).

The initial deadline is set to 200 ms. When a timing failure happens, the reference velocity is set to 16*0.7 = 11.2 m/s. We assume that the response time of the controller is fast, and for simplicity, the AV slows down at -4.5 m/s^2. At 11.2 m/s, the AV is able to tolerate a delay of up to 0.71 seconds and still be safe. The new deadline (0.71 seconds) is computed from a lookup table that represents the relationship between velocity and deadline as indicated in Equation (2). If another timing failure happens (the deadline is greater than 0.71 seconds), the backup routine is executed again. For all velocities greater than 4 m/s, the backup routine ensures that the AV remains in the green zone by reducing the velocity by 30%. If another timing failure happens, the backup routine is executed again and the deadline value is updated. Similarly, when the timing goes back to normal (delay is 0.2 seconds), the system can recover and operate at the nominal velocity (16 m/s).

3.3 System Model

Let us assume that our application has n timing constraints. Without loss of generality, all timing constraints can be simplified to an end-to-end latency constraint. For instance, the timing constraint "the period of executing a function should be 100 ms" can be rewritten as "the end-to-end latency between two function calls should be 100 ms." We assume that for each timing constraint, a safe backup routine exists such that if the backup routine is executed within the timing constraint's deadline, the system remains in a safe state. We represent timing constraints of an application as a set,

$$TC = \{TC_1, TC_2, \ldots, TC_n\}, \tag{3}$$

where each timing constraint is a tuple, $TC_i :=< C_i, T_i >$, where C_i is the WCET of the backup routine i and T_i the execution period of the backup routine i. The execution period of a backup routine, T_i, depends on how frequently the timing constraint is assigned/specified, T_{TC}. For timing constraints that are assigned aperiodically, we consider a lower bound on T_i. Since multiple timing constraints may be specified, there will be multiple backup routines associated with them. To make the execution deterministic, a priority value is assigned to each backup routine by the programmer. Without loss of generality, we assume that the priority of the backup routine i is i and a lower number indicates a higher priority. We do not explicitly model the task model for the normal execution of the software since the priority of all backup routines is higher than tasks for normal execution and the safety-related timing requirements are defined for backup routine and not normal routines.

Assume that dynamics of the CPS are modeled as follows:

$$\begin{cases} \dot{x} = f(x(t), u(t - T(t))) \\ y = g(x(t)) \end{cases}, \tag{4}$$

where $x \subseteq \mathcal{R}^n$ represents the vector of the system state, u is the vector control inputs, y is the vector of measured states, and T is the sensing to actuation delay related to the software's execution time. It should be noted that T is not constant and depends on execution of the software. When a timing failure happens and a backup routine is executed, the control inputs are changed. Therefore, we use hybrid automata [29] to model the complete behavior of the system. Figure 4 shows the overview of the complete system model. In this model, the software is represented using two discrete states (normal mode and backup mode). The initial state and input are specified with x_0, u_0. Initially, the system operates in normal mode. When the execution time or sensing to actuation delay ($T(t)$) is greater than the set deadline, the state machine transitions to the backup mode where the backup routine ($u*$) is applied to the system.

Definition 1 (Safe Backup Routine). Given x_0 is the initial state of the system at time t_0, and $\mathcal{X}_u$ is the set of the unsafe states, the backup routine u^* is a called safe if by applying u^* to the system, states of the system never reach the unsafe set (i.e., $x(t) \notin \mathcal{X}_u$).

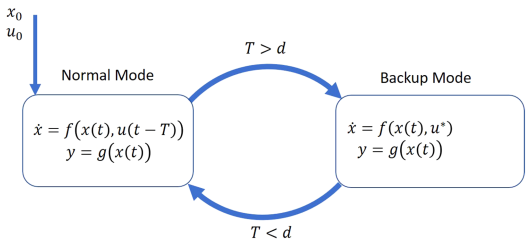

Fig. 4. The complete system (physical dynamics and software abstraction) is modeled using a hybrid automaton to verify if the unsafe set is reached.

One can use Control Barrier Function (CBF) [30–32] to determine control inputs (backup routine) that are proved to be safe. Alternatively, verification tools can be utilized to check if a backup routine is safe. In existing verification tools [33, 34], an assertion is specified that describes the unsafe set, x_u.

For simplicity, we set the deadline for the execution of the backup routine to be the same as the deadline for the original timing constraint (normal mode). Since the execution of the backup routine itself takes some time, the backup routine should be executed a bit earlier. Assume a timing constraint (indexed as i) is specified at time $t = 0$, and its deadline is at $t = d_i$, the activation time of the backup routine (d^i_{BR}) can be calculated as follows:

$$d^i_{BR} = d_i - C_i.$$

In most cases, backup routines are very small (have no more than few lines of code). As a result, the WCET of the backup routine can be estimated using static approaches, which results in much less pessimistic values compared to the case where static analysis is done for the whole program.

3.4 Safety Proof of a Proposed Backup Routine

Our approach is useful only if the proposed backup routine is proved to be safe when applied in time. We can develop the safety proof in different ways, depending on the system model. For a system with a deterministic model and input, one can simulate the system and check if the backup routine is safe (e.g., using the Matlab Stateflow tool). For a system with a non-deterministic set of initial state and input, verification tools like Flow* [33] can be employed. Here, we provide the proof for the aforementioned example in Section 3.2 by simulation. We first define the initial set $X_0 = [0\ 16\ 36.5\ 0]$ meaning that the ego vehicle is at $p = 0$ driving at 16 m/s and its front vehicle is $2v + c = 36.5$m away and is already stopped. The unsafe set is defined as $X_u = \{X | dist(X(1), X(3)) < 0.5\}$, representing cases where the distance between vehicles is less than 0.5 m. We want to show that even if the sensing to actuation delay is infinite, the distance between vehicles remains greater than 0.5 m. The ego AV continues driving at 16 m/s for 0.2 seconds (the first deadline), and after that, it reduces its velocity to 12.8 m/s. Since the maximum deceleration is -4.5 m/s^2, it takes about 0.7 seconds to reach 12.8. Since the second deadline for the velocity at 12.8 m/s is 0.6, the backup routine is executed one more time and the velocity of the ego AV is set to 10.2 m/s. The left side of Figure 5 shows the position, velocity, and set deadline for the vehicle when a safe backup routine (slow down by 70%) is applied. The ego AV does not hit its front vehicle located at 36.5 since the final position value is 36. The set delay and reference velocity are shown in Figure 5. Note that for a similar backup routine that reduces the velocity by 80%, a safety guarantee cannot be achieved. The right side of Figure 5 shows the position, velocity, and set deadline for the vehicle when an unsafe backup routine (slow down by 80%) is applied.

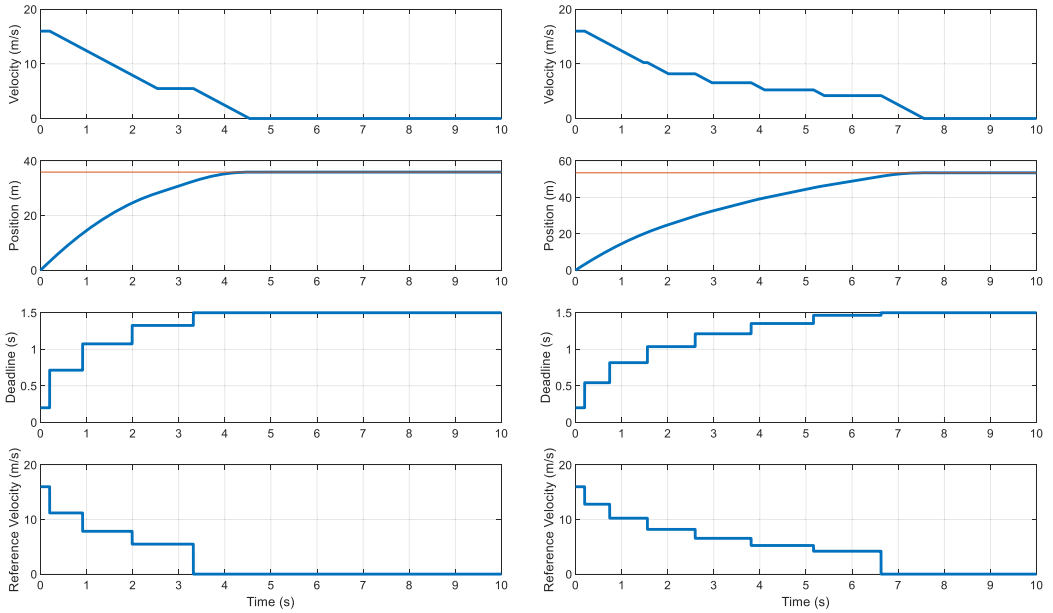

Fig. 5. The trace for the worst-case scenario where consecutive timing failures happen. Left: The backup routine (slow down by 70%) is safe, and the AV is able to stop safely. Right: The backup routine (slow down by 80%) is unsafe, and the AV is unable to stop safely.

3.5 Handling Multiple Backup Routines

Although rare, there can be cases where multiple timing constraints are specified and they are violated at the same time or the execution of their backup routine has an overlap. In such cases, low-priority backup routines are blocked by high-priority ones, and therefore their finish time will be late. As a result, we need to calculate the WCRT for each backup routine and use it for computing the firing time of the backup routine. Figure 6 shows a scenario where three timing constraints are violated at the same time.

To make sure all backup routines will finish their execution before their deadline, we use WCRT (w) instead of WCET to determine the activation time of the backup routine:

$$d_{BR} = d - w.$$

Assuming fixed-priority scheduling for the execution of backup routines, one can calculate the WCRT of the backup routine i using conventional approaches [13] as follows:

$$w_i = c_i + \sum_{j \in hp(i)} \left\lceil \frac{w_i}{T_j} \right\rceil c_j.$$

$hp(i)$ represents the set of all backup routines that have a higher priority than backup routine i and are expected to be executed on the same machine as backup routine i.

3.6 False Positives

It should be noted that there can be cases where the execution of the original code is between 490 and 500, which is fine and there is no need for the execution of the backup routine but the backup routine is executed. Measurement-based WCET analysis is helpful to determine a reasonable upper bound for the execution time of the software (e.g., 500 ms in this example) that will be used to ensure that the timing requirement will be mostly met (e.g., 99.9% of the time), and when it is not

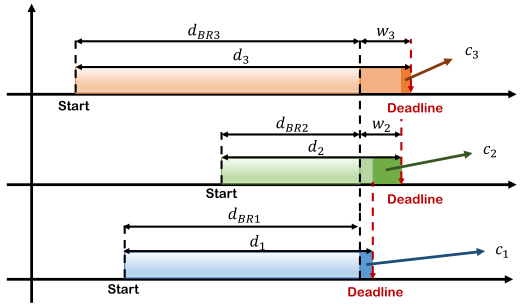

Fig. 6. A scenario where three timing constraints are violated at the same time and the execution of backup routines overlaps.

met (e.g., 0.1% of the time), the backup routine is executed. By correctly adjusting the upper bound, the rate of false positives can be reduced.

The rate of "false positives"—when the execution time of the software falls between d_{BR} and d—is usually low since the size of backup routines is small relative to the whole program. However, the WCRT of low-priority backup routines increases with defining more timing constraints, and the rate of false positives increases as well. First of all, this issue happens rarely. If the rate of a timing failure and execution of a backup routine is low, simultaneous timing failures are rare.

3.7 Handling Distributed Timing Constraints

Plan B allows for building distributed systems with end-to-end timing constraints among nodes. In the general form, two computing nodes collaborate to monitor a distributed timing constraint. Since this collaboration happens by means of communicating over the network, long network delays can disrupt on-time monitoring of an end-to-end timing constraint. To tackle this issue, we convert all aperiodic end-to-end latency constraints into periodic ones where the period is set to be the same as the threshold for the end-to-end latency constraint.

Assume a distributed latency between events e1 and e2, specified on nodes N1 and N2. The monitoring mechanism on the first node (N1) sends the deadline timestamp to the second node (N2) in a periodic manner. On the receiver side, the monitoring mechanism receives the deadline timestamp and locally monitors the timing constraint. If the deadline timestamp is not received within the deadline (d'), the backup routine will be executed.

In Plan B's approach for distributed timing constraints, the first node computes a deadline and converts it to the UTC format and the destination converts it back to a local deadline. As a result, nodes should synchronize their local clocks periodically so that they have the same notion of time. The period of the clock synchronization depends on the frequency variations of the node's clock (oscillator).

Let us consider a platooning scenario where multiple AVs drive together to get better fuel efficiency by reducing air resistance. The front AV sends the intended action (slow down or speed up) and desired velocity to its rear AVs, and they adjust the velocity accordingly. To ensure the safety of AVs, the latency from the front AV sending its information to the rear AVs should be less than a threshold. The backup routine for each AV is to slow down periodically until the timing constraint is met similar to Listing 1. The monitoring mechanism on each AV sets a timeout for receiving the deadline timestamp.

In the next section, we explain how programmers can specify timing constraints and corresponding backup routines in the code and how it is implemented such that on-time delivery of backup routines is guaranteed.

4 PROPOSED TIMING API FOR C/C++ LANGUAGE

In this section, we introduce Plan B's timing constructs that are developed for the C/C++ language. In our approach, timing constraints are defined as *exceptions* that can be caught by a runtime monitoring mechanism. A set of *timing constructs* is provided to specify a timing constraint as a part of the program. In our approach, timing constraints are specified using events. An event corresponds to a line in the code and is annotated using the `_recordEvent(eventName)` construct, where `eventName` indicates the name of the event. Let us consider a simple C program for an embedded control system shown in Listing 2 where the goal is to read data from the sensor, perform some computations on the sensed data, and then actuate based on the result.

```
while(1){
    data = sense();
    result = compute(data);
    actuate(result);
}
```

Listing 2. A sample snippet of code for reading data from sensor performing some computation and then actuating.

We use this example to explain the specification of different timing constraints in the code.

4.1 Latency Constraint

Let us assume that the latency from the sensing to actuation in the example shown in Listing 2 should be less than 100 ms (e.g., to ensure the quality of service). To specify such a timing constraint, programmers can define two events, one before the sensing and one after the actuation, and define a latency constraint as shown in Listing 3. In general, three types of latency constraints can be defined, where the latency among two events should be less than, greater than, or equal to a value. Note that for the equal case, programmers should know how much tolerance is acceptable, because achieving an exact latency is not feasible, and then specify it in terms of two conditions: a greater than and a less than case. Listing 3 shows the modified version of code in Listing 2 where two events (`e1` and `e2`) are annotated and a latency constraint is specified. `p` is the priority for the execution of the corresponding backup routine. The priority value should be known since multiple timing violations can happen simultaneously.

```
while(1){
    data = sense();
    _try{
        _recordEvent(e1);
        result = compute(data);
        _recordEvent(e2);
    }
    _catch(_latency(e1,e2,100) > 100000 , p){
        BR();
    }
    actuate(result);
}
```

Listing 3. Annotating a latency timing constraint inside the code to check if the sensing to actuation latency is less than 100 ms.

Programmers can place a backup routine inside the `catch` segment to specify what happens when the latency timing constraint is not met. The less than case can be defined similarly by modifying the condition for the timing exception.

4.2 Period Constraint

Period constraint specifies that the occurrence period of an event is to be less/greater than or equal to the threshold. Programmers may want to make sure that the period of execution of a task/function is less than, greater than, or equal to a value. Let us assume that we want the executing period of the `compute()` function in the Listing 2 to be equal to 10 ms ± 1 ms. This timing constraint can be specified by defining an event inside the `compute()` function and then specifying two period constraints to specify the desired range as shown in Listing 4.

```
while(1){
    data = sense();
    result = compute(data);
    actuate(result);
}

int compute(int data){
    _try{
        _recordEvent(e3);
        ...
    }
    _catch(_period(e3,100) > 11000, 1){
        BR();
    }
    _catch(_period(e3,100) < 9000, 2){
        BR();
    }
}
```

Listing 4. Timing constructs are added for checking the execution period of the `compute()` function.

Based on the allowed tolerance (1 ms), the desired range for an acceptable period is between 9 and 11 ms. If the period is out of this range, a backup routine (`BR()`) is executed.

4.3 Reusing Events for Multiple Timing Constraints

Sometimes programmers may want to specify multiple timing constraints for a code. This can be done by using multiple `catch` constructs. The code in Listing 5 shows a case where latency and period constraints are specified together. The latency of the `compute` function should be less than 100 ms, and its period should be less than 200 ms. `p1` and `p2` are the priorities for execution of the first and second backup routines, respectively, which are specified by the programmer.

```
while(1){
    data = sense();
    result = compute(data);
    actuate(result);
}

int compute(int data){
    _try{
        _recordEvent(e1);
        ...
        _recordEvent(e2);
    }
    _catch(_latency(e1,e2,100) > 100000, 1){
        BR1();
```

```
15      }
16      _catch(_period(e1,100) > 200000, 2){
17          BR2();
18      }
19  }
```

Listing 5. Specification of multiple timing constraints and reuse of events in two separate timing constructs.

Using a separate `catch` construct for each timing constraint allows programmers to specify a separate backup routine when a timing constraint is not met. In addition, the programmer can reuse an event (e.g., e1 in this example) for specifying different timing constraints.

4.4 Distributed End-to-End Latency

Some CPS are distributed by nature (e.g, drone swarm), and some have a distributed computing platform that includes multiple embedded devices communicating with each other. In such systems, programmers may want to specify end-to-end timing constraints among devices. Let us consider an example of a distributed CPS with two devices, where device 1 collects the data from a sensor, does some preprocessing, and sends the result to device 2. Upon receiving the result, device 2 performs some more processing and then performs an actuation. Note that the communication is synchronous and receive is blocking (not returned until the data is received). For this example, one may define a timing constraint as "The latency from sensing at device 1 to actuation at device 2 should be less than 500 ms." Listings 6 and 7 show the code for device 1 and 2, respectively.

```
1   while(1){
2       _try{
3           _recordEvent(e1);
4           data = sense();
5           result = compute(data);
6           send(result);
7       }
8       _catch(_ackTimeout(e1,100), 1000000, 1){
9           BR1();
10      }
11  }
```

Listing 6. Timing constructs are added for specification of a distributed latency constraint between two points in separate programs: device 1.

```
1   while(1){
2       _try{
3           data = receive();
4           result = compute(data);
5           actuate(result);
6           _recordEvent(e2);
7       }
8       _catch(_distLatency(e1,e2,100) > 500000, 1){
9           BR2();
10      }
11  }
```

Listing 7. Timing constructs are added for specification of a distributed latency constraint between two points in separate programs: device 2.

To implement the monitoring mechanism for an end-to-end timing constraint, device 1 sends the deadline to device 2, and device 2 locally sets up a timer to monitor the timing constraint. Monitoring of a distributed timing constraint can be impacted by network delay, especially if the deadline timestamp is sent over wireless communication. As a result, the timestamp may be delivered to the receiver very late or not being delivered at all. To detect excessive network delays, the receiver is configured to send back an acknowledge (ack) message to the sender upon receiving the deadline timestamp. The sender waits for the ack message to ensure the deadline timestamp was delivered on time. If the ack is not received after a deadline, the specified backup routine will be executed. In addition, clock synchronization and timestamp translation need to be implemented so that devices have the same notion of time because the deadline timestamps are captured by local clocks and will be used by another device. More details are provided in Section 5.

4.5 Distributed Simultaneity

A simultaneity timing constraint can be specified to make sure a set of events happen at the same time—with a small tolerance. Let us assume that the same program as Listing 8 is running on three devices. It is desired to perform sensing simultaneously with 1-ms tolerance. We annotate a simultaneity timing constraint on the event set E1.

```
1  while(1){
2      _try{
3          _recordEvent(E1);
4          data = sense();
5          result = compute(data);
6          sendToServer(result)
7      }
8      _catch(_simultaneity(E1,100), 1){
9          BR1();
10     }
11     _catch(_ackTimeout(E1,100), 2){
12         BR2();
13     }
14 }
```

Listing 8. Timing constructs are added for specification of a simultaneity timing constraint.

To implement this timing constraint, all three devices capture a timestamp before sensing and broadcast it. Upon receiving timestamps of other devices, they verify if the timing constraint is met by checking if the time difference between the earliest and the latest timestamps is less than the specified tolerance (1 ms). This timing constraint also requires maintaining a minimum level of time synchronization and timestamp translation, which is explained in the next section.

We have listed Plan B's timing constructs in Table 1. Note that try constructs cannot be nested and are used to hint the compiler where events are defined. Programmers should define a priority value to specify the execution order of backup routines when multiple timing failures happen. Having fixed priority values makes the execution behavior more deterministic compared to dynamic priority values. The priority value is a number greater than zero. Smaller values correspond to higher priorities, and two timing exceptions cannot have the same priority.

5 PROPOSED IMPLEMENTATION FOR EFFICIENT EXECUTION

In this section, we provide more detail for implementation of the introduced timing constructs. A naive way to monitor a timing constraint is to take timestamps at event annotation locations and perform a simple check on the values of timestamps to see if the timing constraint is met.

Table 1. Plan B Timing Constructs and Their Functionalities

Construct	Functionality
`_try{ ... }`	Indicates the body of the program where events are defined.
`_catch(exception, priority){ ... }`	Indicates the exception that should be caught, the backup routine to be executed, and the execution priority of the backup routine.
`_recordEvent(e)`	Annotate an event in the code labeled as `e`.
`_latency(e1,e2,`ϵ`) ><= d`	Specifies a latency timing constraint between two events `e1` and `e2` (less than, greater than, or equal to d seconds). ϵ defines the acceptable tolerance of the timing constraint.
`_period(e,`ϵ`) ><= T`	The desired occurrence period of the event `e` should be less than, greater than, or equal to T with a tolerance of ϵ.
`_distLatency(e1,e2,`ϵ`) ><= d`	Similar to the local latency constraint but events e1 and e2 are defined in programs running on separate devices.
`_ackTimeout(e1,`ϵ`)`	Specifies a timeout for sending the deadline timestamp of a distributed latency constraint.
`_simultaneity(E,`ϵ`)`	All events in the event set E (event set E is defined on distributed device) should be simultaneous with a tolerance of ϵ.

However, the detection time of the timing failure can be unbounded (e.g., when the program never reaches the location of the second event). Since we are interested in the timely detection of a timing violation, we get help from hardware timers for monitoring. A timer is activated using `_startTimer(TC)` and stopped using `_stopTimer(TC)`. Upon expiration of the timer, `_timerExpired(TC)` is called automatically by the interrupt handler that is attached to the timer. `TC` is a `struct` variable that represents a timing constraint, which includes the deadline for the constraint, the name of the corresponding backup routine, the timing constraint's ID, a timestamp value, and the state of the timing constraint (being active or inactive). Each timing constraint has a tolerance value (ϵ) to indicate how much inaccuracy is acceptable when being monitored since perfect monitoring is not feasible. We consider three error sources for monitoring: monitoring error (e_M), implementation error (e_I), and synchronization error (e_S). Monitoring error depends on the resolution of the captured timestamp, and implementation error depends on the time it takes to set up/reconfigure a timer and the ISR's response time upon expiration of a timer. Synchronization error is considered for distributed timing constraints only (`_distLatency` and `_simultaneity`) and depends on the synchronization level among devices. To verify that the

monitoring system is suitable, the following constraint is checked:

$$e_M + e_I + e_S \leq \epsilon. \tag{5}$$

5.1 Latency Constraint

As discussed in the previous section, latency timing constraints are of two types: less than or greater than (the equal case should be annotated as a less than case and a greater than case). For a greater than timing constraint (e.g., `latency(e1,e2) > 100 ms`), a single-shot timer is started at the annotation location of the first event and it is stopped at the annotation location of the second event. If the timer is expired before reaching the second event, the timing constraint is violated and the specified backup routine should be executed (it will be added to the **Backup Routine Queue (BRQ)** and will be executed). The activation time of the backup routine is set to 99 ms assuming the WCRT of the backup routine BR is 1 ms ($100 - 1 = 99$ ms) according to $TC.activationTime = TC.deadline - TC.WCET$. The code in Listing 9 shows the generated code for the latency example from the previous section (Listing 3).

```
1  TC1.activationTime = 0.099;
2  TC1.priority = 1;
3  jmp_buf buf;
4  actuate_t gResult;
5  while(1){
6      data = sense();
7      if (setjmp(buf)){
8          result = gResult;
9      }else{
10         _startTimer(TC1);
11         result = compute(data);
12         _stopTimer(TC1);
13     }
14     actuate(result);
15 }
16
17 void BR(){
18     gResult = backupRoutine();
19 }
20
21 void _timerExpired(TC1){
22     _BRQManager(BR, TC1.priority);
23     longjmp(buf,1);
24 }
```

Listing 9. Implementation for a maximum end-to-end latency constraint.

When the BRQ is empty and the first BR is added, the BR is being executed as a separate thread using `pthread_create()`. The parent thread (`_BRQManager()`) will wait for the BR to finish its execution (while (!isEmpty(BRQ)). As a result, the control is not given back to the main loop until the BR execution is finished and the gResult value is updated. In our scheme, backup routines are executed based on their priority values and in series (no two backup routines are executed in parallel) to make WCRT calculation of backup routines less conservative. When a high-priority backup routine is added to the BRQ, the BRQ manager places it at the head of the queue, suspends the low priority backup routine by sending a suspend signal to it (`pthread_kill(BR1, signal)`), and executes the high-priority backup routine. In the signal handler of all backup

routines, the thread is either paused (using `pause()`) or resumed depending on the received signal. After executing the high-priority backup routine, the BRQ is updated (removing the high-priority backup routine) and the queue manager sends a resume signal to the low-priority backup routine to resume the execution. The BRQ manager also utilizes a mutex to ensure that two threads or itself (when a backup routine finishes its execution) do not access the queue at the same time. We assume the update time of the queue is negligible and ignore it in the WCRT computation.

If the `compute(data)` method has dynamic memory allocation, more effort is needed to avoid possible memory leak due to jumps. Either the memory leak should be detected manually (e.g., using a flag) and it is deallocated after the jump (after `if setjump(buf)`) or a fixed global memory space is allocated for the compute(data) function and reused. Additionally, non-atomic data structures must be treated as corrupted when the longjmp occurs, as they may be left in an inconsistent state.

For a less than timing constraint (e.g., `latency(e1,e2) < 100 ms`), a timer is started at the annotation location of the first event and the variable `TC.active` is set to true. Upon expiration of the timer, the variable `TC.active` is set to false. We assume that the WCRT of the backup routine is 1 ms. If the program reaches the annotation location of the second event and the `TC.active` variable is still true, the timing constraint is violated and the backup routine is added to the BRQ to be executed. In Listing 10, we show the generated code for the latency example from the previous section (Listing 3) when the timing constraint condition is greater than 100 ms instead of less than 100 ms.

```
1  TC1.activationTime = 0.099;
2  TC1.priority = 1;
3  jmp_buf buf;
4  actuate_t gResult;
5  while(1){
6      if (setjmp(buf)){
7          actuate(gResult);
8      }else{
9          _startTimer(TC1);
10         TC1.active = TRUE;
11         data = sense();
12         result = compute(data);
13         actuate(result);
14         if (TC1.active = TRUE){}
15             _stopTimer(TC1);
16             _BRQManager(BR(), TC1.priority);
17             longjmp(buf,1);
18         }
19     }
20 }
21
22 void _timerExpired(){
23     TC1.active = FALSE;
24 }
25
26 void BR(){
27     gResult = backupRoutine();
28 }
```

Listing 10. Implementation for a minimum end-to-end latency constraint.

5.2 Period Constraint

A period timing constraint can be implemented as a repetitive latency timing constraint. The code in Listing 11 is the generated code for the Listing 4. The `_firstIteration` variable is used to skip the first instantiating of the `stopTimer`, and after that, the timer is used to check the latency between every two consecutive executions of the program. We assume the WCRT of the backup routines is negligible.

```
TC1.activationTime = 0.011;
TC2.activationTime = 0.009;
TC1.priority = 1;
TC2.priority = 2;
jmp_buf buf;
actuate_t gResult;
while(1){
    data = sense();
    result = compute(data);
    actuate(result);
}

_firstIteration = TRUE
int compute(int data){
    if (setjmp(buf)){
        return gResult;
    }else{
        if (_firstIteration == FALSE){
            _stopTimer(TC1);
            if (TC2.active == TRUE){
                _stopTimer(TC2);
                _BRQManager(BR(), TC2.priority);
                longjmp(buf,1);
            }
        }
        _firstIteration = FALSE;
        _startTimer(TC1);
        _startTimer(TC2);
        TC2.active = TRUE;
        ...
        return result;
    }
}

void _timerExpired(TC1){
    _BRQManager(BR(), TC1.priority);
    longjmp(buf,1);
}

void _timerExpired(TC2){
    TC2.active = FALSE;
}

```

```
44  void BR(){
45      gResult = backupRoutine();
46  }
```

Listing 11. Implementation for an equal period constraint.

Two timers are started at the annotation location of the event and stopped in the next iteration. The desired period (10 ms ± 1 ms) is converted into an upper (0.011 ms) and a lower (0.009 ms) bound for the timers. If the period is less than 0.009 ms, TC2 is violated, and if it is greater than 0.011, TC1 is violated. Note that the backup routine for both cases is the same and TC1 and TC2 cannot be violated at the same time.

5.3 End-to-End Distributed Constraint

As mentioned in Section 4.4, implementing a distributed latency constraint is done by capturing a timestamp on the first device, calculating the deadline timestamp, sending the deadline to the second device, and setting up a timer locally on the second device based on the received deadline. Not all end-to-end constraints are periodic, and the second device does not know when to expect a deadline from the first device. This can be problematic when the network delay is large and the deadline arrives late or does not arrive at all. To make sure excessive network delays can be detected, the second device is set to send back an acknowledgment to the sender device. If the sender does not receive the ack in time, it executes a backup routine. We consider a **Worst-Case Round-Trip Delay (WCRTD)**—from sending the timestamp to receiving the acknowledge—to set the deadline for receiving the ack message. To maintain clock accuracy among devices, the `_clockSync()` is called periodically to perform clock synchronization. The following equation shows the relationship among period of clock synchronization in seconds (T_S), the clock's frequency drift in parts per million format (e_{ppm}), and the accuracy of the synchronization method (δ).

$$e_S = \min(\delta_S, T_S e_{ppm} \times 10^{-6}) \tag{6}$$

For clock synchronization, a repetitive timer with a fixed period is set to execute the function `_periodic()` function. In general, devices can synchronize their clock using **Network Time Protocol (NTP)** [35], **Precision Time Protocol (PTP)** [36], or GPS. For this work, we use a simplified version of NTP since PTP requires dedicated hardware and GPS is not always available. The value of δ for NTP, GPS, and PTP is 1 ms, 40 ns, and 10 ns, respectively. `_local2global(ts)` and `_global2local(ts)` functions are also inserted to convert a local timestamp into a global one and vice versa. Listings 12 and 13 show the generated code for the distributed latency constraint example (Listings 6 and 7) presented in the previous section. For this example, we set the WCRTD to be 300 ms.

```
1   TC1.activationTime = 0.3;  // WCRTD = 300 ms
2   TC2.activationTime = 0.5;  // 500 ms
3   TC1.priority = 1;
4   jmp_buf buf;
5   msg_t gResult;
6   while(1){
7       if (setjmp(buf)){
8           send(gResult);
9       }else{
10          ts = _local2global(now + TC2.deadline)
11          _sendTS(ts);
12          _startTimer(TC1);
13          data = sense();
```

```
            result = compute(data);
            send(result);
        }
    }

    void _ackReceived(TC1){
        _stopTimer(TC1);
    }

    void _timerExpired(TC1){
        _BRQManager(BR1(), TC1.priority);
        longjmp(buf);
    }

    void _periodic(){
        _clockSync();
    }

    void BR1(){
        gresult = backupRoutine1();
    }
```

Listing 12. Implementation for a distributed latency constraint: device 1.

```
TC2.priority = 1;
actuate_t gResult;
jmp_buf buf;
while(1){
    if (setjmp(buf)){
        actuate(gResult);
    } else {
        data = receive();
        result = compute(data);
        actuate(result);
        _stopTimer(TC2);
    }
}

void _tsReceived(TC2){
    TC2.deadline = _global2local(TC2.ts - now);
    _startTimer(TC2);
    _sendAck(TC1);
}

void _timerExpired(TC2){
    _BRQManager(BR2(), TC2.priority);
    longjmp(buf, 1);
}

void _periodic(){
    _clockSync();
}

```

```
void BR2(){
    gResult = backupRoutine2();
}
```

Listing 13. Implementation for a distributed latency constraint: device 2.

At the annotation location of the first event, device 1 converts and sends the deadline to device 2, then starts a timer locally. A callback (`_deadlineReceived()`) is set up in device 2's code to receive the sent timestamp (using an interrupt handler), perform timestamp translation, start a timer, and send back an ack to the sender. On the sender side, a callback (`_ackReceived()`) is set up to receive the ack (using an interrupt handler) and stop the timer. If the timer expires before receiving the ack, the `backupRoutine1()` is executed on the sender device. On the receiver side, the `backupRoutine2()` is executed if the timer expires.

5.4 Simultaneity Constraint

The code in Listing 14 shows the implementation code for the program in Listing 8.

```
float ts[3];
i = 0;
n = 0;
TC1.priority = 1;
TC2.priority = 2;
TC1.activationTime = 0.3;   // WCRTD = 300 ms
jmp_buf buf;
msg_t gResult;
while(1){
    if (setjmp(buf)){
        sendToServer(gResult);
    }else{
        ts[i] = _local2global(now);
        _sendTS(ts);
        _startTimer(TC1);
        data = sense();
        result = compute(data);
        sendToServer(result);
    }
}

void _tsReceived(TC1){
    i++;
    ts[i] = _global2local(TC1.ts);
    _sendAck();
    if (i == 2){
        if (!_verifySimultaneityLevel(ts,0.001)){
            _BRQManager(BR1(), TC1.priority);
            longjmp(buf, 1);
        }
        i = 0;
    }
}

```

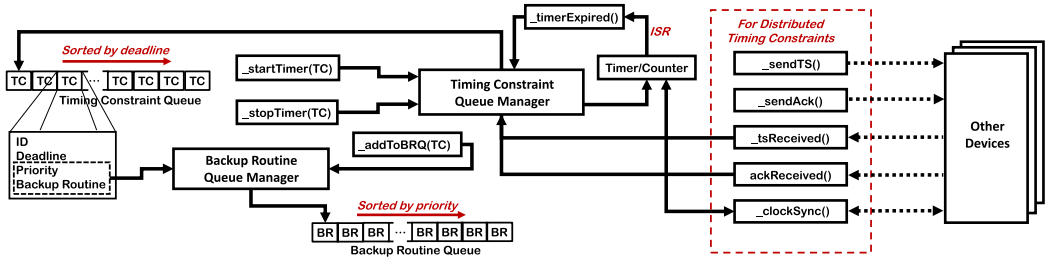

Fig. 7. Overview of the runtime management system for initiation of a timing constraint using timers, performing network communications and clock synchronization (when needed), and achieving deterministic execution of backup routines based on the specified priorities.

```
35  void _ackReceived(TC2){
36      n++;
37      if (n==3){
38          _stopTimer(TC2);
39          n=0;
40      }
41  }
42
43  void _timerExpired(TC2){
44      _BRQManager(BR2(), TC2.priority);
45      longjmp(buf, 1);
46  }
47
48  void _periodic(){
49      _clockSync();
50  }
51
52  void BR1(){
53      gResult = backupRoutine1();
54  }
55
56  void BR2(){
57      gResult = backupRoutine2();
58  }
```

Listing 14. Implementation for a simultaneity timing constraint: devices 1–3.

Each device takes a timestamp before sensing and broadcasts it. Upon receiving a timestamp from other devices, each device verifies that all timestamps meet the simultaneity constraint.

5.5 Virtual Timers

Since a platform may not have enough hardware timers available to independently implement the timing monitoring, we propose an efficient mechanism where multiple timers are implemented using a queue that maintains a list of timing constraints sorted based on their deadlines, and a timer is used to count the timing constraint that is at the head of the queue (with the earliest deadline). We refer to this queue as the **Timing Constraint Queue (TCQ)**. Figure 7 shows the

ALGORITHM 1: _StartTimer(TC)

```
1  if TC.deadline < TCQ[0].deadline then
2      elapsedTime = getTimerValue();
3      TCQ.shiftFrom(0);
4      TCQ[0] = TC;
5  else
6      for (i=1; i<=length(TCQ); i++) do
7          if TC.deadline < TCQ[i].deadline then
8              TCQ.shiftFrom(i);
9              TCQ[i] = TC;
10             break;
11         end
12     end
13 end
14 TCQM();
```

ALGORITHM 2: _StopTimer(TC) and _timerExpired(TC)

```
1  for (i=1; i<=length(TCQ); i++) do
2      TCQ[i].deadline = TCQ[i].deadline - TCQ[0].deadline;
3  end
4  TCQ[0] = [];
5  TCQM();
```

overview of Plan B's runtime monitoring mechanism. When the `_startTimer()` is called, the queue is updated to insert the new deadline. If the head of the queue is changed, the timer value is reset and the values of existing deadlines are updated. Algorithm 1 shows the pseudo-code for `_startTimer()`.

When `_stopTimer(TC)` is called or the timer is expired (`_timerExpired(TC)`), the value of deadlines is updated and the timer is armed again with the deadline of the queue's head. Algorithm 2 shows the pseudo-code for `_stopTimer(TC)` and `_timerExpired(TC)`.

It is possible to allocate more than a timer for the implementation of timing constraints on the TCQ. If n timers are available on a platform, the first n elements on the queue are assigned to existing timers.

If two timers expire at the same time or the interval for the execution of their corresponding backup routine overlaps, the one with higher priority should be executed first. To achieve this functionality, we use another queue, the BRQ, which holds all the backup routines to be executed. Upon violation of a timing constraint, its backup routine is added to the BRQ. The BRQ is managed by the BRQ Manager based on the priority values of backup routines. The BRQ Manager ensures that all backup routines are sorted based on their priority (from high to low) when a backup routine is added or removed and executes the backup routine at the head of the BRQ (highest priority). It may be possible to execute multiple backup routines at the same time, but it requires more information about their data dependency and is not considered in this article. When the timing constraint is distributed, the sender device passes the deadline and the receiver's address to the network send block, which is responsible for TCP communication. For receiving the timestamp and the ack, a non-blocking receive is implemented in network receive and network ack blocks. The synchronization block performs the clock synchronization and uses the highest desired accuracy for timekeeping.

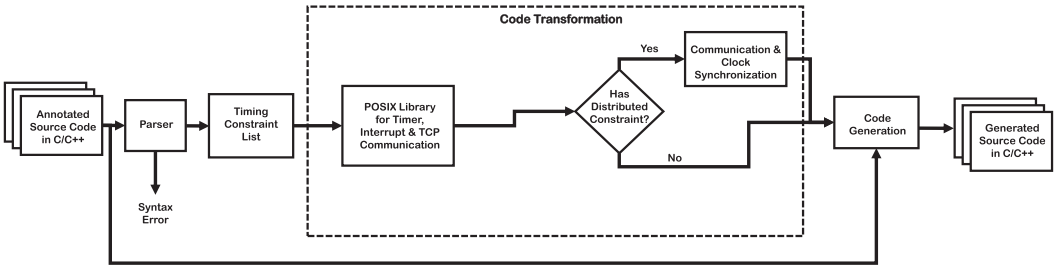

Fig. 8. Overview of the code transformation from source code with annotated timing constructs to C/C++ codes that can be compiled with existing compilers (e.g., gcc).

5.6 Code Transformation

We developed a checker that acts as a source-to-source transformer. Our checker gets the source code(s) that are annotated in C/C++ and checks the consistency of events and throws an error if there are redundant events. In addition, our checker generates the code for creation, configuration, and deletion of timers (see the appendix). We use the **Portable Operating System Interface (POSIX)** library for timer and signal configuration. Figure 8 shows an overview of our source-to-source checker. Initially, the parser checks for syntax error and then extracts timing constraints and events that are involved. Then, it checks if there are distributed timing constraints, and if yes, it add the code to send the timestamps and perform clock synchronization. Finally, the generated code should be compiled (e.g., using g++) to get the binary.

For POSIX-based code generation, we use `timer_create()` and `timer_settime()` functions to create and start a timer. We use `sigevent` and `sigaction` to define a signal event and attach it to the timer and also to specify the handler function that is linked to the signal action. We use real-time signals (RTSIG)—numbered from 32 to 64 in our implementation. We use the TCP socket functions to implement the message passing between devices of a distributed system. The `periodic()` function for synchronizing the clock is called at the fixed rate by setting up a timer at the beginning of the program. We use NTP clock synchronization [35] in our implementation. For platforms without POSIX support, we need to use hardware-dependent functions depending on the platform. Usually, in MCUs, timers and interrupts are set by directly writing into the corresponding registers. Currently, our API only supports the ESP8266 board and uses the `Ticker` library for configuring timers and interrupts. For TCP communication, we use the `ESP8266WiFi` library. After compilation and code transformation, the code for POSIX-based implementation is compiled with `gcc` (version 7.3.0) with `-lrt` flag. The code for ESP8266 is compiled with Arduino IDE, which uses the Xtensa lx106 toolchain.

6 EXPERIMENTS

6.1 Case Study I: Applying Plan B to a Complex Application—AV Full Software Stack

We use Plan B to redesign Apollo [10], an open source software for self-driving cars. We show that higher performance is achieved when the system is designed using Plan B while the safety of the AV is guaranteed. According to the architecture of Apollo software, the sensed data is first processed by the perception module and then passed to the prediction, planning, and control modules. Finally, the data is given to the CAN Bus Chassis module, which is responsible to send the actuation commands to the ECU.

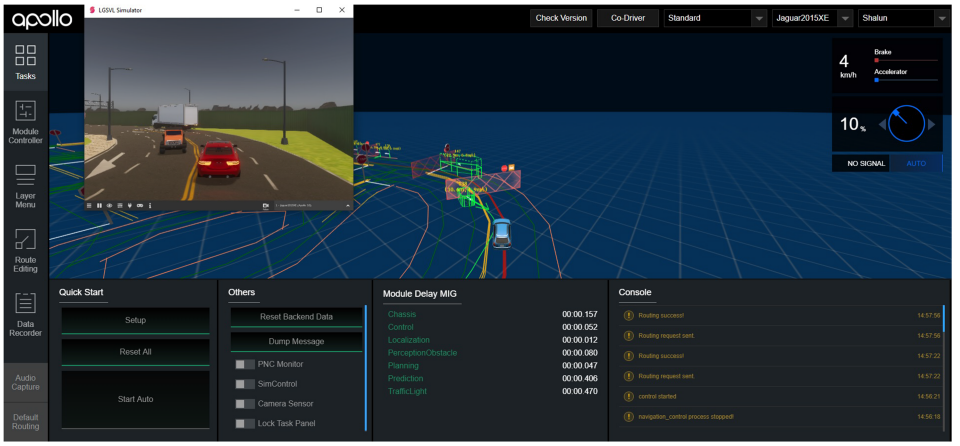

Fig. 9. A screenshot of Apollo [10] and the LG [37] simulator in the Shalun map.

6.1.1 Testbed Information. We used Apollo together with the LG simulator to perform software-in-the-loop (SIL) simulations. Details on how to install and bridge LGSVL with Apollo can be found on GitHub.[1] Due to the intensity of computation, we ran the LGSVL simulator on a Desktop machine and the Apollo on an Ubuntu laptop. The LG simulator was downloaded from GitHub[2] as a stand-alone simulator and executed on a desktop computer with an Intel Core i7-6700 CPU 3.4 GHz, 16 GB of memory, 128-MB Intel HD Graphics plus 8,121-MB shared system memory, and a 64-bit Windows 10 OS. We used Apollo v3.0 [10] and ran it on a high-performance laptop with Intel Core i7-7700 HQ CPU 2.8 GHz, 16 GB of memory, GeForce GTX 1060 PCIe/SSE2 (6 GB), and a 64-bit Ubuntu 18.04.3 LTS OS. The bridge between Apollo and the LGSVL simulator is established by running the `rosbridge.sh` inside the docker. The control panel of Apollo can be accessed from a browser at `localhost:8888`. The selected vehicle for experiments is a 2015 Jaguar XE, and the map for the experiment is Shalun, the Taiwan Car Lab Testing Facility located in Tainan, Taiwan. Figure 9 shows an overview of Apollo's control panel (Dreamview) and the LG simulator in the Shalun map. To spawn the ego AV (controlled by the Apollo software) and other NPC agents such as pedestrians and vehicles at the desired location, we used LGSVL's Python API.[3] After enabling the localization, perception, planning, prediction, routing, and control modules, the AV starts driving.

6.1.2 Safety-Based Timing Constraints. Based on the specification of Apollo software, the end-to-end delay from sensing to actuation should be less than 1.5 seconds and the maximum allowed velocity is 15 m/s (33 mph). We measured the end-to-end delay of the Apollo software for 1,000 executions by inserting a probe that captures a timestamp when the data is collected from the sensors and another one when the actuation signals are given to the CANbus module. Outputs from the execution of the backup routine are published directly to the CANbus topics, overriding the messages that are generated by the control module. The backup routine for cases where the end-to-end delay exceeds the requirement (1.5 seconds) is to stop the vehicle. There are three control commands that are generated by the backup routine: brake = 100%, throttle = 0%, and set_speed = 0.

[1] https://github.com/lgsvl/apollo-3.0.
[2] https://github.com/lgsvl/simulator/releases/.
[3] https://www.svlsimulator.com/docs/python-api/python-api/.

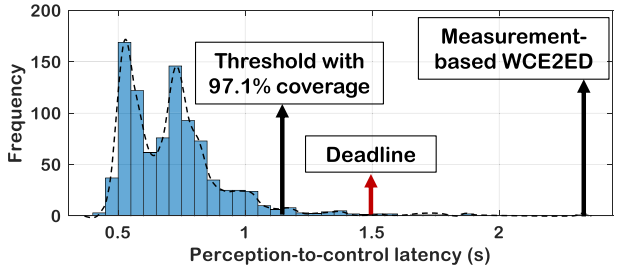

Fig. 10. Histogram of the end-to-end delay from perception to control for the Apollo software.

The other backup routine is to slow down by 70% where the set_speed is dynamically written to and the throttle and brake values are computed based on a simple PID controller as throttle = max(a, 0) and brake = min(a, 0), where the desired acceleration value is computed by a simple PID controller as $a = k_P * e + k_I * e_I + k_D * e_D$. $e = v_r - v$ is the error between the set_speed and actual velocity of the vehicle, e_I is the integral error, and e_D is the derivative error.

Figure 10 shows the histogram of the end-to-end delay values for Apollo software and the newly set threshold (1.5 seconds). As shown in Figure 10, the measurement-based WCE2ED for sensing-to-actuation (the longest observed execution time) is 2.8 seconds and a statically calculated WCE2ED is expected to be much larger. As a result, the AV software (with the current platform) does not meet the timing constraint of 1.5 seconds. If we apply the Plan B approach to bound the execution time by 1.5 seconds, the system is guaranteed to be safe and executes the backup routine if the delay is more than 1.5 seconds. For the case where the deadline is 1.5 seconds, the rate of timing violation is 0.6%. It should be noted that when the backup routine is slowing down by 70%, the initial end-to-end timing constraint is relaxed and the rate of timing violation is reduced even more. Another benefit is that by changing the end-to-end delay threshold, the operating point of the vehicle can change. For instance, if the threshold is set to be 1.2 seconds as indicated by a black arrow on Figure 10, the vehicle can drive at a higher velocity (40 mph) but the rate of timing violations is increased (3.2%).

We statically overestimated the WCET of the backup routine to be 5 ms. Apollo source code has more than 500,000 lines of code, which is almost 2,000 times larger than the size of the backup routine (24 lines). This highlights the benefit of Plan B in avoiding the pessimism of static WCET analysis for the whole software.

To further showcase the robustness of our approach, we injected faults on the Apollo software by inserting delay() functions with random duration at random locations in the code to intentionally cause a timing failure. We ran the Apollo software for 1 hour and injected 200 faults, where 12 of them could have resulted in an accident. Thanks to the Plan B approach, no accidents were observed when a timing failure occurred.

6.2 Case Study II: Resilient and Flexible Design of an Automated Intersection of AVs Using Plan B

We used Plan B to build a 1/10th-scale intersection of AVs.

6.2.1 Testbed Information. Our 1/10th-scale model AVs are built on a Traxxas chassis, are 50 cm long and 30 cm wide, and can drive up to 3.5 m/s. An ESP8266 NodeMCU v3 board is used to control the steering angle and velocity of the vehicle. The ESP8266 also gets data from an HC-SR04 ultrasonic sensor to maintain a safe distance from the front vehicle. ESP8266 boards communicate

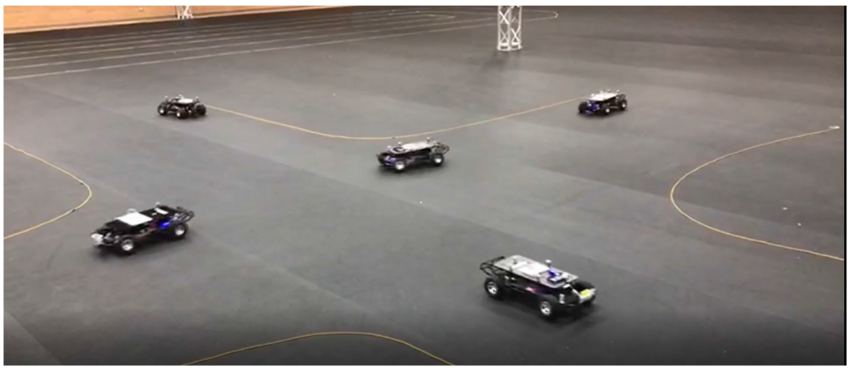

Fig. 11. Our signal-free intersection with 1/10th-scale model AVs. Vehicles use V2V communication to come up with a consensus about who crosses first, who crosses second, and so on.

with each other through a shared Wi-Fi network. A set of trackers are installed on each vehicle, and an OptiTrack system is used to determine the location and orientation of vehicles through high-speed cameras. We used the `mocap_optitrack` ROS (Robot Operating System) library to read the 2D position and heading angle of vehicles from the OptiTrack system. A MATLAB script was written to create a ROS node and pass the pose data to vehicles through the TCP over the Wi-Fi network (2.4 GHz). Figure 11 shows an overview of the intersection. A demo of the intersection can be found on YouTube.[4] The pose data is broadcast every 20 ms. AVs follow the predefined waypoints to enter the intersection, and once they leave the intersection, they make a U-turn and return to the intersection. The decision to whether turn left, right, or go straight is made randomly after making the U-turn.

6.2.2 Safety-Related Timing Constraints. To safely operate vehicles at the intersection, a number of timing constraints (including some distributed timing constraints) must be met, which are listed in Table 2.

We also defined the safe backup routine for timing violations. To show our approach is resilient against unforeseen timing failures, we ran the intersection and applied two types of fault that cause timing violations. First, we inserted a delay_ms() function to a random place in the vehicle's code. The delay has a random activation time, t_A, and a random duration between 1 and 1,000 ms. Second, we conducted a cyberattack on the Wi-Fi network that is used by AVs to resemble a jamming attack. The cyberattack was done using ESP Deauther 2.0 [38] software where a malicious agent disconnects one or more vehicles from the network persistently by sending a deauthentication packet to the server on behalf of the vehicle. Figure 12 shows the overview of applied faults on the intersection management system for violating the safety timing constraints.

We ran the intersection (with the vehicle's original code) for 1 hour and assigned random velocities to vehicles. We injected 200 faults, and 68 accidents were observed. We modified the vehicle code according to Plan B's methodology and added backup routines listed in Table 2. We repeated the previous experiment and injected the same number of faults. In this experiment, all vehicles reacted promptly when a fault was injected and no accident happened.

For system tuning and redesign, we measured the actual end-to-end delay of all timing constraints in Table 2 for 3,000 executions, again by capturing timestamps at different locations in

[4] https://www.youtube.com/watch?v=Q0tPS6uNTeE.

Table 2. List of Timing Constraints for the Autonomous Intersection Case Study and Possible Backup Routine to Be Executed upon Failure of Timing Requirements

#	Type	Name	Description	Backup Routine
1	Local	Obstacle Avoidance	To ensure the vehicle always maintains a safe distance from its front vehicle, the latency from sensors to actuator should be less than 20 ms.	Slow down and stop
3		Waypoint Tracking	To make sure the vehicle does not drive out of road boundaries, the latency from reading orientation and position of the vehicle to actuation should be less than 40 ms.	Change PID gains of the controllers
4	Distributed	Vehicle-to-Vehicle Communication	The latency from one vehicle sending its info to another vehicle receiving the info and writing to the DC motor should be less than 200 ms.	Reduce the reference velocity by 20%
5		Communication Ack	The latency from sending a deadline timestamp to receiving the "ack" should be less than 200 ms.	Reduce the reference velocity by 20%
6		Clock Synchronization	The latency between two clock synchronization should be less than 66 seconds.	Force time sync

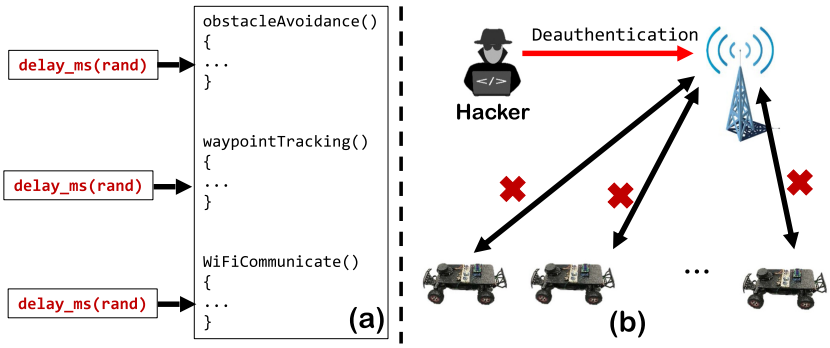

Fig. 12. Faults are injected to induce a timing failure. (a) A delay function with arbitrary duration is randomly inserted in the vehicle's code (b) A deauthentication attack is done to disconnect vehicles from the Wi-Fi network.

the code. We set the threshold for vehicle-to-vehicle timing constraint to be greater than 97.7% of all observed latencies, and for obstacle avoidance timing constraint, we set the threshold to be greater than 99.7% of all observed latencies. We have depicted the histogram of actual delays for obstacle avoidance delay and inter-vehicle communication latency in Figure 13. Using the newly adopted thresholds, AVs can drive up to 3 m/s while safety is guaranteed. However, if the system was designed based on conventional approaches, the max velocity of AVs was limited to 1 m/s. Let us assume the WCE2ED for the inter-vehicle communication is equal to the measurement-based value (i.e., WCE2ED = 563 ms). By selecting the threshold of 140 ms, vehicles at the intersection can drive almost three times faster. In this case, the backup routines (stopping the AVs) are invoked more frequently but the rate is low (0.3%).

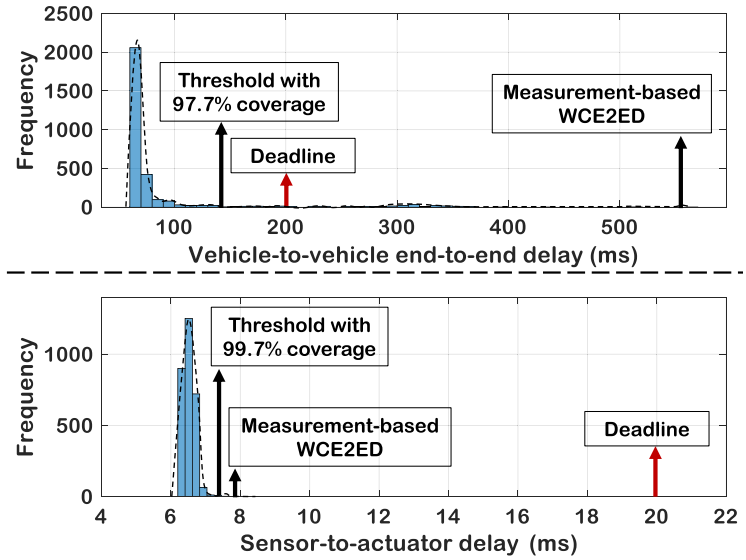

Fig. 13. Top: Histogram of the inter-vehicle end-to-end delay. Bottom: Histogram of the vehicle's obstacle avoidance delay. The original deadline, newly set threshold after the redesign, and actual measurement-based WCE2ED are also shown.

6.3 Case Study III: Resilient and Flexible Control of a Quadcopter Using Plan B

We simulated the behavior of a quadcopter, and its dynamics are modeled using the following equation [39]:

$$\begin{cases} \dot{x} = w(\sin\phi\sin\psi + \cos\phi\cos\psi\sin\theta) - v(\cos\phi\sin\psi - \cos\psi\sin\phi\sin\theta) + u\cos\psi\cos\theta \\ \dot{y} = v(\cos\phi\cos\psi + \sin\phi\sin\psi\sin\theta) - w(\cos\phi\sin\psi - \cos\phi\sin\psi\sin\theta) + u\cos\theta\sin\psi \\ \dot{h} = w\cos\phi\cos\theta - u\sin\theta + v\cos\theta\sin\phi \\ \dot{u} = rv - qw - g\sin\theta \\ \dot{v} = pw - ru + g\cos\theta\sin\phi \\ \dot{w} = qu - pv - F/m + g\cos\phi\sin\theta \\ \dot{\phi} = p + r\cos\phi tan\theta = q\sin\phi tan\theta \\ \dot{\theta} = q\cos\phi - r\sin\phi \\ \dot{\psi} = r\cos\phi/\cos\theta + q\sin\phi/\cos\theta \\ \dot{p} = \tau_\phi/J_x + qr(J_y - J_z)/J_x \\ \dot{q} = \tau_\theta/J_y + pr(J_x - J_z)/J_y \\ \dot{r} = \tau_\psi/J_z + pq(J_x - J_y)/J_z \end{cases}, \quad (7)$$

where x and y are the latitude and longitude of the quadcopter; h is the height; and u, v, and w are the linear velocity toward x, y, and z axes, respectively. ϕ, θ, ψ are the orientation of the quadcopter, and p, q, r are the angular velocities over x, y, and z axes, respectively. m is the mass of the drone, and J_x, J_y, and J_z are the moment of inertia over x, y, and z axes. The defined mission for the drone is to take off, reach 20 m of altitude, and enter a 1×1 m square window. The size of the drone is $40 \times 40 \times 20$ cm.

Table 3. Parameters of the Simulated Drone

J_x, J_y	J_z	m	g	k_1	k_2	k_3, k_7, k_{11}	k_4, k_8, k_{12}	k_5, k_9	k_6, k_{10}
0.012	0.02	1.4	9.8	1	0.5	1	0.5	0.05	0.1

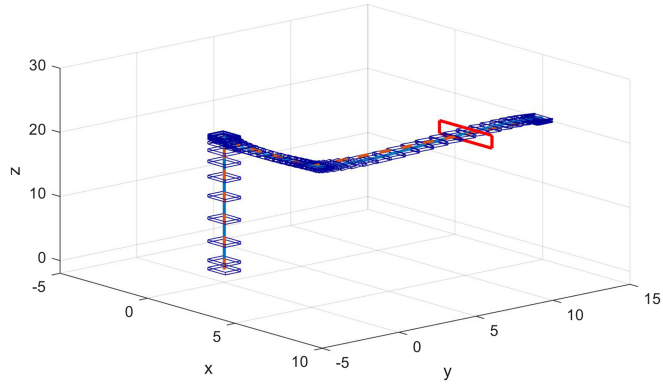

Fig. 14. An overview view of the attitude of the simulated drone for entering a house through a window (depicted in red).

The following PD (Proportional and Derivative) controllers are used to control the location and orientation of the drone over the z-axis (ψ):

$$\begin{cases} F = mg - K(k_1(h - h_r) + k_2 w) \\ \tau_\phi = K(-k_3\phi - k_4 p - k_5(y - y_r) - k_6 v) \\ \tau_\theta = K(-k_7\theta - k_8 q + k_9(x - x_r) + k_{10} u) \\ \tau_\psi = K(-k_{11}(\psi - \psi_r) - k_{12} r) \end{cases}, \tag{8}$$

where $k_1, k_2, \ldots, k_{12}$ are positive constants (gains) and K is a constant multiplier that is modified in the backup routine.

If the propeller speed of motors are ω_1, ω_2, ω_3, and ω_4, the desired speed can be calculated as

$$\begin{cases} \omega_1^2 = F/4b - \tau_\theta/2bl - \tau_\psi/4d \\ \omega_2^2 = F/4b - \tau_\phi/2bl + \tau_\psi/4d \\ \omega_3^2 = F/4b + \tau_\theta/2bl - \tau_\psi/4d \\ \omega_4^2 = F/4b + \tau_\phi/2bl + \tau_\psi/4d \end{cases}, \tag{9}$$

where b is the trust factor, d is the drag factor, and l is the distance between the center of the quadrotor and the center of the propeller.

Figure 14 shows the simulated drone in Matlab and its location at different moments.

The goal for the drone is to enter the house through the designated window (depicted in red) when the delay is variable. The drone starts from [0, 0, 0] at time $t = 0$, and the intermediate reference points are [20, 0, 0] at $t = 2.5$, [20, 0, 5] at $t = 5$, [20, 10, 5] at $t = 6$, and [20, 15, 5] at $t = 8$.

If the delay from the IMU (Inertial Measurement Unit) to the ESC (Electronic Speed Controller) is more than a threshold, the drone either becomes unstable and crashes or cannot finish the designated mission and hits the boundary of the window. We have depicted the relationship between the controller's delay and variation in the gains of the PD controllers in Figure 15. The orange dot represents the nominal operating point of the quadcopter where $k = 1.8$ and the corresponding deadline for the controller is 20 ms.

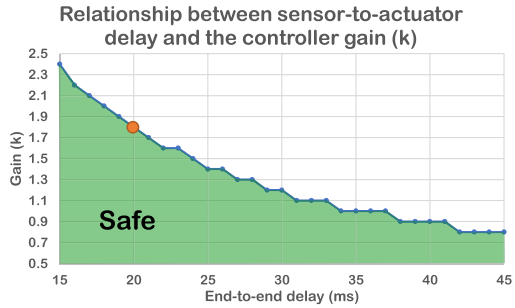

Fig. 15. Relationship between the controller's delay and the controller's gain, k (see Equation (8)). The orange dot shows the nominal operating point of the drone.

7 CONCLUSION

In this article, we presented a novel design methodology for time-sensitive CPS called *Plan B* that allows CPS designers to specify what happens if a timing constraint is not met. Using online monitoring of timing requirements at the runtime, timing violations can be detected and backup routines can be executed on-time to maintain safety. Plan B also relaxes the timing constraints of the system by renegotiating timing contracts to achieve graceful degradation instead of a complete shutdown. We have evaluated our approach on two real case studies, a 1/10th-scale model intersection of AVs and the software of Apollo for autonomous driving together with the LG simulator. Despite injecting faults to cause timing failures, both systems were able to maintain safety through the execution of backup routines. Future improvements can be done by developing an API for different programming languages.

APPENDIX

A SAMPLE CODES

In Listings 15 through 21, we present some sample codes for start timer, stop timer, timer expired, and TCP socket initialization.

```
1  void timerInit()
2  {
3          sa.sa_flags = SA_SIGINFO;
4          sa.sa_sigaction = handler;
5          if (sigaction(SIG, &sa, NULL) == -1)
6                  errExit("sigaction");
7          sev.sigev_notify = SIGEV_SIGNAL;
8          sev.sigev_signo = SIG;
9          sev.sigev_value.sival_ptr = &timerid;
10         if (timer_create(CLOCKID, &sev, &timerid) == -1)
11                 errExit("timer_create");
12         Q.length = 0;
13 }
```

Listing 15. Timer initialization and signal attachment.

```
1  static void handler(int sig, siginfo_t *si, void *uc){
2          _expired();
3  }
```

Listing 16. Handler for timer expiration signal.

```
void _startTimer(TC TCI)
{
        if (Q.length == 0)
        {
                Q.IDs[0] = TCI.ID;
                Q.deadlines[0] = TCI.deadline;
                Q.length = 1;
        }
        else
        {
                timer_gettime(timerid, &its);
                long long timerValue = its.it_value.tv_nsec + its.it_value.
                    tv_sec * 1000000000;
                long long dt = Q.deadlines[0] - timerValue;
                if (printLOG == 1)
                {
                        printf("current␣time␣elapsed␣is␣%lld\n", dt);
                }
                for (int i = 0; i < Q.length; i++)
                { // update deadlines
                        Q.deadlines[i] = Q.deadlines[i] - dt;
                }
                int inserted = 0;
                for (int i = 0; i < Q.length; i++)
                { // insert new TC in the Queue
                        if (TCI.deadline < Q.deadlines[i] && inserted == 0)
                        {
                                for (int j = Q.length; j >= i; j--)
                                {
                                        Q.IDs[j + 1] = Q.IDs[j];
                                        Q.deadlines[j + 1] = Q.deadlines[j];
                                }
                                Q.IDs[i] = TCI.ID;
                                Q.deadlines[i] = TCI.deadline;
                                inserted = 1;
                                Q.length++;
                                break;
                        }
                }
                if (inserted == 0)
                { // if it's the last element
                        Q.IDs[Q.length] = TCI.ID;
                        Q.deadlines[Q.length] = TCI.deadline;
                        inserted = 1;
                        Q.length++;
                }
        }
        freq_nanosecs = Q.deadlines[0]; // last element of the queue...
        its.it_value.tv_sec = freq_nanosecs / 1000000000;
        its.it_value.tv_nsec = freq_nanosecs % 1000000000;
        its.it_interval.tv_sec = its.it_value.tv_sec;
        its.it_interval.tv_nsec = its.it_value.tv_nsec;
```

```
52          if (timer_settime(timerid, 0, &its, NULL) == -1)
53                  errExit("timer_settime");
54          timer_gettime(timerid, &its);
55          long long timerValueF = its.it_value.tv_nsec + its.it_value.tv_sec *
                1000000000;
56          timer_gettime(timerid, &its);
57          timerValueF = its.it_value.tv_nsec + its.it_value.tv_sec *
                1000000000;
58
59  }
```

Listing 17. Start timer.

```
1   void _stopTimer(TC TCI)
2   {
3           int TCLocation = -1;
4           timer_gettime(timerid, &its);
5           long long timerValue = its.it_value.tv_nsec + its.it_value.tv_sec *
                1000000000;
6           long long dt = Q.deadlines[0] - timerValue;
7           for (int i = 0; i < Q.length; i++)
8           { // update deadlines
9                   Q.deadlines[i] = Q.deadlines[i] - dt;
10          }
11          for (int i = 0; i < Q.length; i++)
12          {
13                  if (TCI.ID == Q.IDs[i])
14                  {
15                          TCLocation = i;
16                          break;
17                  }
18          }
19
20          if (TCLocation == -1)
21          { // was not found, it's already expired
22                  // do nothing
23                  if (printLOG == 1)
24                  {
25                          printf("TC␣not␣found...\n");
26                  }
27          }
28          else
29          {
30                  if (TCLocation == Q.length - 1)
31                  { // last element
32                          Q.IDs[TCLocation] = -1;
33                          Q.deadlines[TCLocation] = -1;
34                          Q.length--;
35                  }
36                  else
37                  {
38                          for (int i = TCLocation; i < Q.length - 1; i++)
39                          {
```

```
                                Q.IDs[i] = Q.IDs[i + 1];
                                Q.deadlines[i] = Q.deadlines[i + 1];
                        }
                        Q.length--;
                }
        }

        if (Q.length > 0)
        {
                freq_nanosecs = Q.deadlines[0];
                its.it_value.tv_sec = freq_nanosecs / 1000000000;
                its.it_value.tv_nsec = freq_nanosecs % 1000000000;
                its.it_interval.tv_sec = its.it_value.tv_sec;
                its.it_interval.tv_nsec = its.it_value.tv_nsec;
                if (timer_settime(timerid, 0, &its, NULL) == -1)
                        errExit("timer_settime");
        }
        else
        {
                its.it_value.tv_sec = 0;
                its.it_value.tv_nsec = 0;
                its.it_interval.tv_sec = its.it_value.tv_sec;
                its.it_interval.tv_nsec = its.it_value.tv_nsec;
                if (timer_settime(timerid, 0, &its, NULL) == -1)
                        errExit("timer_settime");
        }
}
```

Listing 18. Stop timer.

```
void _expired()
{
        TC TCI;
        TCI.ID = Q.IDs[0];
        TCI.deadline = Q.deadlines[0];

        int TCLocation = -1;
        timer_gettime(timerid, &its);
        long long dt = Q.deadlines[0];

        for (int i = 1; i < Q.length; i++)
        { // update deadlines
                Q.deadlines[i] = Q.deadlines[i] - dt;
        }

        for (int i = 0; i < Q.length; i++)
        {
                if (TCI.ID == Q.IDs[i])
                {
                        TCLocation = i;
                        break;
                }
        }
```

```
24
25          if (TCLocation == -1)
26          { // was not found, it's already expired
27                  // do nothing
28                  printf("TC not found...\n");
29          }
30          else
31          {
32                  if (TCLocation == Q.length - 1)
33                  { // last element
34                          Q.IDs[TCLocation] = -1;
35                          Q.deadlines[TCLocation] = -1;
36                          Q.length--;
37                  }
38                  else
39                  {
40                          for (int i = TCLocation; i < Q.length - 1; i++)
41                          {
42                                  Q.IDs[i] = Q.IDs[i + 1];
43                                  Q.deadlines[i] = Q.deadlines[i + 1];
44                          }
45                          Q.length--;
46                  }
47          }
48
49          if (Q.length > 0)
50          {
51                  freq_nanosecs = Q.deadlines[0];
52                  its.it_value.tv_sec = freq_nanosecs / 1000000000;
53                  its.it_value.tv_nsec = freq_nanosecs % 1000000000;
54                  its.it_interval.tv_sec = its.it_value.tv_sec;
55                  its.it_interval.tv_nsec = its.it_value.tv_nsec;
56                  if (timer_settime(timerid, 0, &its, NULL) == -1)
57                          errExit("timer_settime");
58          }
59          else
60          {
61                  its.it_value.tv_sec = 0;
62                  its.it_value.tv_nsec = 0;
63                  its.it_interval.tv_sec = its.it_value.tv_sec;
64                  its.it_interval.tv_nsec = its.it_value.tv_nsec;
65                  if (timer_settime(timerid, 0, &its, NULL) == -1)
66                          errExit("timer_settime");
67          }
68
69          if (TCI.ID == 1)
70          {
71                  backupRoutine2();
72          }
73  }
```

Listing 19. Timer expired.

```
int TCP_ServerInitialize()
{
        int server_fd;
        struct sockaddr_in address;
        int opt = 1;
        int addrlen = sizeof(address);
        char buffer[1024] = {0};
        char *hello = "Hello from server";

        if ((server_fd = socket(AF_INET, SOCK_STREAM, 0)) == 0)
        {
                perror("socket failed");
                exit(EXIT_FAILURE);
        }

        if (setsockopt(server_fd, SOL_SOCKET, SO_REUSEADDR | SO_REUSEPORT, &
            opt, sizeof(opt)))
        {
                perror("setsockopt");
                exit(EXIT_FAILURE);
        }
        address.sin_family = AF_INET;
        address.sin_addr.s_addr = INADDR_ANY;
        address.sin_port = htons(PORT);

        if (bind(server_fd, (struct sockaddr *)&address, sizeof(address)) <
            0)
        {
                perror("bind failed");
                exit(EXIT_FAILURE);
        }
        if (listen(server_fd, 3) < 0)
        {
                perror("listen");
                exit(EXIT_FAILURE);
        }
        if ((socket1 = accept(server_fd, (struct sockaddr *)&address, (
            socklen_t *)&addrlen)) < 0)
        {
                perror("accept");
                exit(EXIT_FAILURE);
        }
        if (fcntl(socket1, F_SETFL, fcntl(socket1, F_GETFL) | O_NONBLOCK) <
            0)
        {
                printf("TCP error\n");
        }
        return socket1;
}
```

Listing 20. TCP server initialization.

```
int TCP_ClientInitialize()
```

```
{
        struct sockaddr_in serv_addr;
        char *hello = "Hello from client";
        char buffer[1024] = {0};
        if ((socket1 = socket(AF_INET, SOCK_STREAM, 0)) < 0)
        {
                printf("\n Socket creation error \n");
                return -1;
        }

        serv_addr.sin_family = AF_INET;
        serv_addr.sin_port = htons(PORT);

        if (inet_pton(AF_INET, "127.0.0.1", &serv_addr.sin_addr) <= 0)
        {
                printf("\nInvalid address Address not supported \n");
                return -1;
        }

        if (connect(socket1, (struct sockaddr *)&serv_addr, sizeof(serv_addr)
            ) < 0)
        {
                printf("\nConnection Failed \n");
                return -1;
        }
        if (fcntl(socket1, F_SETFL, fcntl(socket1, F_GETFL) | O_NONBLOCK) <
            0)
        {
                printf("TCP Error\n");
        }

        return socket1;
}
```

Listing 21. TCP client initialization.

REFERENCES

[1] Mohammad Abdullah Al Faruque and Arquimedes Canedo. 2019. *Design Automation of Cyber-Physical Systems*. Springer.

[2] Edward A. Lee. 2008. Cyber physical systems: Design challenges. In *Proceedings of the 2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'08)*. IEEE, Los Alamitos, CA, 363–369.

[3] Edward A. Lee. 2010. CPS foundations. In *Proceedings of the Design Automation Conference*. IEEE, Los Alamitos, CA, 737–742.

[4] Frank Mueller. 2006. Challenges for cyber-physical systems: Security, timing analysis and soft error protection. In *Proceedings of the High-Confidence Software Platforms for Cyber-Physical Systems Workshop (HCSP-CPS'06)*, Vol. 6.

[5] Patricia Derler, Thomas Huining Feng, Edward A. Lee, Slobodan Matic, Hiren D. Patel, Yang Zhao, and Jia Zou. 2008. *PTIDES: A Programming Model for Distributed Real-Time Embedded Systems*. Technical Report. UC Berkeley.

[6] Thomas A. Henzinger, Benjamin Horowitz, and Christoph Meyer Kirsch. 2001. Giotto: A time-triggered language for embedded programming. In *Proceedings of the International Workshop on Embedded Software*. 166–184.

[7] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, et al. 2008. The worst-case execution-time problem—Overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems* 7, 3 (2008), 36.

[8] MIT Technology Review. 2019. Many Cars Have a Hundred Million Lines of Code: Who Gets to Write It? Retrieved October 10, 2019 from https://tinyurl.com/zwczp55.

[9] Marten Lohstroh, Patricia Derler, and Marjan Sirjani. 2018. *Principles of Modeling: Essays Dedicated to Edward A. Lee on the Occasion of His 60th Birthday*. Vol. 10760. Springer.
[10] Apollo Auto. 2019. Apollo: An Open Autonomous Driving Platform. Retrieved October 14, 2019 from https://github.com/ApolloAuto/apollo.
[11] Tulika Mitra and Abhik Roychoudhury. 2002. A framework to model branch prediction for worst case execution time analysis. In *Proceedings of the 2nd International Workshop on Worst-Case Execution Time Analysis*.
[12] Jean-François Deverge and Isabelle Puaut. 2007. Safe measurement-based WCET estimation. In *Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis (WCET'05)*.
[13] Reinder J. Bril, Johan J. Lukkien, and Wim F. J. Verhaegh. 2007. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems (ECRTS'07)*. IEEE, Los Alamitos, CA, 269–279.
[14] Raimund Kirner, Ingomar Wenzel, Bernhard Rieder, and Peter Puschner. 2005. Using measurements as a complement to static worst-case execution time analysis. *Intelligent Systems at the Service of Mankind* 2 (2005), 8.
[15] Heiko Falk, Sascha Plazar, and Henrik Theiling. 2007. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign*. ACM, New York, NY, 143–148.
[16] Yau-Tsun Steven Li and Sharad Malik. 1995. Performance analysis of embedded software using implicit path enumeration. *ACM SIGPLAN Notices* 30 (1995), 88–98.
[17] Christoph Cullmann. 2013. Cache persistence analysis: Theory and practice. *ACM Transactions on Embedded Computing Systems* 12 (2013), 40.
[18] Ingomar Wenzel, Raimund Kirner, Bernhard Rieder, and Peter Puschner. 2005. Measurement-based worst-case execution time analysis. In *Proceedings of the 3rd IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems*. IEEE, Los Alamitos, CA, 7–10.
[19] Sebastian Altmeyer and Robert I. Davis. 2014. On the correctness, optimality and precision of static probabilistic timing analysis. In *Proceedings of the Conference on Design, Automation, and Test in Europe Conference and Exhibition (DATE'14)*. 26.
[20] Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quinones, and Francisco J. Cazorla. 2012. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems (ECRTS'12)*. IEEE, Los Alamitos, CA, 91–101.
[21] Guillem Bernat, Antoine Colin, and Stefan M. Petters. 2002. WCET analysis of probabilistic hard real-time systems. In *Proceedings of the 2002 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE, Los Alamitos, CA, 279–288.
[22] Christian Ferdinand and Reinhold Heckmann. 2004. aiT: Worst-case execution time prediction by static program analysis. In *Building the Information Society*. Springer, 377–383.
[23] Niklas Holsti and Sami Saarinen. 2002. *Status of the Bound-T WCET Tool*. Space Systems Finland Ltd.
[24] Rapita Systems. n.d. RapiTime: WCET Analysis Tool. Retrieved March 27, 2019 from https://www.rapitasystems.com/products/rapitime.
[25] UPPAAL. n.d. Home Page. Retrieved March 27, 2019 from http://uppaal.org.
[26] Simon Schliecker and Rolf Ernst. 2009. A recursive approach to end-to-end path latency computation in heterogeneous multiprocessor systems. In *Proceedings of the 7th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '09)*. ACM, New York, NY, 433–442.
[27] Matthias Becker, Dakshina Dasari, Saad Mubeen, Moris Behnam, and Thomas Nolte. 2017. End-to-end timing analysis of cause-effect chains in automotive embedded systems. *Journal of Systems Architecture* 80 (2017), 104–113.
[28] Stanley Bak, Deepti K. Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. 2009. The system-level simplex architecture for improved real-time embedded system safety. In *Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*. IEEE, Los Alamitos, CA, 99–107.
[29] Rajeev Alur, Thao Dang, and Franjo Ivančić. 2003. Progress on reachability analysis of hybrid systems using predicate abstraction. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*. 4–19.
[30] Wei Xiao, Calin Belta, and Christos G. Cassandras. 2019. Decentralized merging control in traffic networks: A control barrier function approach. In *Proceedings of the 10th ACM/IEEE International Conference on Cyber-Physical Systems*. 270–279.
[31] Mrdjan Jankovic. 2018. Control barrier functions for constrained control of linear systems with input delay. In *Proceedings of the 2018 Annual American Control Conference (ACC'18)*. IEEE, Los Alamitos, CA, 3316–3321.
[32] Stephen Prajna and Ali Jadbabaie. 2004. Safety verification of hybrid systems using barrier certificates. In *Proceedings of the International Workshop on Hybrid Systems: Computation and Control*. 477–492.
[33] Xin Chen, Erika Ábrahám, and Sriram Sankaranarayanan. 2013. Flow*: An analyzer for non-linear hybrid systems. In *Proceedings of the International Conference on Computer Aided Verification*. 258–263.

[34] Georgios E. Fainekos, Sriram Sankaranarayanan, Koichi Ueda, and Hakan Yazarel. 2012. S-TaLiRo. In *Proceedings of the 2012 American Control Conference (ACC'12)*. IEEE, Los Alamitos, CA, 3567–3572.
[35] David Mills. 1985. *Network Time Protocol*. Technical Report RFC 958. M/A-COM Linkabit.
[36] John Eidson and Kang Lee. 2002. IEEE 1588 standard for a precision clock synchronization protocol for networked measurement and control systems. In *Proceedings of the Sensors for Industry Conference*. IEEE, Los Alamitos, CA, 98–105.
[37] GitHub. n.d. LG Simulator Group in Silicon Valley. Retrieved November 10, 2019 from https://github.com/lgsvl/simulator.
[38] Stefan Kremser. n.d. ESP8266 Deauther Version 2.0. Retrieved October 31, 2018 from https://github.com/spacehuhn/esp8266_deauther.
[39] Antonio Eduardo Carrilho da Cunha. 2015. Benchmark: Quadrotor attitude control. *EPiC Series in Computer Science* 34 (2015), 57–72.

Received June 2021; revised December 2021; accepted January 2022