SAFETY AND SECURITY OF CYBER-PHYSICAL SYSTEMS

BY

FARDIN ABDI TAGHI ABAD

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Doctoral Committee:

Professor Marco Caccamo, Chair
Professor Sibin Mohan, Co-Chair
Professor Lui R. Sha
Professor Taylor Johnson, Vanderbilt University

ABSTRACT

The number of embedded controllers in charge of physical systems has rapidly increased over the past years. Embedded controllers are present in every aspect of our lives, from our homes to our vehicles and factories. The complexity of these systems is also more than ever. These systems are expected to deliver many features and high performance without trading off in robustness and assurance. As systems increase in complexity, however, the cost of formally verifying their correctness and eliminating security vulnerabilities can quickly explode. On top of the unintentional bugs and problems, malicious attacks on cyber-physical systems (CPS) can also lead to adverse outcomes on physical plants. Some of the recent attacks on CPS are focused on causing physical damage to the plants or the environment. Such intruders make their way into the system using cyber exploits but then initiate actions that can destabilize and even damage the underlying (physical) systems.

Given the reality mentioned above and the reliability standards of the industry, there is a need to embrace new CPS design paradigms where faults and security vulnerabilities are the norms rather than an anomaly. Such imperfections must be assumed to exist in every system and component unless it is formally verified and scanned. Faults and vulnerabilities should be safely handled and the CPS must be able to recover from them at run-time. Our goal in this work is to introduce and investigate a few designs compatible with this paradigm. The architectures and techniques proposed in this dissertation do not rely on the testing and complete system verification. Instead, they enforce safety at the highest level of the system and extend guaranteed safety from a few certified components to the entire system. These solutions are carefully curated to utilize unverified components and provide guaranteed performance.

ACKNOWLEDGMENTS

No man is an island and no Ph.D. comes to fruition without the inspiration, companionship, and support of an entire community. It is impossible for me to thank each soul that helped me along the way. There are a great many who contributed in ways I may have understood at the time and did so with kindness.

Before all else, I have to thank my advisor Professor Marco Caccamo for providing his mentorship and intellectual support during my Ph.D. My path was not smooth, nor straight but Marco always had clear, constructive advice that helped me navigate through the numerous challenges. This dissertation simply would not have been possible without him. Thank you, Professor Sibin Mohan, for being especially energetic and insightful on all our collaborations. I am particularly grateful for your willingness to view me as a colleague and valuing my ideas and opinions in ways that always provided me with much-needed confidence. I would like to also thank Professor Lui Sha for sharing not only his academic and technical expertise but also his personal wisdom and experiences with me. I am also grateful to Professor Taylor Johnson for taking part in my committee and providing his valuable insight into my work. To the University of Illinois at Urbana-Champaign, Department of Computer Science and its excellent staff, I am thankful for allowing me to learn from and work alongside the world-class experts. I would like to seize the opportunity to thank my professors in University of Tehran and my teachers in Modarres high school of Khoy who helped me forge a solid scientific foundation.

I am fortunate to have been surrounded by many bright souls during my time in Champaign-Urbana. First and foremost, I want to thank Renato Mancuso for being so generous with his sharp mind and brilliant humor which made all the difference. Thanks to you I have watched Rick and Morty three times so far and check Reddit regularly every day. Cheers to Neriman Tokcan who quickly became and remains the closest thing I have had in the States to a sister. While I am pretty sure she is grateful I introduced her to Renato, I can honestly say that my Ph.D. experience was richer, more humane, and more colorful because Renato and Neriman Were in it.

I am indebted to Stanley Bak for his mentorship and guidance in the early years of my graduate school. I also must thank Rohan Tabish, Chien-Ying Chen, and Monowar Hassan for being excellent collaborators and friends. Through long days, long nights, hard projects, and tough reviewers, we all came out stronger. I am also grateful to Or Dantsker for his hard work building our experimental airplanes and how well he helped me get to know the

local cornfields when searching through them later.

I also want to express how thankful I am for the community of friends I found in Urbana-Champaign. Cheers to all my roommates who helped me acclimate and build a home base. Many breakthroughs happened among friends in a grad student living room with a glass of tea in one hand and a question in the other. I must thank Nikita Spirin for the way we could bond over remarkable ideas. To Mohammad Babaeizadeh and Hadi Hashemi, I am deeply grateful though I cannot explain just how meaningful their insights and friendships were. Hadi and Shekoofeh Mokhtari kept me rooted and whole while Mohammad always brought a new game or debate to the table in the style that only he can. Cheers to Adel Ahmadyan, the one person that I can always turn into for anything and he is always there, with the most vibrant, generous smile. To my other roommate, Faraz Faghri, and for the friendships of Siddharth Gupta, Jackie Alexandra, and Amin Ansari I am also indebted. Amin was and is my go-to person whenever I face hard decisions and , somehow, he always manages to find an angle that I have missed. I am grateful to Paul Rauch. Despite never really living in Champaign-Urbana, he is an indispensable part of my most unforgettable times there. I want to thank my dear friend Hossein Tagharobi Nia whom I have known since high school and has been like an older brother to me – despite being the same age. I am also grateful and consider myself lucky that I have known Prakalp Srivastava, Jeremy Goodsitt. Hassan Eslami, Babak Behzad, Sam Hamedirad, Austin Walters, Rasoul and Jalal Etesami, Xing Gao, and Kuan-Yu Tseng.

I want to thank my wonderful wife, Alia Bellwood especially. I could not have asked for a better partner. Not only she is the most supportive person at times of stress and uncertainty, but also she is a very gifted writer whose expertise I have greatly benefited from during my Ph.D., in ways that only she is aware of. I am thankful to University of Illinois that introduced her to my life.

Finally, I dedicate this entire dissertation to the memory of my father, Mehran Abdi, and to my mother, Hajiyeh Soltani, who both sacrificed the most for it. My mom is undoubtedly the happiest that it is now finished. And to my little brother, Armin Abdi, I am impressed with how well he held down the fort without me. He thrived on raw determination and love and I could not be prouder. I know that his hard work gave me peace of mind to focus on the task I now finish.

Fardin Abdi
Apr 2019, Seattle

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

Embedded controllers with smart capabilities are increasingly used to implement cyber-physical systems (CPS) with applications in many areas such as the Internet of Things. Modern medical devices, smart home appliances, smart vehicles, and avionics systems to name a few, are required to deliver increasingly high performance without trading off in robustness and assurance. Unfortunately, satisfying the increasing demand for smart capabilities and high performance means deploying increasingly complex systems. Even seemingly simple embedded control systems often contain a multitasking kernel, support networking, utilize many open source libraries [1], and a number of specialized hardware components (GPUs, DSPs, DMAs, *etc.*). As systems increase in complexity, however, the cost of formally verifying their correctness and eliminating security vulnerabilities can quickly explode.

In the past, many products that fell into the CPS category were also considered highly safety critical and therefore, were subject to serious safety standards. Such projects were required by various government and/or safety regulatory organizations to demonstrate a certain level of reliability and assurance through testing and/or verification before they could acquire operation certificates. These projects also were expected by corporations and government to be expensive, high-budget with a long development to market cycle and infrequent upgrades. Numerous examples of such projects can be found in areas such as avionics, medical equipment, or manufacturing devices and automotive industry. For these projects, the cost of verification and testing could add up to 40 percent of the total production cost.

Today, we live in a different world where refrigerators that keep our food safe also connect to the Internet and thermostats that control the temperature of our infants' rooms also connect to our smart-phones. Many of these seemingly benign products can cause real damage that in some cases is comparable with earlier safety-critical systems. Think about the following scenario; a malfunctioning thermostat can drop cause the temperature to fall below a level that is unsafe for a newborn baby. Or a malfunctioning refrigerator may cause food poisoning for the entire household. Modern seemingly benign CPS compose a significant part of the Internet of Things applications. The physical components of these systems may not be as sophisticated and unstable as the traditional safety-critical CPS, but the risks they can pose are indeed as real.

In an ideal world, software is fully verified and scanned for logical and implementation correctness as well as security vulnerabilities. This is, however, infeasible for the modern

real-world systems due to their level of complexity. Today, Linux Kernel contains more than 250,000 lines of code which resembles the complexity level in many other operating systems as well. To achieve the stringent safety requirements[1], the software and hardware need to be exhaustively tested to ensure nothing will ever go wrong. That includes covering every condition on every line of code. Most tools developed to work with the programming language and software architecture need to be formally verified as well. On top of that, costly testing needs to be done to evaluate any changes. This creates a lot of inertia and means adding new features much less a complete overhaul of an existing product or starting from scratch for a new product is often not cost-effective.

Simply put, formal verification and exhaustive testing are not the right solutions given the realities of the current market. Many of today's start-ups and corporations need to bring their ideas into the market and iterate through many versions very rapidly. Customers also expect new features and upgrades every year. It is not possible to expect companies like SAMSUNG or Google to develop their next smart thermostat in the same time-line and using the same budgets that are expected for the development of airplane parts. Given the reliability requirements of these new products, there is a need to embrace new CPS design paradigms where faults and security vulnerabilities are the norms rather than an anomaly. Such imperfections must be assumed to exist in every system and component. Faults and vulnerabilities must be handled safely and the CPS must be able to recover from them at runtime.

My goal in this work is to demonstrate that it is possible to create low-cost, safe-by-design architectures for CPS that allow quick iterations and frequent upgrades. The architectures and techniques introduced in this dissertation do not rely on the testing and full verification. Instead, they enforce safety at the highest level of the system and extend guaranteed safety from a few certified components to the entire platform. These solutions are carefully curated to utilize unverified components and provide guaranteed performance.

In Chapter 4, I present a restart-based fault-tolerant design that can be applied to a wide variety of embedded controller to ensure the safety of the physical plant under control. In the same chapter, we offer a complementary approach enables the use of this design in plants with non-linear dynamics. Applying this approach CPS with large number of dimensions can be challenging at times. In Chapter 5, I provide an alternative restart-based approach such that safe-by-construct architecture is not impacted by the complexity of physical dynamics. When a CPS is under attack, many of the assumptions necessary for the previous techniques do not hold anymore. Consequently, those architectures will not provide safety guarantees

[1]In avionics, for example, Level A certified software often requires one failure per 10^9 operation hours.

under attacks. In chapter 6, we study safety in presence of an intelligent adversary and lay out a design pattern that can maintain the physical safety. Finally, in Chapter 7, I investigate the safety of multiple coordinating CPS where the communication channels are not reliable. We demonstrate the conditions necessary to maintain safety and progress and incorporate them in a case-study.

In the rest of this chapter, I review the contribution made in each chapter of this dissertation.

1.1 SAFETY OF SINGLE NODE CPSS

The work presented in Chapter 4 and 5 of this dissertation provides techniques and designs that allow utilizing unverified software components to implement a CPS controller with guaranteed safe performance. Presented designs rely on a critical observation that restarting a computing system and reloading a fresh image of all the software (*i.e.,* RTOS, and applications) from a read-only source appears to be an effective approach to recover the system from unexpected faults.

Restartable Controller Design

Chapter 4 proposes a software/hardware co-design methodology that provides *fault-tolerance* and *liveliness* guarantees only using one commercial-off-the-shelf (COTS) computing platform. I provide a procedure for the synthesis of abstraction-based correct-by-construction controllers for linear and nonlinear physical systems that enables the entire computing system to be safely restarted at runtime. This controller can keep the control system inside a subset of safety region, only by updating the actuator input at least once after every system restart.

The key contributions of this chapter are:

- Construction of formally verified base controllers for safety-critical applications with linear and nonlinear physical components which guarantee safe full system restart for application and system level fault tolerance.
- Tolerating application-level faults as well as system-level faults using only one COTS processing unit.
- Empirical validation of both the practicality of our proposed design and the safety guarantees through fault-injection testing on a prototype controller for the nonlinear inverted pendulum system and a 3-degree-of-freedom helicopter.

Schedulability Analysis of Restarts

In Chapter 5, I propose another restart-based fault-tolerant design that is not impacted by the complexity or number of dimensions of the dynamics of the physical plant under control.

The controllers are constructed following the principles of Simplex architecture [2]. As a result, the task set running on the platform can be divided into two categories of critical and non-critical tasks such that the timely execution of critical task set is sufficient for maintaining the system safety. I propose the following: as soon as a fault that disrupts the execution of critical software tasks is detected, the entire system is restarted. After a restart, all the safety-critical tasks that were impacted by the restart are re-executed. If restart and re-execution of critical tasks can be performed *fast enough*, i.e., such that timing constraints are always met in spite of task re-execution, the physical system will remain oblivious to and will not be impacted by the occurrence of faults.

From a scheduling perspective, safety is guaranteed if safety controller tasks have enough CPU cycles to re-execute and finish before their deadlines in spite of restarts. In this chapter, I present the conditions for a periodic task set to be schedulable in the presence of restarts and re-execution. I assume that when a restart occurs, the task instance executing on the CPU and any of the tasks that were preempted before their completion will need to re-execute after the restart. In particular, I make the following contributions:

- I propose a design based on Simplex Architecture that enables fault-recovery via restarts and can be implemented on a single processing unit;
- I derive the response time analysis under fixed-priority with fully preemptive and fully non-preemptive disciplines in the presence of restart-based recovery and discuss the pros and cons of each one;
- I propose response time analysis of fixed-priority scheduling in the presence of restarts for tasks with preemption thresholds [3] and non-preemptive ending intervals [4] to improve feasibility of task sets;
- To evaluate the practicality of the design, I perform a proof-of-concept implementation on a 3-DOF helicopter and test the system against various types of faults.

1.2 SECURITY OF SINGLE NODE CPS

There are always unforeseen vulnerabilities that enable intruders to bypass the security mechanisms and gain administrative access to the controllers. Once an attacker gains such

access, all bets are off with regards to the safety of the physical subsystem. In Chapter 6, I *develop analytical methods that can formally guarantee the baseline safety of the physical plant even when the controller unit's software has been entirely compromised.* The main idea is to carry out consecutive evaluations of physical safety conditions, inside secure execution intervals, separated in time such that an attacker with full control will not have enough time to destabilize or crash the physical plant in between two consecutive intervals. These intervals are referred by Secure Execution Interval (SEI). The time between consecutive SEIs is dynamically calculated in real time, based on the mathematical model of the physical plant and the current state. The key to providing such formal guarantees is to make sure that each SEI takes place before an attacker can cause any physical damage.

I discuss two different approaches to achieve this goal; *(i) restart-based implementation* Under which, control platform is restarted in each cycle, and the uncompromised image of the controller software is reloaded from read-only storage *(ii) TEE-based implementation* which utilizes Trusted Execution Environments (TEE) such as ARM TrustZone [5] or Intel's Trusted Execution Technology (TXT) [6] that are available in some hardware platforms. This design can significantly improve the applicability of our method to physical plants with faster dynamics.

In summary, the contributions of the work are:

- A design method is introduced for embedded control platforms with *formal guarantees on the base-line safety of the physical subsystem* when the software is under attack.
- A restart-based design implementation is proposed that enables trusted computation in an untrusted environment using platform restarts and common-off-the-shelf (COTS) components, without requiring chip customizations or specific hardware features.
- An alternative design implementation is proposed using TEE features that eliminates the restarting overhead and enables the core safety-guarantees to be provided on more challenging physical plants.
- I implemented and tested our approach against attacks through a prototype implementation for a realistic physical plant and a hardware-in-the-loop simulation. I will compare both design implementation options and illustrate their use cases.

1.3 GUARANTEED SAFETY IN DISTRIBUTED CPS

In this chapter, I investigate the safety of distributed CPSs – safety premise of each node is defined with regards to the environment as well as the state of other nodes in the

system. Distributed CPSs combine networked communication along with interactions with the physical world. In particular, I consider a CPS scenario consisting of several embedded computing components each interacting and sensing the physical world and communicating with a central coordinator over an unreliable channel, such as wireless or the Internet. A distributed CPS is considered globally safe if and only if all the nodes are safe. These low-level controllers attempt to accomplish some task in a coordinated fashion. Since the physical world is being manipulated, it is essential that the supervisory control logic is carefully designed and satisfies strict safety requirements. This system is difficult to reason about because both (1) the communication layer can experience unbounded message delays and drops, and (2) the dynamics of the physical world are represented by interacting relationships in a continuous space.

In the context of a distributed CPS, a designer is typically interested in two properties: safety and progress. Proof of safety will guarantee that the system will never enter an undesirable state. I formally specify safety as a predicate on the variables of the agents of the distributed CPS which is true at all times (a safety invariant). The notion of progress that I consider is that roughly speaking, all the agents will receive and follow the desired goal command in finite time. The ultimate guarantee that this design provides is that the system will remain safe at all times (even if the network fails) while being able to meet the progress property as long as the communication network is functioning.

In this chapter, I propose a Runtime Command Monitor interposed between the supervisory control logic and the network. If the supervisory control logic attempts to send control commands which, for any amount of message delay, can lead to a system state that violates the safety predicate, the Runtime Command Monitor will reject commands which could lead to unsafe states. It will be demonstrated that this design results in a fail-safe system.

The main contributions are as follows:

- I prove that run-time properties provide necessary and sufficient conditions for safety in a distributed CPS system. By encoding these checks into a Runtime Command Monitor, a fail-safe system can be developed (Section 7.1).

- The proposed run-time properties require computing the reachability of the system online, which can be an expensive operation. Through a combination of *reachability reduction transformations* and *input and state enumeration*, this operation is performed off-line (Section 7.3).

- I provide sufficient conditions for providing progress guarantees. This requires

constructing a chain of compatible actions, as well as a network which eventually delivers packets that are sent. (Section 7.2)

- The presented approach is applied to a simulated system of vehicles moving in a shared environment, where the runtime command monitor prevents vehicle collisions. (Section 7.4)

CHAPTER 2: RELATED WORK

Hereby, I review some of most relevant work in the literature.

2.1 SINGLE NODE CPS SAFETY

The notion of restarting as a means of recovery from faults and improving system availability was previously studied in the literature. Most of the previous work, however, target traditional *non*-safety-critical computing systems such as servers and switches. These approaches are generally divided into two categories, *viz., i*) *revival*, reactively restart a failed component and *ii*) *rejuvenation*, prophylactically restart functioning components to prevent state degradation [7]. Our designs, as described in Chapters 4 and 5 of this dissertation, fit in the former category. Authors in [8] introduce recursively restartable systems as a design paradigm for highly available systems and use a combination of revival and rejuvenation techniques. Earlier literature [9, 10, 11] illustrates the concept of microreboot which consists of having fine-grain rebootable components and trying to restart them from the smallest component to the biggest one in the presence of faults. The works in [12, 13, 14] focus on failure and fault modeling and try to find an optimal rejuvenation strategy for various systems. In this context, our previous work in Reset-Based Recovery [15] was an attempt to utilize restarting as a recovery method for computing systems in safety-critical environments. In an earlier work [15], we used System-Level Simplex architecture and proposed to restart only the complex subsystem upon the occurrence of faults. This is feasible because the safety subsystem runs on a dedicated hardware unit and is not impacted by the restarts in the complex subsystem. The approach of the current paper is significantly different and uses only one hardware unit.

The concept of utilizing an unverified, complex controller along with a simple, verified safety controller for fault tolerance was initially proposed as Simplex Architecture in [16, 2, 17]. The Simplex Architecture was developed as an approach to increase system safety for individual Linear Time-Invariant (LTI) control systems, by filtering commands from an untrusted controller and switching over to a safe backup mode. It deploys two controllers: *(i)* a high-performance (yet unverifiable) controller and *(ii)* a high-assurance, formally verified, safety controller. A decision module (formally verifiable) is used to take over the control in the case that the high-performance controller is pushing the physical system beyond a pre-computed safety envelope.

In earlier simplex designs, fault tolerance was achieved in one of two ways. In some of

these designs such as [16, 18, 2, 17, 19], all three components (safety controller, complex controller, and decision unit) share the same computing hardware (processor) and software platform (OS, middleware). As a result, these designs only protect the safety against the faults in the application logic of the complex controller and do not guarantee the correct behavior in the presence of system-level faults.

Some Simplex-based designs such as System-Level Simplex [20], Secure System Simplex Architecture (S3A)[21], and other variants [22] run the safety controller and the decision logic on an isolated, dedicated hardware unit. By doing so, the trusted components are protected from the faults in the complex subsystem. However, exercising System-Level Simplex design on most COTS multicore platforms/SoC (system on chip) is challenging. The majority of commercial multicore platforms are not designed to achieve strong inter-core fault isolation due to the high-degree of hardware resource sharing. For instance, a fault occurring in a core with the highest privilege level may compromise power and clock configurations of the entire platform. To achieve full isolation and independence, one has to utilize two separate boards/systems. In contrast, the approaches proposed in Chapters 4 and 5 need only one processor and tolerate system-level faults.

One way to achieve fault-tolerance in real-time systems is to use time redundancy. Using time redundancy, whenever a fault leads to an error, and the error is detected, the faulty task is either re-executed or a different logic (recovery block) is executed to recover from the error. It is necessary that such recovery strategy does not cause any deadline misses in the task set. Fault tolerant scheduling has been extensively studied in the literature. Hereby, those works that are more closely related are surveyed.

A feasibility check algorithm under multiple faults, assuming EDF scheduling for aperiodic preemptive tasks is proposed in [23]. An exact schedulability tests using checkpointing for task sets under fully preemptive model and transient fault that affects one task is proposed in [24]. This analysis is further extended in [25] for the case of multiple faults as well as for the case where the priority of a critical task's recovery block is increased.

In [26], authors propose the exact feasibility test for fixed-priority scheduling of a periodic task set to tolerate multiple transient faults on uniprocessor. In [27] an approach is presented to schedule under fixed priority-driven preemptive scheduling at least one of the two versions of the task; simple version with reliable timing or complex version with potentially faulty.

Authors in [28] consider a similar fault model to ours, where the recovery action is to re-execute all the partially executed tasks at the instant of the fault detection *i.e.,* executing task and all the preempted tasks. This work only considers preemptive task sets under rate monotonic and shows that single faults with a minimum inter-arrival time of largest period in the task set can be recovered if the processor utilization is less than or equal to 50%. In

[29], the authors investigate the feasibility of task sets under fault bursts with preemptive scheduling. The recovery action in this work is to re-execute the faulty job along with all the partially completed (preempted) jobs at the time of fault detection.

Most of these works are only applicable to transient faults (*e.g.,* faults that occur due to radiation or short-lived HW malfunctions) that impact the task and do not consider faults affecting the underlying system. Additionally, most of these works assume that an online fault detection or acceptance test mechanism exists. While this assumption is valid for detecting transient faults or timing faults, detecting complex system-level faults or logical faults is non-trivial.

2.2 SINGLE NODE SECURITY

There is a considerable number of techniques in the area of fault-tolerant CPS design that focuses on protecting the physical components in the presence of faults[1]. Although similar, there are fundamental differences between protecting against faults vs. protecting against an intelligent adversary. In what follows I review some of the papers and elaborate the differences and similarities.

As mentioned earlier, Simplex architecture [2] is a well known fault-tolerant design for CPS. It deploys two controllers: *(i)* a high-performance (yet unverifiable) controller and *(ii)* a high-assurance, formally verified, safety controller. A decision module (formally verifiable) is used to take over the control in the case that the high-performance controller is pushing the physical system beyond a pre-computed safety envelope. A few variants of Simplex design exist; some use a varying switching logic [30, 31] while others utilize a different safety controller [32, 33]. Nevertheless, all these designs assume that only a subset of the software misbehaves (for instance, they assume that switching unit cannot misbehave), which is invalid when the systems are under attack, and no other mechanism – such as restarts or TEE features are employed.

Another variant of the Simplex architecture is System-Level Simplex [20] where the safety controller and the decision module run on dedicated hardware to isolate them from any fault or malicious activities on the complex controller (*i.e.,* the high-performance controller). Techniques based on this architecture [20, 15, 34, 35] guarantee the safety of the physical plant even when the complex controller is under attack. However, implementing the

[1]Where the safety invariants of the physical plant must be preserved despite the possible implementation and logical errors in the software. Here, 'faults' refer to bugs in the software implementations. Another definition for faults exists that includes physical problems (*e.g.,* broken sensors/actuators/etc) – such faults are out of scope of this work.

System-Level Simplex design on most COTS platforms is challenging since most commercial multicore platforms are not designed to support strong inter-core isolation (due to the high degree of hardware resource sharing). For instance, an adversary residing in the high-privileged core may compromise power and clock configurations of the entire system. Hence, full isolation can only be achieved by utilizing two separate boards.

Trusted hardware features are commonly employed in the literature to achieve security goals. Some works have deployed the Trusted Platform Module (TPM) to build trusted computing environments on servers and hypervisors [36, 37, 38]. ARM TrustZone has been utilized in recent literature [39, 40, 41] to implement security monitors in the secure world. Authors in [42], leverage TrustZone and propose TZ-RKP to protect the integrity of the operating system kernel running in the normal, non-secure world. The analytical framework proposed in Chapter 6 could be combined into these techniques to develop a diverse set of CPS platforms that can provide physical safety guarantees.

Restart-based recovery is previously explored in some of the aforementioned Simplex-based works [34, 15]. Specifically, these works restart the isolated, dedicated complex controller unit – equivalent to the mission controller. Restarting the complex controller while a safety controller running on separate hardware maintains the safety during the restart is more straightforward than restarting the entire platform. Another Simplex-based work in which the authors use a single hardware unit implements full-system restarts [32]. Nevertheless, this work assumes that the safety controller and the decision module may not be compromised and are always correct. Again, this assumption is invalid in the security context, and the physical safety cannot be guaranteed when the system is under attack.

There is a trend in systems dependability that applies the concepts and mechanisms of fault tolerance in the security domain, intrusion tolerance (or Byzantine fault tolerance) [43, 44]. These works advocate for designing intrusion-tolerant systems rather than implementing prevention against intrusion. Many works in intrusion-tolerant systems have targeted distributed services in which replication and redundancy are feasible. Their goals are mainly to ensure the availability of the system service even if some of its nodes are compromised. One paper proposes to proactively restore the system code from a secure source to eliminate any potential transformations carried out by an attacker [43]. With proactive recovery, the system can tolerate up to f faults/intrusions, as long as no more than f faults occur in between rejuvenations. In [45], the authors propose a general hybrid model for distributed asynchronous systems with partially synchronous components, named *wormholes*. In [46], the authors take wormholes as a trusted secure component which proactively recovers the primary function of the system. The authors suggest that such a component can be implemented as a separate, tamper-proof hardware module in which the separation

is physical; or it can be implemented on the same hardware with virtual separation and shielding enforced by software. A proactive-reactive recovery approach is introduced in [47] (built on top of [46]) that allows correct replicas to force the recovery of a faulty replica. While these techniques are useful for some safety-critical applications such as supervisory control and data acquisition (SCADA), they may not be directly applicable to safety-critical CPS. Potentially, a modified version of these solutions might be utilized to design a cluster of replicated embedded controllers in charge of a physical plant.

2.3 DISTRIBUTED CPS SAFETY

Networked control systems have been employed in a variety of industrial automation applications. Industrial wireless protocols and products have been developed as replacements for wired control systems [48, 49]. These were made not only to reduce costs due to materials (wiring), installation and wire maintenance, but also provide benefits in flexibility by allowing easy modification to the existing communication infrastructure. One benefit of using these solutions is that they strive to reduce (but cannot eliminate) problems arising from communication delay and packetloss when wireless is used in industrial control systems.

A network extension of Simplex has also been developed [50]. This work extended the Simplex approach to Linear Parameter Varying (LPV) systems, and incorporated network delays into the design. However, the analysis requires having a fixed upper bound on communication delay with no packetloss, which cannot be guaranteed under wireless communication. Our guarantee of safety holds without a fixed upper bound on communication delay and we allow unrestricted packet loss to occur.

Our approach for safety of distributed CPS draws inspiration from the NASS framework developed to provide safety for medical systems communicating over wireless [51]. This system uses discrete dynamics with formal safety properties in a supervisory control system over wireless. Each command message includes a backup command vector, which is used if no further commands arrive. A safety filter provides protection from faults in the high-level control. This filter needs to reason about the worst-case packet delivery combinations, which in the case of the considered discrete system involves model-checking the possible combinations of packet reception and agent states.

Run-time approaches have been considered to create verified systems [52]. The advantage of this approach is, since at runtime some of the variables are known, only a smaller state space needs to be considered. This is also the argument we make when advocating the design of the Runtime Command Monitor. Real-time ways of computing reachability have also been recently developed [53], which are quick enough to run in within a control loop

(on the order of tens of milliseconds). Such methods can reduce the burden of enumeration from the proposed work, but at the cost of extremely limited runtime, which can increase error in the computed over-approximation of the reachable set of states.

For partially synchronous systems, where messages get bounded nondeterministic delays or dropped, a sufficient condition for verifying convergence properties has been established [54]. The sufficient conditions require that (i) messages get delivered infinitely often and (ii) there exist some invariant neighborhood topology of the system satisfying a Lyapunov-type property.

For asynchronous distributed systems, where messages get nondeterministic but bounded delay, a static approach for reasoning about the convergence of an asynchronous system has been proposed [55]. The approach shows that under some additional assumptions about the shape of the sublevel sets of the Lyapunov function, if convergence occurs in perfect communication, where messages get delivered instantly without dropping, convergence will also occur in the corresponding synchronous system.

Similar problems also exist in the area of air traffic management where at any given moment each, all the airplanes need to maintain a minimum distance from each other and avoid collision. On a typical day, more than 40,000 commercial flights operate within the US airspace [56]. In order to efficiently and safely route this air traffic, current traffic flow control relies on a centralized, hierarchical routing strategy that performs flow projections ranging from one to six hours. As an aircraft travels through a given airspace division, it is monitored by the one or more air traffic controllers responsible for that division. The controllers monitor this plane and give instructions to the pilot. As the plane leaves that airspace division and enters another, the air traffic controller passes it off to the controllers responsible for the new airspace division.

CHAPTER 3: PRELIMINARIES

In this section, I provide definitions and background on some of the concepts used throughout the rest of the dissertation.

3.1 SIMPLEX DESIGN

Our proposed approach is designed for the control tasks that are constructed following Simplex verified design guidelines [16, 2, 17]. In the following, I review Simplex design concepts which are essential for understanding the methodology of this chapter. The goal of original Simplex approach is to design controllers, such that the faults in controller software do not cause the physical plant to violate its safety conditions.

Definition 3.1. *Admissible and Inadmissible States: States that do not violate any of the operational constraints of the physical plant are referred to as admissible states and denoted by $\mathcal{S}$. Likewise, those states that do violate the constraints are referred to as inadmissible states and denoted by $\mathcal{S}'$.*

Operational limits and safety constraints of the physical system dictate what $\mathcal{S}$ is and it is outside of our control.

Under Simplex Architecture, each controlled physical process/component requires a safety controller, a complex controller, and a decision module. In the following, the properties of each component are defined.

Definition 3.2. *Recoverable states: are defined with regards to a given Safety Controller (SC) and denoted by $\mathcal{R}$. $\mathcal{R}$ is a subset of $\mathcal{S}$ such that if the given SC starts controlling the plant from the state $x \in \mathcal{R}$, all future states will remain admissible.*

Definition 3.3. *Safety Controller is a controller for which a subset of the admissible states called recoverable states exists with the following property; If the safety controller starts controlling the plant from one of those states, all future states will remain admissible. The set of recoverable states is denoted by $\mathcal{R}$. Safety controller is formally verified i.e., it does not contain logical or implementation errors.*

Definition 3.4. *Complex Controller or Mission Controller is the main controller task of the system that drives the plant towards mission set points. However, it is unverified i.e., it may contain unsafe logic or implementation bugs. As a result, it may generate commands that force the plant into inadmissible states.*

Definition 3.5. *Decision Module includes a switching logic that can determine if the physical plant will remain safe (stay within the admissible states) if the control output of complex controller is applied to it.*

Note: Safety Controller is only capable of keeping plant safe and does not push it towards its goal/mission. A meaningful system, therefore, cannot run under SC at all times and requires another *mission controller* to make progress.

There are multiple approaches to design a verified safety controller and decision module. Ideally, we would want a SC that can stabilize the system from all the admissible states $\mathcal{S}$. However, it is not usually possible. The first proposed way is based on solving linear matrix inequalities [57], which has been used to design Simplex systems as complicated as automated landing maneuvers for an F-16 [58]. According to this approach, safety controller is designed by approximating the system with linear dynamics in the form: $\dot{x} = Ax + Bu$, for state vector x and input vector u. In this approach, *safety constraints* are expressed as linear constraints in the form of linear matrix inequalities. These constraints, along with the linear dynamics for the system, are the inputs to a convex optimization problem that produces both linear proportional controller gains K, as well as a positive-definite matrix P. The resulting linear-state feedback controller, $u = Kx$, yields closed-loop dynamics in the form of $\dot{x} = (A + BK)x$. Given a state x, when the input Kx is used, the P matrix defines a Lyapunov potential function ($x^T Px$) with a negative-definite derivative. As a result, the stability of the physical plant is guaranteed using Lyapunov's direct or indirect methods. Furthermore, matrix P defines an ellipsoid in the state space where all safety constraints are satisfied when $x^T Px < 1$. If sensors' and actuators' saturation points were provided as constraints, the states inside the ellipsoid can be reached using control commands within the sensor/actuator limits.

In this way, when the gains K define the safety controller, the ellipsoid of states $x^T Px < 1$ is the set of recoverable states $\mathcal{R}$. This ellipsoid is used to determine the proper switching logic of the decision module. As long as the system remains inside the ellipsoid, any unverified, complex controller can be used. If the state approaches the boundary of the ellipsoid, control can be switched to the safety controller which will drive the system towards the equilibrium point where $x^T Px = 0$.

An alternative approach for constructing a verified safety controller and decision module is proposed in [30]. Here, safety controller is constructed similar to the above approach [57]. However, a novel switching logic is proposed for decision module to decide about the safety of complex controller commands. Intuitively, this check is examining what happens if the complex controller is used for a single control interval of time, and then the safety controller

is used thereafter. If the reachable states contain an inadmissible state (either before the switch or after), then the complex controller cannot be used for one more control interval. Assuming the system starts in a recoverable state, this guarantees it will remain in the recoverable set for all time.

A system that adheres to this architecture is guaranteed to remain safe only if safety controller and decision module execute correctly. In this way, the safety premise is valid only if safety controller and decision module execute in every control cycle. Original Simplex design, only protects the plant from faults in the complex controller. For instance, if a fault in the RTOS crashes the safety controller or decision module, safety of the physical plant will get violated.

3.2 REAL-TIME REACHABILITY

For runtime computation of reachable states of a plant within a future time, this work makes use of a real-time reachability tool (Bak *et al.* [30]). This low-cost algorithm is specifically designed for *embedded systems with real-time constraints and low computation power*.

Note that constructing a safety controller similar to that specified in section 3.1 (*e.g.,* having a recoverable region where any trajectory starting from that region will stay within that region) is generally not possible for non-linear systems. However, for specific classes of non-linear systems, our approach will be applicable if: *(i)* a safety controller with the properties mentioned above can be constructed and *(ii)* we can define a function that returns the minimum and maximum derivative in each dimension given an arbitrary box in the state space. This technique can also handle hybrid systems where the state invariants are disjoint and cover the continuous state $\mathbb{R}^n$, there are no reset maps in the transitions between discrete states and the state invariants define the guards of incoming transitions. In these piecewise systems, the state of the hybrid automaton can be determined solely by the continuous state; although separate differential equations can be used in various parts of the state space. This algorithm requires that the derivatives are defined in the entire state space and that they are bounded.

This technique uses the mathematical model of the dynamics of the plant and a n-dimensional box to represent the set of possible control inputs and the reachable states. A set of *neighborhoods*, $N[i]$ are constructed around each $face_i$ of the tracked states with an initial width. Next, the maximum derivative in the outward direction, d_i^{max}, inside each $N[i]$ is computed. Then, crossing time $t_i^{crossing} = width(N[i])/d_i^{max}$ is computed over all neighborhoods and the minimum of all the $t_i^{crossing}$ is chosen as time to advance, t^a. Finally,

every face is advanced to $face_i + d_i^{max} \times t^a$. For further details on inward neighborhood versus outward neighborhoods, and the choosing of neighborhood widths and time steps refer to [30]. In this algorithm a parameter called `reach-time-step` is used to control neighborhood widths. This parameter lets us tune the total number of steps used in the method, and therefore alter the *total runtime* to compute the reachable set. This allows us to cap the total computation time of the reachable set – which is essential in any real-time setting.

Moreover, authors have demonstrated that this algorithm is capable of producing useful results within very short computation times *e.g.,* result achieved with computation times as low as $5ms$ using embedded platforms [30]. All these features make this approach a suitable tool for our target platforms as well.

3.3 NOTATIONS

The symbols $\mathbb{N}$, $\mathbb{N}_0$, $\mathbb{Z}$, $\mathbb{R}$, $\mathbb{R}^+$, and $\mathbb{R}_0^+$ denote the set of natural, nonnegative integer, integer, real, positive, and nonnegative real numbers, respectively. We use $\mathbb{R}^{n\times m}$ to denote a vector space of real matrices with n rows and m columns. The identity matrix in $\mathbb{R}^{n\times n}$ is denoted by I_n and zero matrix in $R^{n\times m}$ is denoted by $0_{n\times m}$. For $a, b \in (\mathbb{R} \cup \{-\infty, \infty\})^n$, $a \leq b$ component-wise, the closed hyper-interval is denoted by $[\![a, b]\!] := \mathbb{R}^n \cap ([a_1, b_1] \times [a_2, b_2] \times \ldots \times [a_n, b_n])$. We identify the relation $R \subseteq A \times B$ with the map $R : A \to 2^B$ defined by $b \in R(a)$ iff $(a, b) \in R$. Given a relation $R \subseteq A \times B$. $R^{-1} = \{(b, a) \in B \times A \mid (a, b) \in R\}$. $Q \circ R$ denotes the composition of maps Q and R, $Q \circ R(x) = Q(R(x))$. The map R is said to be strict when $R(a) \neq \emptyset$ for every $a \in A$.

3.3.1 Control Systems

Definition 3.6 (Nonlinear control systems)**.** *A nonlinear control system is a tuple* $\Sigma = (\mathbb{R}^n, \mathsf{U}, \mathcal{U}, f)$*, where* $\mathbb{R}^n$ *is the state space;* $\mathsf{U} \subseteq \mathbb{R}^p$ *is a bounded input set;* $\mathcal{U}$ *is a subset of the set of all functions of time from* $\mathbb{R}_0^+$ *to* U*; and* f *is a locally Lipschitz continuous map from* $\mathbb{R}^n \times \mathsf{U}$ *to* $\mathbb{R}^n$*.*

The trajectory ξ is said to be a solution of Σ if there exists $\upsilon \in \mathcal{U}$ satisfying:

$$\dot{\xi}(t) = f(\xi(t), \upsilon(t)), \tag{3.1}$$

for any $t \in \mathbb{R}_0^+$. It should be emphasized that the locally Lipschitz continuity assumption on f ensures existence and uniqueness of solution ξ [59]. We use notation $\xi_{x,\upsilon}(t)$ to denote the value of solution at time t under the input signal υ and starting from initial state $x = \xi_{x,\upsilon}(0)$.

Definition 3.7 (Linear control systems)**.** *A linear, time-invariant control system is a special case of non-linear systems where f is defined as $\dot{\xi}(t) = A\xi(t) + B\upsilon(t)$ where $A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times m}$.*

3.3.2 Formulating Safety

In physical systems, maintaining all states and control inputs within safe limits is very important in order to avoid damages to the system itself or the environment around it. In this chapter, the safety region $\mathcal{S}$ is defined as a subset of the state space. For example, one can define it as:

- polytope $\mathcal{S} = \{x \in \mathbb{R}^n \mid H_x \cdot x \leq h_x\}$ parameterized by $H_x \in \mathbb{R}^{q\times n}$, $h_x \in \mathbb{R}^q$, or
- ellipsoid $\mathcal{S} = \{x \in \mathbb{R}^n \mid \|L(x - y)\|_2 \leq 1\}$ parameterized by $L \in \mathbb{R}^{n\times n}$ and $y \in \mathbb{R}^n$.

In a similar way, the bounds on operational ranges of control inputs can be expressed as:

- polytope $\mathcal{S}_u = \{u \in \mathsf{U} \mid H_u \cdot u \leq h_u\}$ parameterized by $H_u \in \mathbb{R}^{\bar{q}\times p}$ and $h_u \in \mathbb{R}^{\bar{q}}$, or
- ellipsoid $\mathcal{S}_u = \{u \in \mathsf{U} \mid \|L_u(u - \overline{u})\|_2 \leq 1\}$ parameterized by $L_u \in \mathbb{R}^{p\times p}$ and $\overline{u} \in \mathsf{U}$.

The nonlinear control system Σ is said to be safe if the states of the system remain inside $\mathcal{S}$ using only the control commands in $\mathcal{S}_u$.

3.3.3 Reachable Set

Consider a nonlinear control system as in (3.1) and a set $X_0 \subset R^n$. The reachable set of states that can be reached starting from set X_0 under input signal υ at time τ is given by $\mathbf{Reach}_\tau(X_0, \upsilon) := \bigcup_{x\in X_0} \xi_{x,\upsilon}(\tau)$. We use notation $\mathbf{Reach}_{[0,\tau]}(X_0, \upsilon)$ to denote the reachable set that can be reached starting from X_0 under input signal υ up to time τ and can be defined as $\mathbf{Reach}_{[0,\tau]}(X_0, \upsilon) := \bigcup_{t\in[0,\tau]} \mathbf{Reach}_t(X_0, \upsilon)$. We use the notation $\overline{\mathbf{Reach}}_\tau(X_0, \upsilon)$ to denote an over-approximation of the set $\mathbf{Reach}_\tau(X_0, \upsilon)$.

CHAPTER 4: SINGLE NODE CPS SAFETY THROUGH CONTROLLER DESIGN

In this chapter, I propose a novel approach to design a controller that provides safety guarantees for the physical component in the presence of application-level and system-level faults. Our solution provides *fault-tolerance* and *liveliness* guarantees using only commercial-off-the-shelf (COTS) computing platform. This approach uses *full system restart* to recover from such application and system-level faults. Restarting in the safety-critical environment is very challenging and this chapter provides a procedure for the synthesis of abstraction-based correct-by-construction controllers for linear and nonlinear physical systems that enables the entire computing system to be safely restarted at runtime. This controller can keep the control system inside a subset of safety region, only by updating the actuator input at least once after every system restart. In this chapter, we refer to this controller as *Base Controller* (BC).

Restarting a system is an effective approach for recovery from unknown faults at runtime, with a very predictable outcome. As soon as a fault occurs that disrupts the execution of critical software components, a hardware watchdog timer (WD) restarts the system. After a restart, a fresh image of all the software (middleware, RTOS and applications) is loaded from a read-only storage that recovers the system into an operational state. Prior to this work, restarting was proposed as a way to increase the availability of non-safety critical systems [8, 9, 10, 11, 12, 13, 14]. In addition, some works investigated the partial restarting of safety-critical systems using extra hardware [15, 20]. To the best of our knowledge, this is the first work that proposes safe restarting of the entire system in a safety-critical environment that contains nonlinear physical components.

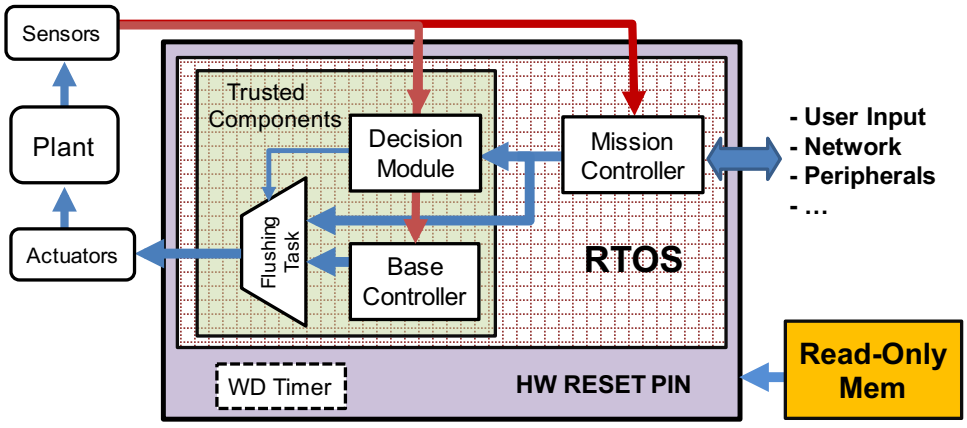


Figure 4.1: The logical view of the proposed design.

Having only BC and the WD mechanism (to enable restarting), allows the system to remain safe, tolerate faults and recover from them. However, it does not aid in making progress towards its mission goal. To address this issue, BC is complemented with a *Mission Controller* (MC) (*e.g.*, a neural network) and a *Decision Module* (DM). The MC is an unverified, high-performance, complex controller that drives the system towards the mission setpoints. It may contain unsafe logic or bugs that jeopardize safety. To maximize the progress towards the mission goals, in every control cycle, DM checks the MC command. If it satisfies the safety requirements, DM allows it to be sent to the actuators. Otherwise, BC command is applied to the system. By doing so, MC drives the system for as long as possible, and, whenever it is not possible, BC takes the control.

In the proposed design, the only components that need to be verified for correct functionality are BC, DM and Flushing Task. Any fault in the system software (System-Level or Application-Level) that results in a fail-silent failure (also known as fail-stop) of these two components leads to WD triggering a system-wide restart and recovery. However, our design does not protect the system from faults that alter the logic of BC or DM at execution times. In summary, this design enables the system to provide formal safety guarantees by verifying only the correctness of BC, DM and a Flushing Task – we will discuss this later – instead of entire MC, RTOS and middleware.

The key contributions of this chapter are:

- Construction of formally verified base controllers for safety-critical applications with nonlinear physical components which guarantee safe full system restart for application and system level fault tolerance.
- Tolerating application-level faults as well as system-level faults using only one COTS processing unit.
- Empirical validation of both the practicality of our proposed design and the safety guarantees through fault-injection testing on a prototype controller for the nonlinear inverted pendulum system and a 3-DOF helicopter.

4.1 DESIGN APPROACH

As depicted in Figure 4.1, the proposed design consists of three main components; Base Controller (BC), Mission Controller (MC) and Decision Module (DM).

The BC is a verified, reliable controller that is only concerned with safety. It does not make progress towards the mission set points of the system (*i.e.,* it does not provide *liveness*). The

MC, on the other hand, is the main controller which is concerned with the mission-critical requirements. This controller may have complex logic, can be changed and upgraded while the system is running and may even contain unsafe logic and bugs. As an example, MC may be a neural network resulted from machine learning techniques.

All the components of the system run on top of the RTOS. The length of one control cycle of the system is τ_c. The kth control cycle refers to the period $[(k-1)\tau_c, k\tau_c]$, where $k \in \mathbb{N}$. The cycles count and the time origin are restarted after every system restart. Therefore, $k = 1$ always refers to the first cycle after the latest system restart. Furthermore, we assume that the length of the restart time[1], *i.e.*, τ_r, of the system is an integer multiple[2] of τ_c (*i.e.*, $\tau_r = m\tau_c$, where $m \in \mathbb{N}$). While the system is running, sensor values are sampled at $t = k\tau_c - \epsilon$ where $\epsilon \ll \tau_c$ and actuator inputs are updated at $t = k\tau_c$.

In every control cycle, after MC runs and generates its output u_{mc}, DM evaluates the safety requirements under u_{mc} and decides whether u_{mc} can be applied to the actuators. Then, DM writes its output, along with the corresponding MC command and a timestamp (cycle number) to a fixed memory address.

At the end of the control cycle, at time $k\tau_c - \epsilon$ after sensors are sampled, BC runs and generates u_{bc}. Then a flushing task retrieves u_{mc}, u_{bc}, the decision of DM and the corresponding timestamp from the memory. If the timestamp matches with the current cycle number, k, it updates the actuator commands with u_{mc} or u_{bc} based on the decision of DM and resets watchdog timer (WD). Non-matching timestamps indicate that one or both of the DM and BC tasks did not execute or missed their deadlines. In such cases, the flushing task does not update the WD. Consequently, WD expires at $t = k\tau_c$ and triggers a restart. Note that as a result of this mechanism, restarts are only triggered at times $t = k\tau_c$ and do not occur in between control cycles. The steps are illustrated in Figure 4.2.

In the rest of this section, the assumptions and the fault model of the system are discussed followed by a description of the properties of the BC and how it is able to safely handle the restarts. Finally, the safe switching logic of the DM is presented.

4.1.1 Assumptions and Fault Model

In this chapter, following assumption are made about the faults and the components of the system.

- Hardware faults are not a concern in this chapter and we assume that hardware is

[1]It includes the time for reloading the bootloader, OS, and the applications from the read-only storage, initializing the necessary sensors and peripheral, booting the OS and executing the control applications.

[2]Restart time can be rounded up to match the closest $k\tau_c$.

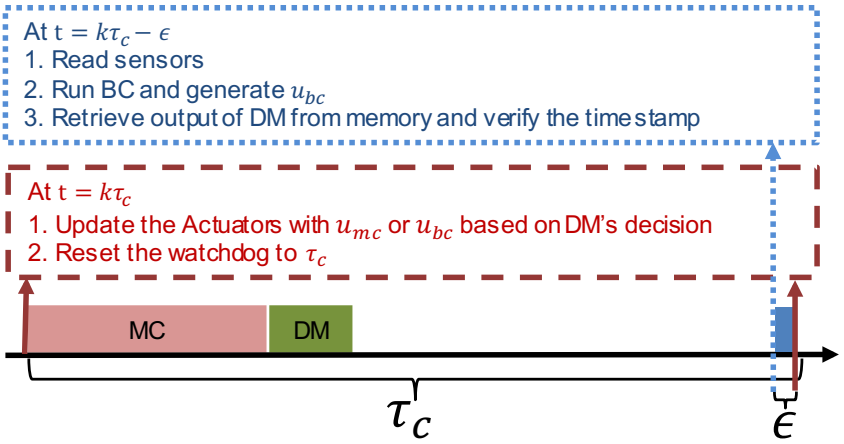


Figure 4.2: Sequence of events within one control cycle.

reliable.

- BC, DM, and flushing task are independently verified and fault-free. They might, however, fail silently (no output is generated) due to the faults in the previously dependent software layers or other applications.
- System-level and application-level faults may cause BC, DM, and flushing task to fail silently but may not change their logic or alter their output.
- Once a command is sent to an actuator input; the actuator holds that value until the control system sends a new actuation command. Therefore, during a system-level restart, the actuators operate with the last command that was sent before the restart occurred[3].
- It is assumed that the system-level faults do not happen within the first τ_r seconds after the boot is complete so that the BC has the chance to execute correctly at least once. In other words, this assumption implies that the system is not completely dysfunctional. In Section 4.1.2, we demonstrate the necessity of this assumption.

4.1.2 Properties of the Base Controller

In this section, we provide the properties required for the BC as follow:
There exists a subset $\mathcal{I}$ of the state space, such that for all $x \in \mathcal{I}$ at time $t_0 \in \mathbb{R}_0^+$, there exists a control command $u_{bc} \in \mathcal{S}_u$, such that:

[3]Commercial chips such as [60] are available that provide programmable PWM controller. Using these intermediate chips one can prevent the invalid signals that may appear on the general-purpose input/output (GPIO) port of the board during a restart, from changing the actuation command.

(i) $\xi_{x,u_{bc}}(t_0 + \tau_c) \in \mathcal{I}$,

(ii) $\xi_{x,u_{bc}}(t_0 + \tau_c + \tau_r) \in \mathcal{I}$, and

(iii) $\xi_{x,u_{bc}}(t) \in \mathcal{S}$ for $t \in [t_0, t_0 + \tau_c + \tau_r]$.

Note that, in the rest of the paper we assumed that the actuators hold the control input constant within the period of $[t_0, t_0 + \tau_c + \tau_r]$.

Intuitively, above properties imply that if the current state of the system is inside $\mathcal{I}$, BC is able to generate a control command that keeps the physical system safe. For the intuition, consider $t_0 = k\tau_c$. Property (i) implies that one control cycle after u_{bc} is applied to the actuators, at the end of $(k+1)$th cycle, state is inside $\mathcal{I}$. Therefore, if the system is still running and no faults have occurred, BC is able to find another safe command at $t = (k+1)\tau_c$. If a fault had occurred within the $(k+1)$th cycle, a restart will be triggered at the end of the cycle and BC will not be available to update the actuator input. Property (ii) implies that in such a case, the system will be in $\mathcal{I}$, after the restart completes. This guarantees that the system can be kept safe after the restart completes. Finally, property (iii) ensures that the system remains inside the safety region during $(k+1)$th cycle and a possible consequent restart.

A BC with the above properties, without any other components, can keep the system safe, only if it updates the actuator commands at least once after every restart τ_r. Therefore, it is necessary for the system to not have any system-level faults within the first τ_r seconds after the restart.

4.1.3 Switching Logic of DM

A system with only BC remains safe and tolerates restarts but it does not make any progress towards the mission goal. In order to maximize the progress towards the mission goal, it is desirable to use the MC command in every cycle whenever it is possible.

In every cycle k, DM runs and evaluates the following conditions. If those conditions hold, u_{mc} is safe to be applied to the actuator inputs at the end of the cycle (*i.e.,* at time $t = k\tau_c$). Otherwise, DM chooses u_{bc}. Following conditions guarantee that the system remains safe and recoverable under u_{mc} whether it restarts or not.

(i) $\mathbf{Reach}_{\tau_c}(\bar{x}[k], u_{mc}) \subseteq \mathcal{I}$

(ii) $\mathbf{Reach}_{\tau_r+\tau_c}(\bar{x}[k], u_{mc}) \subseteq \mathcal{I}$

(iii) $\mathbf{Reach}_{[0,\tau_r+\tau_c]}(\bar{x}[k], u_{mc}) \subseteq \mathcal{S}$

Here, τ_r and τ_c are the length of the restart time and of the control cycle of the platform. Notation $\bar{x}[k]$ denotes the state of the system when the actuator command is going to be applied to the system (*i.e.,* the end of the cycle at time $t = k\tau_c$).

From properties of the BC, it is known that if the state is inside $\mathcal{I}$, BC can find a control command that keeps the system in safe and restartable region. Condition (i) ensures that one control cycle after u_{mc} is applied to the system the state will be inside $\mathcal{I}$. If no faults occur within the control cycle, BC is guaranteed to be able to find a safe control for the system. However, if a fault occurs within the cycle, WD triggers a system restart at the end of the cycle. Condition (ii) ensures that state will be inside $\mathcal{I}$ when the restart completes (*i.e.,* at $\tau_c + \tau_r$). Furthermore, condition (iii) guarantees that during the control cycle and restart time (if it happens) state remains inside the safety region.

Note that, in the real implementation, calculating reachable set and therefore evaluating these conditions requires time and does not happen instantaneously. Therefore, assuming k is the current cycle, above conditions have to be assessed before $t = k\tau_c$. At this time, however, $x[k] = x(k\tau_c)$ (state of the system when the actuator command is going to be updated) is not available yet. To address this issue, above conditions use $\bar{x}[k]$ which is the over-approximated prediction of $x[k]$ based on $x[k-1]$ (sampled sensor values in the previous cycle). Prediction $\bar{x}[k]$ can be computed in the following way:

$$\bar{x}[k] = \overline{\mathbf{Reach}}_{\tau_c}(x[k-1], u_{k-1}),$$

where $x[k-1]$ is the sampled state at the previous cycle (state of the system at the beginning of the current control cycle). Input u_{k-1} is the control command sent to the actuators in the previous cycle. Since, in the first control cycle after a restart, $u_{\text{k-1}}$ is not available, the DM always chooses the BC in the first cycle. To compute an over-approximation of reachable set for nonlinear control systems there are various approaches available in literature for example see [61, Section VIII.c], [62], and [63].

4.2 BASE CONTROLLER DESIGN

In this section, a systematic approach is provided to design base controllers ensuring properties mentioned in Subsection 4.1.2. To design BC, we use symbolic controller synthesis approach which uses the discrete abstractions of nonlinear physical systems [64]. The advantage of using this approach is that it provides formally verified controllers for high-level specifications (usually expressed as linear temporal logic (LTL) formulae [65]). One can readily see that the properties given in Subsection 4.1.2 are equivalent to invariance

specification.

4.2.1 Transition Systems and Equivalence Relation

We recall the notion of *transition system* introduced in [64] which will later be used as unified framework to represent nonlinear control systems and corresponding discrete abstractions.

Definition 4.1 (Transition system)**.** *A transition system is a tuple* $S = (X, X_0, U, \longrightarrow)$ *where* X *is a set of states,* $X_0 \subseteq X$ *is a set of initial states,* U *is a set of inputs,* $\longrightarrow \subseteq X \times U \times X$ *is a transition relation.*

We denote by $x \xrightarrow{u} x'$ an alternative representation for transition $(x, u, x') \in \longrightarrow$, where state x' is called a u-successor (or simply successor) of state x , for some input $u \in U$. $Post_u(x)$ denotes the set of all u-successors of state x, and by $U(x)$ the set of all admissible inputs $u \in U$ for which $Post_u(x)$ is non-empty. Now, I provide the notion of feedback refinement relation between two transition systems, introduced in [61], which is later used to construct discrete abstractions and base controllers for nonlinear control systems Σ.

Definition 4.2 (Feedback refinement relation)**.** *Consider two transition systems* $S_1 = (X_1, X_{10}, U_1, \underset{1}{\longrightarrow})$ *and* $S_2 = (X_2, X_{20}, U_2, \underset{2}{\longrightarrow})$ *with* $U_2 \subseteq U_1$*. A strict relation* $Q \subseteq X_1 \times X_2$ *is a feedback refinement relation from* S_1 *to* S_2 *if following conditions hold for every pair* $(x_1, x_2) \in Q$*:*

(i) $U_2(x_2) \subseteq U_1(x_1)$,

(ii) $u \in U_2(x_2) \Rightarrow Q(Post_u(x_1)) \subseteq Post_u(x_2)$,

and the feedback refinement relation from S_1 *to* S_2 *is denoted by* $S_1 \preceq_Q S_2$*.*

Intuitively, the above relation says that all admissible inputs of S_2 can be used in transition system S_1 such that all transitions in S_1 are associated with corresponding transitions in S_2. As a result, one can easily refine controller synthesized for S_2 using feedback refinement relation Q to make it compatible for S_1. Further details about feedback refinement relation and its role in the controller synthesis can be found in [61].

4.2.2 Sampled-Data Control System as a Transition System

As discussed in the previous sections, the sampling time can take any value in $h = \{\tau_c, \tau_r + \tau_c\}$ depending on the occurrence of fault. It is assume that the value of control

input is held for the respective sampling period. The transition system associated with the nonlinear control system Σ with such a sampling behavior can be given by the tuple

$$S_h(\Sigma) = (X_h, X_{h0}, U_h, \underset{h}{\longrightarrow}), \tag{4.1}$$

where

- $X_h = \mathbb{R}^n$, $X_{h0} = \mathbb{R}^n$, $U_h = \mathsf{U}$, and
- $x \xrightarrow[h]{u} x'$ is a transition if and only if there exists $x' = \xi_{x,u}(\tau_c)$ or $x' = \xi_{x,u}(\tau_r + \tau_c)$, where $u \in U_h$.

Note that we abuse notation above by identifying u with the constant input curve with domain $[0, \tau_c]$ or $[0, \tau_r + \tau_c]$ and value u.

For the transition system $S_h(\Sigma)$, the finite or infinite run generated from initial state $x_0 \in X_{h0}$ is given by $x_0 \xrightarrow[h]{u_0} x_1 \xrightarrow[h]{u_1} x_2 \xrightarrow[h]{u_2} \dots$ such that $x_i \xrightarrow[h]{u_i} x'_{i+1}$, for $i \in \mathbb{N}_0$.

By considering properties of BC mentioned in Subsection 4.1.2, one can view it as a safety controller synthesis problem for $S_h(\Sigma)$.

Definition 4.3 (Safety controller)**.** *Consider a safe set $\mathcal{S} \subseteq \mathbb{R}^n$ as given in Subsection 3.3.2, a safety controller for $S_h(\Sigma)$ is given by a map $C_h : X_h \to 2^{U_h}$ such that:*

(i) for all $x \in X_h$, $C_h(x) \subseteq U_h(x)$,

(ii) its domain $dom(C_h) = \{x \in X_h \mid C_h \neq \emptyset\} \subseteq \mathcal{S}$,

(iii) for all $x \in dom(C_h)$ and $u \in C_h(x)$, $Post_u(x) \subseteq dom(C_h)$.

Essentially, a safety controller generates infinite runs $x_0 \xrightarrow[h]{u_0} x_1 \xrightarrow[h]{u_1} x_2 \xrightarrow[h]{u_2} \dots$ such that $x_i \in \mathcal{S}$, for all $i \in \mathbb{N}_0$. At the end of this section, we provide a systematic way to compute such controller for linear control systems. However, finding such a control strategy for complex nonlinear control systems is quite difficult. This motivates the use of abstraction-based synthesis methods described below.

4.2.3 Discrete Abstraction

To design controllers for the concrete system $S_h(\Sigma)$ from its abstraction, the system and its abstraction must satisfy formal behavioural inclusions in terms of feedback refinement

relations. Consider sampling times $\tau_c, \tau_r + \tau_c \in \mathbb{R}^+$ and quantization parameter $\eta \in (\mathbb{R}^+)^n$. The *discrete abstraction* of $S_h(\Sigma)$ is given by the tuple

$$S_q(\Sigma) = (X_q, X_{q0}, U_q, \underset{q}{\longrightarrow}), \tag{4.2}$$

where

- X_q is a cover of X_h and elements of the cover X_q are nonempty, closed hyper-intervals referred to as cells. For computation of the abstraction, we consider subset $\overline{X}_q \subseteq X_q$ of congruent hyper-rectangles aligned on a uniform grid parameterized with quantization parameter $\eta \in (\mathbb{R}^+)^n$ and given by $\eta\mathbb{Z}^n = \{c \in \mathbb{R}^n \mid \exists_{k\in\mathbb{Z}^n} \forall_{i\in\{1,2,\ldots,n\}} c_i = k_i\eta_i\}$, *i.e.*, $x_q \in \overline{X}_q$ implies that there exists $c \in \eta\mathbb{Z}^n$ with $x_q = c + [\![\frac{\eta}{2}, \frac{\eta}{2}]\!]$. The remaining cells $X_q \setminus \overline{X}_q$ are considered as "overflow" symbols, see [66, Sec III.A]
- $X_{q0} \subseteq X_q$,
- U_q is a finite subset of U_h,
- for $x_q \in \overline{X}_q$ and $u \in U_q$,
  define $A := \{x'_q \in X_q \mid (x'_q \cap \overline{\mathbf{Reach}}_{\tau_c}(x_q, u_q)) \cup (x'_q \cap \overline{\mathbf{Reach}}_{\tau_r+\tau_c}(x_q, u_q)) \neq \emptyset\}$. If $A \subseteq \overline{X}_q$, then $Post_u(x_q) = A$, and otherwise $Post_u(x_q) = \emptyset$. Moreover, $Post_u(x_q) = \emptyset$ for all $x_q \in X_q \setminus \overline{X}_q$.

For the exact procedure to compute such discrete abstraction, we refer interested readers to [67].

Theorem 4.1. *If $S_q(\Sigma)$ is a discrete abstraction of $S_h(\Sigma)$ with sampling times $\tau_c, \tau_r + \tau_c \in \mathbb{R}^+$, and quantization parameter $\eta \in (\mathbb{R}^+)^n$, then $S_h(\Sigma) \preceq_Q S_q(\Sigma)$.*

Proof. The proof is similar to the proof of [61, Theorem VIII.4].

The abstract safe set $\hat{\mathcal{S}}$ for $S_q(\Sigma)$ is given by $\hat{\mathcal{S}} := \{x_q \in \overline{X}_q \mid Q^{-1}(x_q) \subseteq \mathcal{S}\}$.

4.2.4 Controller Synthesis and Refinement

In this section, I consider the problem of synthesis of safety controller C_h for $S_h(\Sigma)$ and safe set $\mathcal{S}$. Because of the feedback refinement relation, the safety controller synthesis problem can be solved for the discrete abstraction $S_q(\Sigma)$ and abstract safe set $\hat{\mathcal{S}}$. Let $C_q : X_q \to U_q$ be the maximal safety controller satisfying conditions in Definition 4.3 for $S_q(\Sigma)$ and safe set $\hat{\mathcal{S}}$. Since $S_q(\Sigma)$ has finite states and inputs, standard maximal fixed-point computation

algorithm [64] canbe used for the computation of C_q. One can easily refine this controller for $S_h(\Sigma)$ and safe set $\mathcal{S}$ using the following theorem:

Theorem 4.2. *If $S_h(\Sigma) \preceq S_q(\Sigma)$ and C_q is the safety controller for $S_q(\Sigma)$ and $\hat{\mathcal{S}}$, then the refined controller $C_h := C_q \circ Q$ solves the safety problem for $S_h(\Sigma)$ and $\mathcal{S}$.*

Proof. The proof is similar to the proof of [61, Theorem VI.3].

Intuitively, the refined controller C_h for S_h can naturally be obtained from the abstract controller C_q by using the feedback refinement relation Q as a *quantizer* to map x_h to $x_q \in Q(x_h)$.

Remark 4.1. *The obtained controller C_h solves the safety problem for the sampled system, i.e., the obtained base controller satisfies the first two properties mentioned in Subsection 4.1.2 with invariant set $\mathcal{I} = dom(C_h)$. However, one can ensure safety guarantee of inter-sampling trajectory (i.e., third property in Subsection 4.1.2) by shrinking the safe set by a magnitude computed using the global Lipschitz continuity property of map f.*

Despite the applicability of the proposed approach for complex and nonlinear control systems, it suffers from the curse of dimensionality, i.e., the computational complexity increases exponentially with state-space dimensions of concrete systems. There are few results available to address this issue for some class of nonlinear control systems [68, 69]. In next subsection, I provide an alternative approach to compute invariant set $\mathcal{I}$ and BC for linear control systems.

4.2.5 Base Controller for Linear Control Systems

In this subsection, I provide an algorithm to compute $\mathcal{I}$ using discretized linear-control systems. The continuous linear control system can be converted to a discrete control system with the sampling time of τ_c as:

$$\dot{\xi}(t) = A\xi(t) + B\upsilon(t) \rightarrow x[k+1] = A_d x[k] + B_d u[k], \tag{4.3}$$

where $A_d = e^{A\tau_c} = \sum_{k=0}^{\infty} \frac{1}{k!}(A\tau_c)^k \simeq \sum_{k=0}^{p} \frac{1}{k!}(A\tau_c)^k$, and $B_d = \left(\int_0^{\tau_c} e^{At} dt\right) \cdot B$.

In this subsection, it is shown how to construct a BC with the properties: $\forall x[k] \in \mathcal{I}, \exists u_0$, where $u[p] = u_0, p \in \{k, k+1, ..., k+m\}$ such that (i) $x[k+1] \in \mathcal{I}$ and (ii) $x[k+1+m] \in \mathcal{I}$, where $m = \tau_r/\tau_c$ and $m \in \mathbb{N}$.

In this section, I will show how to construct a BC that satisfies the following property,

$$\begin{gathered}\forall x[k] \in \mathcal{I}, \exists u_0 \text{ where } u[p] = u_0, p \in \{k, k+1, ..., k+m\} \\ \text{such that } (i)\ x[k+1] \in \mathcal{I} \text{ and } (ii)\ x[k+1+m] \in \mathcal{I},\end{gathered} \tag{4.4}$$

where $m = \tau_r/\tau_c$ and $m \in \mathbb{N}$.

4.2.6 Readjusting the Safety Region

There is one issue that needs to be addressed before calculating $\mathcal{I}$ and the BC. The property presented in Equation 4.4 implies that the state will remain inside $\mathcal{I}$ after one control cycle and one restart time after that. However, it does not imply anything about the trajectory of state within the restart time. To guarantee safety, trajectory of the plant during the restart interval must remain inside $\mathcal{S}$.

To enforce this, I find a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that if $x[t_0] \in \mathcal{S}'$, then $\xi(t) \in \mathcal{S}$ for any $v(t) \in \mathcal{S}_u$ and $t \in [t_0, t_0 + \tau_r]$. Later on, we enforce $\mathcal{I}$ to be a subset of $\mathcal{S}'$. This ensures that if the state is in $\mathcal{I}$ at the sampling time, it cannot go outside of $\mathcal{S}$ within τ_r time. This approach uses some similar concepts that are used in [30] for computing real-time reachability.

Before explaining the procedure, some notations and definitions are necessary. Note that from the definition of $\mathcal{S}$ in Section 3.3.2, $\mathcal{S}$ is a convex polyhedron because it is the intersection of a finite number of half-spaces. For a real vector c and a real number d, a linear inequality $c^T x \leq d$ is called valid for $\mathcal{S}$ if $c^T x \leq d$ holds for all $x \in \mathcal{S}$. A subset f of a polyhedron $\mathcal{S}$ is called a *face* of $\mathcal{S}$ if there exists a valid inequality $c^T x \leq d$ for $\mathcal{S}$, so that f is represented as $f = \mathcal{S} \cap \{x : c^T x = d\}$.

For a given face f, let its surface normal be $\vec{n}$. The outward direction normal will be either $\vec{n}$ or $-\vec{n}$. To determine which, let v be a point such that $v \in \mathcal{S}$ and $v \notin f$ and let one of the vertices of the face f be p. Now, consider the two vectors $\vec{n}$ and $\vec{pv} = v - p$. If $\vec{n} \cdot \vec{pv}$ is negative, then $\vec{n}$ is facing outwards or vice versa (Figure 4.3). Furthermore, for a given face f, there exists a linear inequality $c^T x \leq d$ that is valid for $\mathcal{S}$ and we have $f = \mathcal{S} \cap \{x : c^T x = d\}$. The inward neighborhood of the face f with width $l \geq 0$ is $n_f = \mathcal{S} \cap \{x : c^T x \geq d - l\}$.

Following steps describe the procedure to find $\mathcal{S}'$:

1. The maximum outward derivative along each face (in the direction of outward normal vector of the face) of the $\mathcal{S}$ over all the inputs ($\mathcal{S}_u$) is computed. One inward neighborhood is constructed for each face (Figure 4.3), where the width

of the corresponding neighborhood is based on the observed maximum outward derivative (the width is the derivative multiplied by the τ_r).

2. The neighborhoods are all constructed based on the computed widths, such that the edges overlap as shown in Figure 4.3.

3. In each constructed neighborhood (n_i), the maximum outward derivative is calculated over the states in that neighborhood (n_i) and all the inputs ($\mathcal{S}_u$). If it is larger than the previously observed maximum, the width of the neighborhoods are recomputed, and the process repeats by returning to step 2.

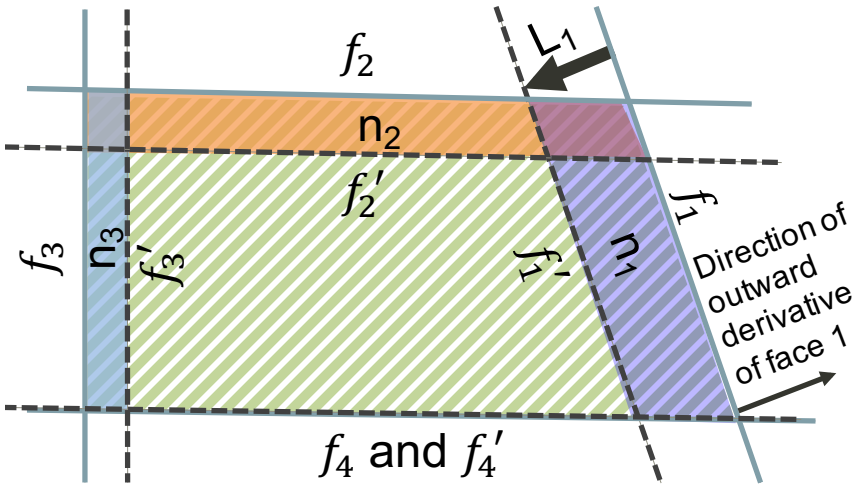


Figure 4.3: An example to illustrate construction of $\mathcal{S}'$ from $\mathcal{S}$. The area confined with f_1, f_2, f_3 and f_4 is $\mathcal{S}$. The area confined with f_1', f_2', f_3' and f_4' is $\mathcal{S}'$.

The computed subset is called *adjusted safety region*, is denoted by $\mathcal{S}'$, and can be represented with some matrix H_x^a and a vector h_x^a of appropriate dimensions in the form of $H_x^a \cdot x \leq h_x^a$, where the inequality is interpreted by components. Any trajectory starting from a point in the $\mathcal{S}'$, will not reach any state outside of $\mathcal{S}$ within a τ_r time unit.

To guarantee the termination, the number of times the algorithm can return to step[4] 2 are limited. For some systems, this procedure may result in an empty set. An empty set indicates that the restart time τ_r of the platform is too long for the given physical plant *i.e.,* physical plant has fast dynamics and its state can change very quickly relative to the time it takes to restart the controller unit.

[4]For the given system with linear dynamics, if a maximum derivative exists over the given $\mathcal{S}$ and for all inputs, the procedure is guaranteed to terminate in a finite number of steps.

Note that, to compute a more efficient $\mathcal{S}'$, the algorithm above can be repeated $q \in \mathbb{N}$ times and each time using a time parameter of τ_r/q instead of τ_r in steps 1, 2, and 3. Increasing the value of q can lead to finding a larger $\mathcal{S}'$ region and also may increase the computation time of $\mathcal{S}'$.

4.2.7 Finding the Invariant Subset $\mathcal{I}$

To compute the set $\mathcal{I}$, we closely follow the usual construction method based on backwards reachable sets to compute the *largest* invariant set for linear discrete-time systems (see e.g. in [70]). This procedure is modified slightly (Algorithm 4.1) to compute the subset $\mathcal{I} \subseteq \mathcal{S}'$, such that for the discrete-time system in Equation 4.3, $\mathcal{I}$ satisfies the properties in Section 4.1.2 or Equation 4.4.

Algorithm 4.1: Computing the invariant subset $\mathcal{I}$.

1 `ComputeInvRegion(`H_x^a,h_x^a,H_u,h_u,A_d,B_d,$A_d^{(m+1)}$,$B_d^{(m+1)}$`)`
2 $I^{(0)}$:= Polytope($H_x^a \cdot x \leq h_x^a$) and k = 0
3 **while** $p <$ p_{max} **do**
4 $[H'_x, h'_x]$:= PolytopeToMatrix($I^{(k)}$)
5 pt := Polytope

$$\left(\begin{bmatrix} H'_x \\ H'_x \\ H_u \end{bmatrix} \begin{bmatrix} A_d^{(m+1)} & B_d^{(m+1)} \\ A_d & B_d \\ 0_{m\times n} & I_{m\times m} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \begin{bmatrix} h'_x \\ h_u \end{bmatrix} \right)$$

6 $I^{(p+1)}$:= pt.projectOnStateSpace()
7 **if** $I^{(p)} \subseteq I^{(p+1)}$ **then**
8 $[H_x^{\mathcal{I}}, h_x^{\mathcal{I}}]$:= PolytopeToMatrix($I^{(p)}$)
9 STOP successfully.
10 **else if** $I^{(p+1)}$ *is empty* **then**
11 STOP unsuccessfully.
12 **else**
13 p:=p+1
14 **end**
15 **end**
16 **return** $H_x^{\mathcal{I}}, h_x^{\mathcal{I}}$

In this algorithm, matrix H_x^a and vector h_x^a represent the adjusted safety region $\mathcal{S}'$, and matrix $H_x^{\mathcal{I}}$ and vector $h_x^{\mathcal{I}}$ represent $\mathcal{I}$. We have $m = \tau_r/\tau_c$. $A_d^{(m+1)}$ and $B_d^{(m+1)}$ are the matrices to find the state after $m+1$ cycles *i.e.,* $x[k+m+1] = A_d^{(m+1)}x[k] + B_d^{(m+1)}u[k]$. We have $A_d^{(m+1)} = (A_d)^{m+1}$ and $B_d^{(m+1)} = (A_d^m + A_d^{m-1} + ... + I)B_d$.

Intuitively, this algorithm starts from $\mathcal{S}'$ as initial region (line 2). In every iteration of this algorithm, this region is augmented in the extended state-control space $\mathbb{R}^{n+m}$ (line 5).

This linear inequality is then projected back into the state space (line 6). The outcome of lines 5 and 6 is to calculate $\mathcal{I}^{(p+1)}$ which is the subset of states in $\mathcal{I}^{(p)}$ where a control value in $\mathcal{S}_u$ exists such that, the state in one cycle and $m+1$ cycle after is inside $\mathcal{I}^{(p)}$.

The algorithm proceeds until either $\mathcal{I}^{(p)} \subseteq \mathcal{I}^{(p+1)}$ or $\mathcal{I}^{(p+1)} = \emptyset$. In the former case, procedure successfully ends (lines 7 to 8). The latter case indicates that the dynamics of the system does not allow such a region, for the given restart time. There are cases in which the procedure does not stop in a finite number of steps unless a finite p_{max} is fixed. This may happen if $\mathcal{I}^{(\infty)}$ has an empty interior, but it is not empty [70].

If matrix A_d and B_d are controllable, we can use ideas from [71] to ensure convergence. However, in general we cannot guarantee that the procedure in Algorithm 4.1 will converge to a non-empty $\mathcal{I}$. In such cases, one may have to loosen the safety constraints of the system (*i.e.,* $\mathcal{S}$) or may have to switch to a hardware platform with a shorter restart time, to take advantage of our approach.

4.2.8 Base Controller in Runtime

All the previous steps introduced in Subsections 4.2.6 and 4.2.7 are offline and take place at the design time. What remains is to describe how the BC calculates the control command in runtime. Assuming that k is the current sampling instance, the goal of BC is to find a control input $u[k]$ that satisfies following conditions:

$$\begin{cases} H_u \cdot u[k] & \leq h_u \\ H_x^{\mathcal{I}} \cdot x[k+1] & \leq h_x^{\mathcal{I}} \\ H_x^{\mathcal{I}} \cdot x[k+m+1] & \leq h_x^{\mathcal{I}} \end{cases} \tag{4.5}$$

With replacing the $x[k+1]$ and $x[k+m+1]$ from the discrete-time model (4.3) in the above equations we have the following linear inequalities:

$$\begin{cases} H_u u[k] & \leq h_u \\ H_x^{\mathcal{I}} B_d u[k] & \leq h_x^{\mathcal{I}} - H_x^{\mathcal{I}} A_d x[k] \\ H_x^{\mathcal{I}} B_d^{(m+1)} u[k] & \leq h_x^{\mathcal{I}} - H_x^{\mathcal{I}} A_d^{(m+1)} x[k] \end{cases} \tag{4.6}$$

All of the parameters of the above linear inequalities except $x[k]$ and $u[k]$ are known at the design time. At runtime, BC samples the sensors values *i.e.,* $x[k]$, and calculates $u[k]$ by solving the inequalities in Equation 4.6. From properties of $\mathcal{I}$, it is guaranteed that if $x[k] \in \mathcal{I}$, the solution of these linear inequalities, solved for $u[k]$, is a non-empty set.

4.3 CASE STUDY AND EVALUATION

To demonstrate the practicality of the proposed approach, we implemented a controller for two benchmark systems: (i) inverted pendulum system and (ii) 3-DOF helicopter [72] and empirically verify fault-tolerance guarantees. We utilize one COTS platform to implement our controller. We inject faults in the control logic, control application, and the operating system to demonstrate that the system remains safe, despite the faults, and recovers.

4.3.1 Experimental Setup

For the prototype of the proposed design, an i.MX7D application processor is used. This SoC provides two general purpose ARM Cortex-A7 cores capable of running at the maximum frequency of 1 GHz and one real-time ARM Cortex-M4 core that runs at the maximum frequency of 200MHz. The real-time core runs from tightly coupled memory to ensure predictable behavior required for the real-time applications/tasks. The real-time core of the considered platform runs FreeRTOS [73], an operating system for real-time applications. Because our control tasks have real-time constraints, we implement our controller on the real-time core. Ideally, the general purpose cores would have been completely disabled for the experiments. However, in i.MX7D platform, only Cortex-A7 cores have direct access to the flash memory and, only these two cores can load the binary images of the real-time core from flash into the real-time core's memory after each restart. Hence, instead of permanently disabling those cores, they are only disabled after the software of the real-time core is loaded from flash into the memory. Note that, this mechanism is specific to this particular platform and does not impact the generality of our proposed technique.

The manufacturer's boot procedure of the board is designed to boot the general purpose cores and the real-time core at the same time. It includes extra initialization procedures that are necessary only for running the general purpose core's kernel and mounting its file system. It loads the real-time core code only after those procedures are completed.

To reduce the boot time of the real-time core, we made two modifications to the bootloader (u-boot) source code which can be found in [74]. (i) We included the binary of the real-time core executables (FreeRTOS, MC, BC, DM, and flushing task) as a static array in the u-boot source code and made it part of the u-boot binary after compilation. (ii) In our modified boot process, at the boot time, the general purpose processor copies u-boot binary (that includes the FreeRTOS and application binaries) from the SD-card into the RAM. After successful initialization of only the necessary peripherals and configuring the clock by the u-boot procedures, u-boot loads the binaries of the real-time core in its tightly

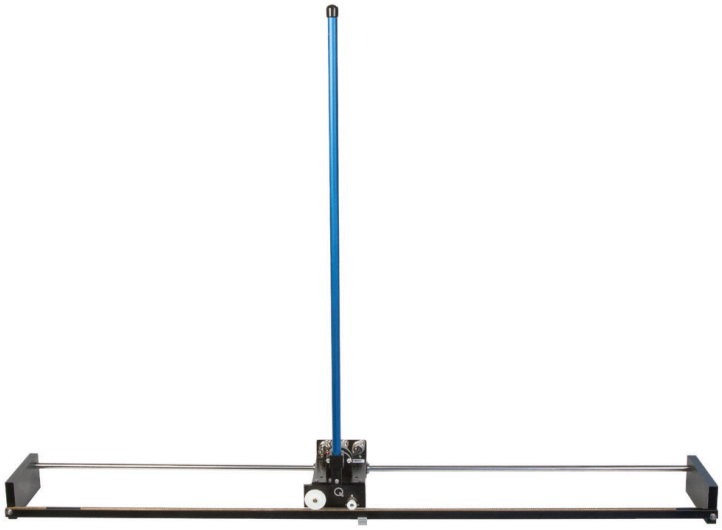


Figure 4.4: Inverted pendulum

coupled memory and releases it from reset. These modifications reduce the real-time core's boot time from seconds to less than 250ms[5].

4.3.2 Example 1: Inverted Pendulum

For the first case study, we consider a nonlinear inverted pendulum system [66] given by nonlinear differential equations as:

$$\begin{aligned}\dot{\xi}_1(t) &= \xi_2(t)\\ \dot{\xi}_2(t) &= -\omega^2\big(\sin(\xi_1(t)) + \cos(\xi_1(t))\upsilon(t)\big) - 2\gamma\xi_1(t),\end{aligned} \tag{4.7}$$

with parameters $\omega = 1$ and $\gamma = 0.0125$. The states ξ_1 and ξ_2 are the angular position with respect to a downward vertical axis and the angular velocity of the pendulum, respectively. The control input $\upsilon(t)$ is restricted to $[-4, 4]$. We design MC as $\upsilon_{mc} = 2(\pi - \xi_1 - \xi_2)$ to stabilize the pendulum at upright position that is $\xi = [\pi, 0]^T$ which runs with frequency of 20Hz (i.e. τ_c =50ms) on real-time core of i.MX7D. To ensure safety of the system (i.e. to avoid pendulum to fall down), we consider safety region for the states given by a polytope

[5]BC task activates one of the GPIO pins immediately after it executes. The restart time is measured externally using the signal on this pin. After multiple experiments, a conservative upper bound was picked for the restart time.

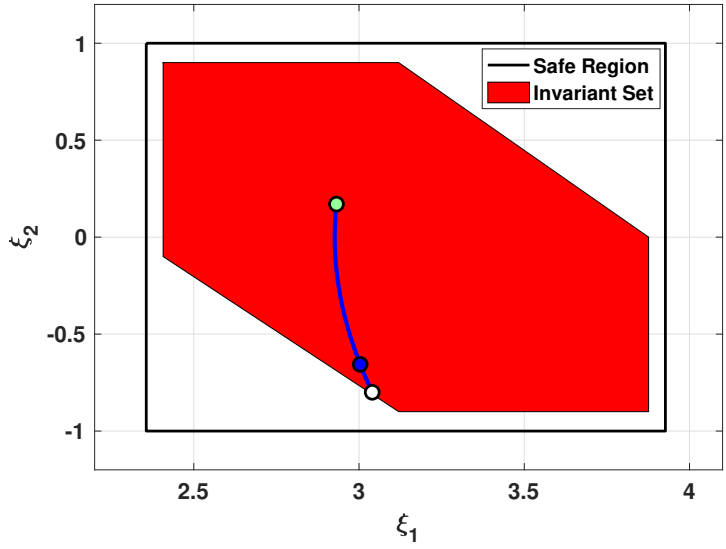


Figure 4.5: Invariant set obtained using abstraction based approach and a simulated closed-loop trajectory of the system under $u = 3$ which is inside $\mathcal{I}$ (red region) at times $\tau_c = 50$ ms (blue mark) and $\tau_c + \tau_r = 300$ ms (green mark). White circle marks the beginning of the trajectory $\xi = [3.04, -0.8]^T$.

parameterized by

$$H_x = \begin{bmatrix} -1 & 0 \\ 1 & 0 \\ 0 & -1 \\ 0 & 1 \end{bmatrix} \text{ and } h_x = \begin{bmatrix} -0.75\pi \\ 1.25\pi \\ 1 \\ 1 \end{bmatrix}.$$

To ensure fault-tolerance and safety during restart, we designed BC using abstraction-based approach as discussed in Section 4.2. To synthesize BC, we first constructed a discrete abstraction of the pendulum system in (4.7) using quantization parameter $\eta = [0.05, 0.1]^T$, sampling time $\tau_c = 0.050$, and restart time $\tau_r = 0.250$. Further, we synthesize a safety controller using maximal fixed point computation algorithm. For the controller synthesis, we used toolbox SCOTS [67] with some modifications to adapt the construction of abstraction given in Subsection 4.2.3 . The invariant states computed using the proposed approach is shown in Figure 4.5. To verify the efficacy of the designed controller, we implemented it on our experimental setup (i.MX7D) and tested in the closed-loop with inverted pendulum dynamics simulated in the computer under various test scenarios discussed in Subsection 4.3.4.

4.3.3 Example 2: 3-DOF Helicopter

3-DOF helicopter (displayed in Figure 4.6) is a simplified helicopter model, ideally suited to test intermediate to advanced control concepts and theories relevant to real-world applications of flight dynamics and control in the tandem rotor helicopters, or any device with similar dynamics [72]. It is equipped with two motors that can generate force in the upward and downward direction, according to the given actuation voltage. It also has three sensors to measure elevation, pitch, and travel angle as shown in Figure 4.6. We use the linear model of this system obtained from the manufacturer manual [72]. The BC is designed as discussed in Subsection 4.2.5.

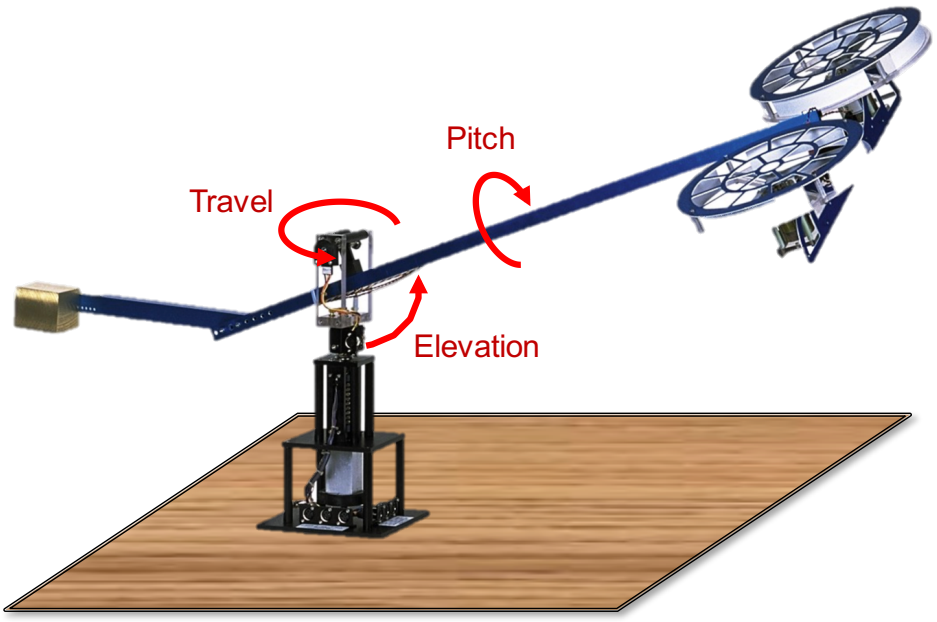
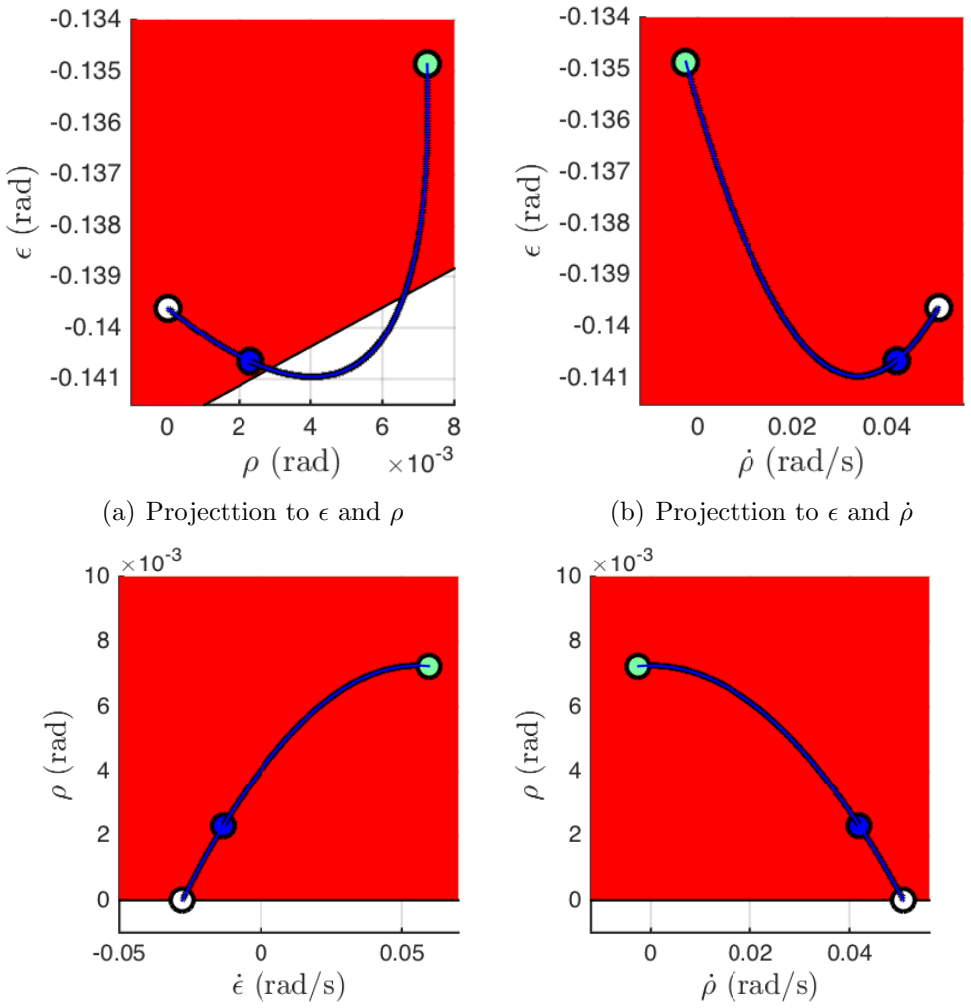


Figure 4.6: 3 Degree of freedom (3-DOF) helicopter.

For 3-DOF helicopter, the safety region is defined in such a way that the helicopter fans do not hit the surface underneath, as shown in Figure 4.6, while respecting the maximum angular velocities. The six dimensional state vector is given by $x = [\epsilon, \rho, \lambda, \dot{\epsilon}, \dot{\rho}, \dot{\lambda}]^T$, where variables ϵ, ρ, and λ are the elevation, pitch, and travel angles, respectively, $\dot{\epsilon}$, $\dot{\rho}$,and $\dot{\lambda}$ are the corresponding angular velocities. The $u = [v_l, v_r]^T$ represents input vector, where v_l and v_r are the voltages applied to right and left motors. The safe region for the state and input spaces are represented using polytopes as discussed in Subsection 3.3.2 and parametrized



(a) Projecttion to ϵ and ρ

(b) Projecttion to ϵ and $\dot{\rho}$

(c) Projecttion to $\dot{\epsilon}$ and ρ

(d) Projecttion to ρ and $\dot{\rho}$

Figure 4.7: Simulated trajectory of the system under $v_r = 0.6863$ and $v_l = 0.7709$ is inside $\mathcal{I}$ (red region) at times $\tau_c = 50$ ms (blue mark) and $\tau_c + \tau_r = 300$ ms (green mark). White circles mark the beginning of the trajectory. The trajectory is projected into the four planes for clarity.

with

$$H_x = \begin{bmatrix} -1 & -0.33 & 0 & 0 & 0 & 0 \\ -1 & 0.33 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix}, h_x = \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \\ 0.4 \\ 1.5 \\ 1.5 \end{bmatrix}, H_u = \begin{bmatrix} -1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix}, \text{and } h_u = \begin{bmatrix} 1.1 \\ 1.1 \\ 1.1 \\ 1.1 \end{bmatrix}.$$

FreeRTOS on the Cortex-M4 core restarts in 250 ms (upper bound). By using Algorithm 4.1, we computed readjusted safety constraint parameters as $h_x^a = [0.1418, 0.1418, 0.2828, 0.2828, 0.0825, 0.0825]^T$ and $H_x^a = H_x$. Using this readjusted safety constraints the invariant region and BC are constructed using the Algorithms described in Subsections 4.2.7 and 4.2.8. Algorithm 4.1 computed a region $\mathcal{I}$ confined with 106 inequalities after 14 iterations. The offline computation took four hours on Mac Book Pro with 2.5 GHz Intel Core i7 and 16 GB of memory. Finally, the BC is derived by solving linear inequalities in (4.6).

Hardware Interface

The control tasks on the real-time core of i.MX7D run with a frequency of 20 Hz (τ_c = 50 ms). Our controller interfaces with the 3DOF helicopter through a PCIe-based *Q8 High-Performance H.I.L. Control and data acquisition unit* [75] and an intermediate Linux-based PC. The PC communicates with the i.MX7D through the serial port. At the end of every control cycle, a flushing task on the real-time core communicates with the PC to receive the sensor readings (elevation, pitch, and travel angles) and send the motors' voltages. It also updates the hardware WD of the platform after sending the motor voltages. The PC uses a custom driver written for Linux to send the voltages to the 3DOF helicopter motors and reads the sensor values.

Testing the Base Controller

To verify that the constructed base controller has the desired properties, we simulated the system with this controller from all vertices of region $\mathcal{I}$ as starting points and observed that the system's state at τ_c and $\tau_c + \tau_r$ time units after actuation was inside $\mathcal{I}$. Figure 4.7 outlines one extreme example. The trajectory starts at $\epsilon = -0.1410$, $\rho = 0$, $\dot{\epsilon} = -0.0281$ and $\dot{\rho} = 0.0513$ (λ and $\dot{\lambda}$ do not impact safety). The control command in this trajectory is $v_r = 0.6863$ and $v_l = 0.7709$. As shown in Figure 4.7, the trajectory remains inside the safety region.

Further, we implemented the obtained BC on the i.MX7D platform to validate our design approach under different fault scenarios given in the next section.

4.3.4 Fault Injection

In Table 4.1, a list of faults that were tested on the implementations are provided. We also compare them with Application-Level Simplex and System-Level Simplex. For the

application-level faults, we verified that the mission controller was able to actuate the system as long as it did not jeopardize the safety and when the system states approached the states where the safety conditions violated, BC took over and ensure safety. For the system-level faults, we observed that the WD restarted the system and after restart, the system continued its operation.

Some of these faults are elaborated in the rest of this section.

Maximum Control Input in Wrong Way

The system should not leave safe region even if the MC outputs a control input that normally would result in a crash. We consider an extreme case of this scenario where the MC generates a control input that forces system towards the unsafe region. The unsafe MC commands were detected by DM (they did not satisfy the system safety conditions), and the control was switched to the BC until the system was in the safety region and then control was handed back to MC.

Timing Faults (CPU and Resource)

The proposed solution also protects the system from timing faults. A faulty task may behave differently in runtime from its expected/reported behavior. For instance, it may lock a particular resource used by other critical tasks for more than the intended duration. Or, it may run for more time than its reported worst-case execution time (WCET) which was used for the schedulability test of the system. Timing faults may also originate from RTOS or driver misbehaviors. If the fault delays/stops the execution of the DM or BC, WD will trigger a system-wide restart. This recovers the system from the fault and keeps the physical system safe. We perform two experiments to test the fault-tolerance against timing faults.

In the first experiment, we run an additional task on the system that uses the serial port in parallel to the flushing task to communicate with the PC. We inject a fault into this task so that in random execution cycles, it holds the lock on the serial port for more than its intended period. This prevents the flushing task from updating the actuator (which needs the serial port) before the end of the control cycle. As a result, WD expires and restarts the system. We verified that the system recovers from the fault and remains safe during the restart.

In the second test, we introduce a task that runs at the same priority as the BC and DM. We inject a fault into the task such that in some cycles, its execution time exceeds its reported WCET. FreeRTOS runs the tasks with equal priority using round-robin scheduling

Failure Type	Fault Category	Safety			Restarted
		App-Level Simplex (Single HW Board/SoC)	Sys-Level Simplex (Additional HW/SoC)	Our Approach (Single HW Board/SoC)	
No Output	App.	✔	✔	✔	No
Maximum Voltage	App.	✔	✔	✔	No
Time Degraded Control	App.	✔	✔	✔	No
Timing Fault - CPU	OS/App.	✗	✔	✔	Yes
Timing Fault - Resource	OS/App.	✗	✔	✔	Yes
FreeRTOS Freeze	RTOS	✗	✔	✔	Yes
Computer Reboot	RTOS	✗	✔	✔	Yes

Table 4.1: Our approach tolerates system-level faults using only one hardware unit. Whereas, System-Level Simplex [20] needs an extra board/SoC to tolerate these faults.

with a context switch at every 1ms. Therefore, the faulty task delays the response time of the DM and BC. If the interference is too long, the output of BC may not be ready by the time the flushing task needs to update the actuators. When this happens, WD restarts the system.

4.4 SUMMARY AND DISCUSSION

In the modern complex systems, faults are a norm rather than an abnormality and safety-critical systems need to be designed with faults in mind. For many traditional computing systems, restarting has always been an effective way to recover the system and bring it back to a functional state so much that the very first diagnosis tip from any product customer service agent would be "have you tried restarting it?". Restarting safety-critical CPS, specially in runtime, is, however, challenging and non-trivial. The work we presented in this chapter enables a system to utilize this tool in a calculated manner, enabling safety-critical CPS to tolerate software faults and continue operation despite their presence. This design, as we mentioned in the Introduction chapter, enables low cost development of safe CPS.

As any other approach, this design has also known limitations. it is limited in handling software faults that modify the logic or output of the BC and the DM at the execution time. The runtime calculation of these units is the very important logic that enables the system to preserve safety. If the logic gets corrupted, the system can fail catastrophically.

Another limitation of this work is the fact that as the restart time of the platform increases, the domain of the BC shrinks. For many systems with very quick physical dynamics, relative to the restart time of the platform that runs the controller, the BC domain may be empty – such systems cannot take advantage of this design. In such cases, the architect may need to loosen the system safety requirements, reduce the restart time of the platform, or find

another platform with a shorter restart time. The proposed design, therefore, may not suit some platforms, particularly, the ones running on more complex hardware with long restart time.

These restrictions are the main motivation behind the alternative designs presented in the next two chapters. In Chapter 5, instead of relying on the controller to maintain the safety, we rely on timing analysis to ensure that the tasks interrupted due to a restart can completed in a timely manner and meet their deadlines. This approach allows the use of a simpler controller with a larger domain. This approach can be applied to certain system where the current design cannot. In Chapter 6, we incorporate the Trusted Execution Platforms (TEE) such as ARM TrustZone [76] into our architecture and use it to limit access to the critical software components as well as reduce the frequency of system restarts. This allows the system to handle more diverse types of faults (faults that may have altered the logic of BC and DM can be handled with this design) and more applications (lower restart frequency enables this design on systems with even faster dynamics or longer platform restart times).

CHAPTER 5: SINGLE NODE CPS SAFETY THROUGH SCHEDULABILITY ANALYSIS

By this point in this dissertation, hopefully, it is established that restarting a computing system and reloading a fresh image of all the software (*i.e.,* RTOS, and applications) from a read-only source appears to be an effective approach to recover from unexpected faults. In the previous chapter we discussed how to construct a BC able to tolerate system restart and preserve physical safety. The caveat is that for complex physical dynamics, computation of BC could become very expensive and effectively impossible. To avoid this problem, we must ensure that the critical control tasks will meet their deadline despite a system restart and complete within a safe time window. If this is the case, the physical system will be oblivious to the ongoing system restarts and the control system will operate as expected.

In this chapter we describe this new design and all the required safety conditions. The work in this chapter relies on a key observation learned from working with embedded system for many years: by performing careful boot-sequence optimization, many embedded platforms and RTOS that are used in automotive industry, avionics, manufacturing, *etc.*can be **entirely restarted** in a very short frame of time.

Here, we propose *a software/hardware co-design methodology* to deploy safety-critical CPS that (i) provides strong safety guarantees and (ii) can utilize unverified software components to implement complex safety-critical functionalities. More specifically, as soon as a fault that disrupts the execution of critical components is detected, the entire system is restarted. After a restart, all the safety-critical applications that were impacted by the restart are re-executed. If restart and re-execution of critical tasks can be performed *fast enough*, i.e. such that timing constraints are always met in spite of task re-executions, the physical system will not be impacted by the occurrence of faults.

The effectiveness of the proposed restart-based recovery relies on timely detection of faults to trigger a restart. Since detecting logical faults in complex control applications can be challenging, we utilize the Simplex Architecture [16, 2, 17] to construct the control software. In the Simplex architecture, each control application is divided into three tasks; safety controller, complex controller and decision module. The safety of the system relies solely on timely execution of the safety controller tasks. From a scheduling perspective, safety is guaranteed if the safety controller has enough CPU cycles to re-execute and finish before their deadlines in spite of restarts. In this chapter, we analyze the conditions for a periodic task set to be schedulable in the presence of restarts and re-executions. We assume that when a restart occurs, the task instance executing on the CPU and any of the tasks that were preempted before their completion will need to re-execute after the restart. In particular,

we make the following contributions:

- We propose a Simplex Architecture that can be recovered via restarts and implemented on a **single processing unit**;
- We derive the response time analysis, under fixed-priority, with fully preemptive and fully non-preemptive disciplines in presence of restart-based recovery and discuss pros and cons of each one;
- We propose response time analysis of fixed-priority scheduling in presence of restarts for tasks with preemption thresholds [3] and non-preemptive ending intervals [4] to improve feasibility of task sets;

5.1 SYSTEM MODEL AND ASSUMPTIONS

In this section we formalize the considered system and task model, and discuss the assumptions under which our methodology is applicable.

5.1.1 Periodic Tasks

We consider a task set $\mathcal{T}$ composed of n periodic tasks $\tau_1 \dots \tau_n$ executed on a uniprocessor under fixed priority scheduling. Each task τ_i is assigned a priority level π_i. We will implicitly index tasks in decreasing priority order, *i.e.*,, τ_i has higher priority than τ_k if $i < k$. Each *periodic task* τ_i is expressed as a tuple (C_i, T_i, D_i, ϕ_i), where C_i is the worst-case execution time (WCET), T_i is the period, D_i is the relative deadline of each task instance, and ϕ_i is the phase (the release time of the first instance). The following relation holds: $C_i \leq D_i \leq T_i$. Whenever $D_i = T_i$ and $\phi_i = 0$, we simply express tasks parameters as (C_i, T_i). Each instance of a periodic task is called *job* and $\tau_{i,k}$ denotes the k-th job of task τ_i. Finally, $hp(\pi_i)$ and $lp(\pi_i)$ refer to the set of tasks with higher or lower priority than π_i *i.e.,* $hp(\pi_i) = \{\tau_j \mid \pi_i < \pi_j\}$ and $lp(\pi_i) = \{\tau_j \mid \pi_i > \pi_j\}$. We indicate with T_r the minimum inter-arrival time of faults and consequent restarts; while C_r refers to the time required to restart the system.

5.1.2 Critical and Non-Critical Workload

It is common practice to execute multiple controllers for different processes of physical plant on a single processing unit. In this work, we use the Simplex Architecture [16, 2, 17] to implement each controller. As a result, three periodic tasks are associated with every

controller: (i) a safety controller (SC) task, (ii) a complex controller (CC) task, and (iii) a decision module (DM) task. In typical designs, the three tasks that compose the same controller have the same period, deadline, and release time.

Remark 5.1. *SC's control command is sent to the actuator buffer immediately before the termination of that job instance. Hence, the timely execution of SC tasks is* **necessary and sufficient** *for the safety of the physical plant.*

As a result, out of the three tasks, SC must execute first and write its output to the actuator command buffer. Conversely, DM needs to execute last, after the output of CC is available, to decide if it is safe to replace SC's command which is already in the actuator buffer. Hence, the priorities of the controller tasks need to be in the following order[1]: $\pi(DM) < \pi(CC) < \pi(SC)$. Note that, the precedence constraint that SC, CC and DM tasks must execute in this order can be enforced through the proposed priority ordering if self-suspension and blocking on resources are excluded and if the scheduler is work-conserving. We consider fixed priority scheduling, which is work-conserving and we assume SC, CC and DM tasks do not self-suspend. Moreover, tasks controlling different components are independent; SC, CC and DM tasks for the same component share sensors and actuator channels. Sensors are read-only resources, do not require locking/synchronization and therefore cannot cause blocking. A given SC task, may only share actuator channels with the corresponding DM task. However, SC jobs execute before DM jobs and do not self-suspend, hence DM cannot acquire a resource before SC has finished its execution.

The set of all the SC tasks on the system is called *critical* workload. All the CC and DM tasks are referred as *non-critical* workload. Safety is guaranteed if and only if all the critical tasks complete before their deadlines. Whereas, execution of non-critical tasks is not crucial for safety; these tasks are said to be mission-critical but not safety-critical. We assume that the first n_c tasks of $\mathcal{T}$ are critical. Notice that with this indexing strategy, any critical task has a higher priority than any non-critical task.

5.1.3 Fault Model

In this chapter, we consider two types of fault for the system; application-level faults and system-level faults. We make the following assumptions about the faults that our system safely handles:

A1 The original image of the system software is stored on a read-only memory unit (*e.g.,* E^{2}PROM). This content is unmodifiable at runtime.

[1]We assume enough priority levels to assign distinct priorities.

A2 Application faults may only occur in the unverified workload (*i.e.,* all the application-level processes on the system except SC and DM tasks).

A3 SC and DM tasks are independently verified and fault-free. They might, however, fail silently (no output is generated) due to faults in software layers or other applications on which they depend.

A4 We only consider system- and application-level faults that cause SC and DM tasks to fail silently but do not change their logic or alter their output.

A5 Faults do not alter sensor readings.

A6 Once SC or CC tasks have send their outputs to the actuators, the output is unaffected by system restart. As such, a task does not need to be re-executed if it has completed correctly before a restart.

A7 Re-executing a task even if it has completed correctly does not negatively impact system safety.

A8 Monitoring and initializer tasks (Section 5.2) are independently verified and fault-free. We assume that system faults can only cause silent failures in these tasks (no output or correct output).

A9 T_r is larger than the least common multiple (hyper-period[2]) of critical tasks, i.e. $T_r > \text{LCM}\{T_k \mid k \leq n_c\}$

5.1.4 Scheduler State Preservation and Absolute Time

In order to know what tasks were preempted, executing, or completed after a restart occurs, it is fundamental to carry a minimum amount of data across restarts. As such, our architecture requires the existence of a small block of non-volatile memory (NVM). We also require the presence of a monotonic clock unit (CLK) as an external device. CLK is used to derive the absolute time after a system restart. Since we assume periodic tasks, the information provided by CLK is enough to determine the last release time of each task. Whenever a critical task is completed, the completion timestamp obtained from CLK is written to NVM, overwriting the previous value for the same task. We assume that a timestamp update in NVM write can be performed in a transactional manner.

[2]Length of the hyper-period can be significantly reduced if the control tasks have harmonic periods.

Figure 5.1: Example of fully preemptive system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, and restart at $t = 10 - \epsilon$ ($C_r = 0$). The taskset is schedulable without restarts, however, restart and task re-execution causes a deadline miss at $t = 22$.

5.1.5 Recovery Model

The recovery action we assume in this paper is to restart the entire system, reload all the software (RTOS and applications) from a read-only storage unit, and re-execute all the jobs that were released but not completed at the time of restart. The priority of a re-executing instance is the same as the priority of the original job. Within C_r time units, the system (RTOS and applications) reloads from a read-only image, and re-execution is initiated as needed. Figure 5.1 depicts how restart and task re-execution affect the scheduling of 3 real-time tasks ($\tau_1, \tau_2,$ andτ_3). When the restart happens at $t = 10 - \epsilon$, τ_1 was still running. Moreover, τ_2 and τ_3 were preempted at time $t = 9$ and $t = 8$, respectively. Hence all the three task will need to be re-executed after the restart.

System restart is triggered only after a fault is detected. The following definition of fault is used throughout this paper:

Definition 5.1 (Critical Fault:)**.** *any system misbehavior that leads to a non-timely execution of any of the critical tasks is a critical fault.*

It follows that (i) the absence of critical faults guarantees that every critical task completes on time; that (ii) the timely completion of all the critical tasks ensures system safety by Assumptions A3-A7; and that (iii) being able to detect all critical faults and re-execute critical tasks by their deadline is enough to ensure timely completion of critical tasks in spite of restarts. We discuss critical fault detection in Section 5.2; and we analyze system

schedulability in spite of critical faults in Section 5.3 and 5.4. Since handling critical faults is necessary and sufficient (Remark 5.1) for safety, in the rest of this paper, the term fault is used to refer to critical faults.

5.1.6 RBR-Feasibility

A task set $\mathcal{T}$ is said to be feasible under restart based recovery (RBR-Feasible) if the following two conditions are satisfied; *(i)* there exists a schedule such that all jobs of all the critical tasks, or their potential re-executions, can complete successfully before their respective deadlines, even in the presence of a system-wide restart, occurring at any arbitrary time during execution. *(ii)* All jobs, including instances of non-critical tasks, can complete before their deadlines when no restart is performed.

5.2 FAULT DETECTION AND TASK RE-EXECUTION

As described in the previous section, a successful fault-detection approach must be able to detect any fault before the deadline of a *critical* task is missed, and to trigger the recovery procedure. Another key requirement is being able to correctly re-execute *critical* jobs that were affected by a restart.

Fault detection with watchdog (WD) timer: to explain the detection mechanism, we rely on the concept of *ideal worst-case response time*, i.e. the worst-case response time of a task when there are no restarts (and no re-executions) in the system. We use $\hat{\mathcal{R}}_i$ to denote the ideal worst-case response time of τ_i. $\hat{\mathcal{R}}_i$ can be derived using traditional response-time analysis, or with the analysis proposed in Section 5.3 and 5.4 by imposing all the overhead terms $O_y^x = 0$.

If no faults occur in the system, every instance of τ_i is expected to finish its execution within at most $\hat{\mathcal{R}}_i$ time units after its arrival time. This can be checked at runtime with a monitoring task. Recall that each critical job records its completion timestamp t_i^{comp} to NVM. The monitoring task checks the latest timestamp for τ_i at time instants $kT_i + \hat{\mathcal{R}}_i$. If $t_i^{comp} < kT_i$ it means that τ_i has not completed by its ideal worst-case response time. Hence, a restart needs to be triggered. A single WD can be used to always ensure a system reset if any of the critical tasks does not complete by its ideal worst-case response time. The following steps are performed:

1. Determine the next checkpoint instant t_{next} and checked critical task τ_i as follows:

$$t_{next} = \min_{i \leq n_c} \left(\lfloor (t - \phi_i)/T_i \rfloor T_i + \phi_i + \hat{\mathcal{R}}_i \right). \tag{5.1}$$

   In other words, t_{next} captures the earliest instant of time that corresponds to the elapsing of the ideal worst-case response time of some critical task τ_i;

2. Set the WD to restart the system after $t - t_{next} + \epsilon$ time units;

3. Terminate and set wake-up time at t_{next};

4. At wake-up, check if τ_i completed correctly: if t_i^{comp} obtained from NVM satisfies $t_i^{comp} \geq \lfloor (t - \phi_i)/T_i \rfloor T_i + \phi_i$, then acknowledge the WD so that it does not trigger a reset. Otherwise, do nothing, causing a WD-induced reset after ϵ time units.

5. Continue from Step 1 above.

Notice that this simple solution utilizes only one WD timer, and handles all the silent failures. The advantage of using hardware WD timers is that if any faults in the OS or other applications, prevent the time monitor task from execution, the WD which is already set, will expire and restart the system.

To determine which tasks to execute after a restart, we propose the following. Immediately after the reboot completes, a initializer task calculates the latest release time of each task τ_i using $\lfloor (t - \phi_i)/T_i \rfloor T_i + \phi_i$ where t is the current time retrieved from CLK. Next, it retrieves the last recorded completion time of the task, t_i^{comp}, from NVM. If $t_i^{comp} < \lfloor (t-\phi_i)/T_i \rfloor T_i + \phi_i$, then the task needs to be executed, and is added to the list of ready tasks. It is possible that a task completed its execution prior to the restart, but was not able to record the completion time due to the restart. In this case, the task will be executed again which does not impact the safety due to Assumption A7.

5.3 RBR-FEASIBILITY ANALYSIS

As mentioned in Section 5.2, re-execution of jobs impacted by a restart must not cause any other job to miss a deadline. Also, re-executed jobs need to meet their deadlines as well. The goal of this section is to present a set of sufficient conditions to reason about the feasibility of a given task set $\mathcal{T}$ in presence of restarts (RBR-feasibility). In particular, in Sections 5.3.1 and 5.3.2, we present a methodology that provides a sufficient condition for exact RBR-Feasibility analysis of preemptive and non-preemptive task sets.

Definition 5.2. Length of level-i preemption chain *at time t is defined as sum of the executed portions of all the tasks that are in the preempted or running state, and have a priority greater than or equal to π_i at t.* Longest level-i preemption chain *is the preemption chain that has the longest length over all the possible level-i preemption chains.*

For instance, consider a fully preemptive task set with four tasks; $C_1 = 1$, $T_1 = 5$, $C_2 = 3$, $T_2 = 10$, $C_3 = 2$, $T_3 = 12$, $C_4 = 4$, $T_3 = 15$, and $\pi_4 < \pi_3 < \pi_2 < \pi_1$. For this task set, the longest level-3 and level-4 preemption chains are 6 and 10, respectively.

5.3.1 Fully Preemptive Task Set

Under fully preemptive scheme, as soon as a higher priority task is ready, it preempts any lower priority tasks running on the processor. To calculate the worst-case response time of task τ_i, we have to consider the case where the restart incurs the longest delay on finishing time of the job. For a fully preemptive task set, this occurs when every task τ_k for $k \in \{2, \ldots, i\}$ is preempted immediately prior to its completion by τ_{k-1} and system restarts right before the completion of τ_1. In other words, when tasks τ_1 to τ_i form the longest level-i preemption chain. An example of this case is depicted in Figure 5.1. In this case, the restart and consequent re-execution causes a deadline miss at $t = 22$. The example uses only integer numbers for task parameters, hence tasks can be preempted only up to 1 unit of time before their completion. In the rest of the paper, we discuss our result assuming that tasks' WCETs are real numbers.

Theorem 5.1 provides RBR-feasibility conditions for a fully preemptive task set $\mathcal{T}$, under fixed priority scheduling.

Theorem 5.1. *A set of preemptive periodic tasks $\mathcal{T}$ is RBR-Feasible under fixed priority algorithm if the response time R_i of each task τ_i satisfies the condition: $\forall \tau_i \in \mathcal{T}, R_i \leq D_i$. R_i is obtained for the smallest value of k for which we have $R_i^{(k+1)} = R_i^{(k)}$.*

$$R_i^{(k+1)} = C_i + \sum_{\tau_j \in hp(\pi_i)} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + \mathcal{O}_i^p \tag{5.2}$$

where the restart overhead $\mathcal{O}_i^p$ on response time is

$$\mathcal{O}_i^p = \begin{cases} C_r + \sum_{\tau_j \in hp(\pi_i) \cup \{\tau_i\}} C_j & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{5.3}$$

Proof. First, note that Equation 5.2 without the overhead term $\mathcal{O}_i^p$, corresponds to the

classic response time of a task under fully preemptive fixed priority scheduling [77]. The additional overhead term represents the worst-case interference on the task instance under analysis introduced by restart time and the re-execution of the preempted tasks. We need to show that the overhead term can be computed using Equation 5.3. Consider the scenario in which every task τ_k is preempted by τ_{k-1} after executing for δ_i time units where $k \in \{2, ..., i\}$. And, a restart occurs after τ_1 executed for δ_1 time units. Due to the restart, all the tasks have to re-execute and the earliest time τ_i can finish its execution is $C_r + \delta_i + ... + \delta_1 + C_i + ... + C_1$. Hence, it is obvious that the later each preemption or the restart in τ_1 occurs, the more delay it creates for τ_i. Once a task has completed, it no longer needs to be re-executed. Therefore, the maximum delay of each task is felt immediately prior to the task's completion instant. Thus, the overhead is maximized when each τ_k is preempted by τ_{k-1} for $k \in \{2, .., i\}$ and restart occurs immediately before the end of τ_1.

As seen in this section, the worst-case overhead of restart-based recovery in fully preemptive setting occurs when system restarts at the end of longest preemption chain. Therefore, to reduce the overhead of restarting, length of the longest preemption chain must be reduced. In order to reduce this effect we investigate the non-preemptive setting in the following section.

5.3.2 Fully Non-Preemptive Task set

Under this model, jobs are not preempted until their execution terminates. At every termination point, the scheduler selects the task with the highest priority amongst all the ready tasks to execute. The main advantage of non-preemptive task set is that at most one task instance can be affected by restart at any instant of time.

Authors in [78] showed that in non-preemptive scheduling, the largest response time of a task does not necessarily occur in the first job after the critical instant. In some cases, the high-priority jobs activated during the non-preemptive execution of τ_i's first instance are pushed ahead to successive jobs, which then may experience a higher interference. Due to this phenomenon, the response time analysis for a task cannot be limited to its first job, activated at the critical instant, as done in preemptive scheduling, but it must be performed for multiple jobs, until the processor finishes executing tasks with priority higher than or equal to π_i. Hence, the response time of a task needs to be computed within the longest *Level-i Active Period*, defined as follows [79, 80].

Definition 5.3. *The* Level-i Active Period L_i *is an interval* $[a, b)$ *such that the amount of processing that still needs to be performed at time* t *due to jobs with priority higher than or*

equal to π_i, released strictly before t, is positive for all $t \in (a, b)$ and null in a and b. It can be computed using the following iterative relation:

$$L_i^{(q)} = B_i + C_i + \sum_{j \in hp(\pi_i)} \lceil L_i^{(q-1)}/T_j \rceil C_j + \mathcal{O}_i^{np} \tag{5.4}$$

Here, $\mathcal{O}_i^{np}$ is the maximum overhead of restart on the response time of a task. In the following we describe how to calculate this value. L_i is the smallest value for which $L_i^{(q)} = L_i^{(q-1)}$. This indicates that the response time of task τ_i must be computed for all jobs $\tau_{i,k}$ with $k \in [1, K_i]$ where $K_i = \lceil L_i/T_i \rceil$.

Theorem 5.2 describes the sufficient conditions under which a fault and the subsequent restart do not compromise the timely execution of the critical workload under fully non-preemptive scheduling. Notice that, as mentioned earlier, it is assumed that the schedule is resumed with the highest priority active job after restart.

Theorem 5.2. *A set of non-preemptive periodic tasks is RBR-feasible under fixed-priority if the response time R_i of each task τ_i, calculated through following relation, satisfies the condition: $\forall \tau_i \in \mathcal{T}; R_i \leq D_i$.*

$$R_i = \max_{k \in [1, K_i]} \{F_{i,k} - (k-1)T_i\} \tag{5.5}$$

where $F_{i,k}$ is the finishing time of job $\tau_{i,k}$ given by

$$F_{i,k} = S_{i,k} + C_i \tag{5.6}$$

Here, $S_{i,k}$ is the start time of job $\tau_{i,k}$, obtained for the smallest value that satisfies $S_{i,k}^{(q+1)} = S_{i,k}^{(q)}$ in the following relation

$$S_{i,k}^{(k+1)} = B_i + \sum_{\tau_j \in hp(\pi_i)} \left(\left\lfloor \frac{S_{i,k}^{(k)}}{T_j} \right\rfloor + 1 \right) C_j + \mathcal{O}_i^{np} \tag{5.7}$$

In Equation 5.7, term B_i is the blocking from low priority tasks and is calculated as $B_i = \max_{\tau_j \in lp(\pi_i)} \{C_j\}$. The term $\mathcal{O}_i^{np}$ represents the overhead on task execution introduced by restarts and is calculated as follows:

$$\mathcal{O}_i^{np} = \begin{cases} C_r + max\left\{\{C_j \mid j \in hp(\pi_i)\} \cup C_i\right\} & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{5.8}$$

Proof. Equation 5.7 and 5.6, without the restart overhead term $\mathcal{O}_i^{np}$, are proposed in [79, 80] to calculate the worst-case start time and response time of a task under non-preemptive setting.

We need to show that the overhead term can be computed using Equation 5.8. Under non-preemptive discipline, restart only impacts a single task executing on the CPU at the instant of restart. There are two possible scenarios that may result in the worst-case restart delay on finish time of task τ_i. First, when τ_i is waiting for the higher priority tasks to finish their execution, a restart can occur during the execution of one of the higher priority tasks τ_j and delay the start time τ_i by $C_r + C_j$. Alternatively, a restart can occur infinitesimal time prior to the completion of τ_i and cause an overhead of $C_r + C_i$. Hence, the worst-case delay due to a restart is caused by the task with the longest execution time among the task itself and the tasks with higher priority (Equation 5.8). The restart overhead is not included in the response-time of non-critical tasks ($\mathcal{O}_i^{np} = 0$ for $i > n_c$).

Figure 5.2: Example of fully non-preemptive system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, and restart at $t = 5 - \epsilon$ ($C_r = 0$). Restart and task re-execution causes a deadline miss at $t = 9$.

Unfortunately, under non-preemptive scheduling, blocking time due to low priority tasks, may cause higher priority tasks with short deadlines to be non-schedulable. As a result, when preemptions are disabled, there exist task sets with arbitrary low utilization that despite having the lowest restart overhead, are not RBR-Feasible. Figure 5.2 uses the same task parameters as in Figure 5.1. The plot shows that the considered task system is not schedulable under fully non-preemptive scheduling when a restart is triggered at $t = 5 - \epsilon$.

5.4 LIMITED PREEMPTIONS

In the previous section, we analyzed the RBR-Feasibility of task sets under fully preemptive and fully non-preemptive scheduling. Under full preemption, restarts can cause a significant overhead because the longest preemption chain can contains all the tasks. On the other hand, under non-preemptive scheduling, the restart overhead is minimum. However, due to additional blocking on higher priority tasks, some task sets, even with low utilization, are not schedulable.

In this section we discuss two alternative models with limited preemption. Limited preemption models are suitable for restart-based recovery since they enable the necessary preemptions for the schedulability of the task set, but avoid many unnecessary preemptions that occur in fully preemptive scheduling. Consequently, they induce lower restarting overhead and exhibit higher schedulability.

5.4.1 Preemptive tasks with Non-Preemptive Ending

As seen in the previous sections, reducing the number and length of preempted tasks in the longest preemption chain, can reduce the overhead of restarting and increase the RBR-Feasibility of task sets. On the other hand, preventing preemptions entirely is not desirable since it can impact feasibility of the high priority tasks with short deadlines. As a result, we consider a hybrid preemption model in which, a job once executed for longer than $C_i - Q_i$ time units, switches to non-preemptive mode and continues to execute until its termination point. Such a model allows a job that has mostly completed to terminate, instead of being preempted by a higher priority task. Q_i is called the size of non-preemptive ending interval of τ_i and $Q_i \leq C_i$. The model we utilize in this section, is a special case of the model proposed in [4] which aims to decrease the preemption overhead due to context switch in real-time operating systems. In Figure 5.3, we consider a task set with the same parameters as in Figure 5.1, where in addition task τ_3 has a non-preemptive region of length $Q_3 = 1$. The preemption chain that caused the system in Figure 5.1 to be non-schedulable cannot occur and the instance of the task becomes schedulable under restarts. With the same setup, Figure 5.4 considers the case when a reset occurs at $t = 9 - \epsilon$.

RBR-Feasibility Analysis

Theorem 5.3 provides the RBR-feasibility conditions of a task-set with non-preemptive ending intervals. In this theorem, $S_{i,k}$ represents the worst case start time of the non-

preemptive region of the re-executed instance of job $\tau_{i,k}$. Similarly, $F_{i,k}$ is used to represent the worst-case finish time. The arrival time of instance k of task $\tau_{i,k}$ is $(k-1)T_i$.

Theorem 5.3. *A set of periodic tasks $\mathcal{T}$ with non-preemptive ending regions of length Q_i, is RBR-Feasible under a fixed priority algorithm if the worst-case response time R_i of each task τ_i, calculated from Equation 5.9, satisfies the condition: $\forall \tau_i \in \mathcal{T}, R_i \leq D_i$.*

$$R_i = \max_{k \in [1, K_i]} \{F_{i,k} - (k-1)T_i\} \tag{5.9}$$

where

$$F_{i,k} = S_{i,k} + Q_i \tag{5.10}$$

and $S_{i,k}$ is obtained for the smallest value of q for which we have $S_{i,k}^{(q+1)} = S_{i,k}^{(q)}$ in the following

$$S_{i,k}^{(q+1)} = B_i + (k-1)C_i + C_i - Q_i + \sum_{\tau_j \in hp(\tau_i)} \left(\left\lfloor \frac{S_{i,k}^{(q)}}{T_j} \right\rfloor + 1 \right) C_j + \mathcal{O}_i^{npe} \tag{5.11}$$

Here, the term B_i is the blocking from low priority tasks and is calculated by

$$B_i = \max_{\tau_k \in lp(\pi_i)} \{Q_k\}. \tag{5.12}$$

$\mathcal{O}_i^{npe}$ is the maximum overhead of the restart on the response time and is calculated as follows:

$$\mathcal{O}_i^{npe} = \begin{cases} C_r + \mathcal{WCWE}(i) & i \leq n_c \\ 0 & i > n_c \end{cases} \tag{5.13}$$

where $\mathcal{WCWE}(i)$ is the worst-case amount of the execution that may be wasted due to the restarts. It is given by the following where $\mathcal{WCWE}(1) = C_1$ and

$$\mathcal{WCWE}(i) = C_i + max\Big(0, \mathcal{WCWE}(i-1) - Q_i\Big) \tag{5.14}$$

K_i in Equation 5.9 can be computed from Equation 5.4 by using $\mathcal{O}_i^{npe}$ instead of $\mathcal{O}_i^{np}$.

Proof. Authors in [3] show that the worst-case response time of task τ_i is the maximum difference between the worst case finish time and the arrival time of the jobs that arrive within the level-i active period (Equation 5.9).

Hence, we must compute the worst-case finish time of job $\tau_{i,k}$ in the presence of restarts. When a restart occurs during the execution of $\tau_{i,k}$ or while it is in preempted state, $\tau_{i,k}$ needs to re-execute. Therefore, the finish time of the $\tau_{i,k}$ is when the re-executed instance

completes. As a result, to obtain the worst-case finish time of $\tau_{i,k}$, we calculate the response time of each instance when a restart with longest overhead has impacted that instance. We break down the worst-case finish time of $\tau_{i,k}$ into two intervals: the worst-case start time of the non-preemptive region of the re-executed job and the length of the non-preemptive region, Q_i (Equation 5.10). $S_{i,k}$ in Equation 5.10, is the worst-case start time of non-preemptive region of job $\tau_{i,k}$ which can be iteratively obtained from Equation 5.11. Equation 5.11 is an extension of the start time computation from [80]. In the presence of non-preemptive regions, an additional blocking factor B_i must be considered for each task τ_i, equal to the longest non-preemptive region of the lower priority tasks. Therefore, the maximum blocking time that τ_i may experience is $B_i = \max_{\tau_j \in lp(\pi_i)} \{Q_j\}$. B_i is added to the worst-case start time of the task in Equation 5.11.

For a task τ_i with the non-preemptive region of size Q_i, there are two cases that may lead to the worst-case wasted time. First case is when the system restarts immediately prior to the completion of τ_i, in which case the wasted time is C_i. Second case occurs when τ_i is preempted immediately before the non-preemptive region begins (*i.e.,* at $C_i - Q_i$) by the higher priority task τ_{i-1}. In this case, the wasted execution is $C_i - Q_i$ plus the maximum amount of the execution of the higher priority tasks that may be wasted due to the restarts (*i.e.,* $\mathcal{WCWE}(i-1)$). The worst-case wasted execution is the maximum of these two values *i.e.,* $\mathcal{WCWE}(i) = max(C_i, C_i - Q_i + \mathcal{WCWE}(i-1)) = C_i + max(0, \mathcal{WCWE}(i-1) - Q_i)$. Similarly, $\mathcal{WCWE}(i-1)$ can be computed recursively.

Optimal Size of Non-Preemptive Regions

RBR-Feasibility of a taskset depends on the choice of Q_is for the tasks. In this section, we present an approach to determine the size of non-preemptive regions Q_i for the tasks to maximize the RBR-Feasibility of the task set.

First, we introduce the the notion of *blocking tolerance* of a task β_i. β_i is the maximum time units that task τ_i may be blocked by the lower priority tasks, while it can still meet its deadline. Algorithm 5.1, uses binary search and the response time analysis of task (from Theorem 5.3) to find β_i for a task τ_i.

In Algorithm 5.1, $R_{i,B_i=middle}$ is computed as described in Theorem 5.3 (Equation 5.9), where instead of using the B_i from Equation 5.12, the blocking time is set to the value of *middle*.

Note that, if Algorithm 5.1 cannot find a β_i for task τ_i, this task is not schedulable at all. This indicates that there is not any selection of Q_is that would make $\mathcal{T}$ RBR-Feasible.

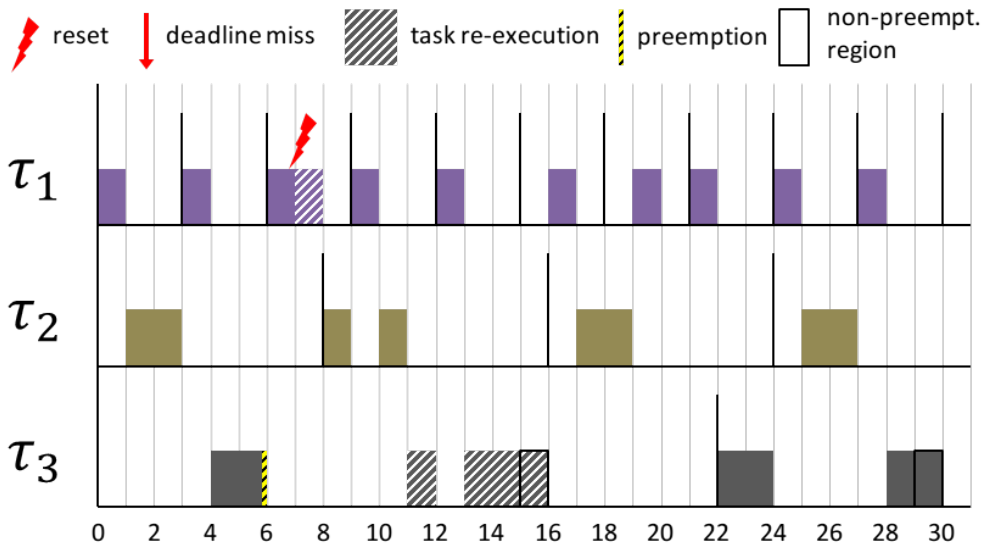


Figure 5.3: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_3 has a non-preemptive region of size $Q_3 = 1$. Restart occurs at $t = 7 - \epsilon$ ($C_r = 0$). The task set is schedulable with restarts.

Algorithm 5.1: Binary Search for Finding β_i

```
1 FindBlockingTolerance(τ_i, 𝒯, Q_1, ..., Q_i)
2     start = 0; end = T_i /* Initialize the interval */
3     if R_i(start) > T_i then return τ_i Not Schedulable;
4     while end - start > ε do
5         middle = (start + end)/2
6         if R_{i,B_i=middle} > T_i then end = middle ;
7         else start = middle
8     end
9     return β_i = start;
```

Given that task τ_1 has the highest priority, it may not be preempted by any other task; hence we set $Q_1 = C_1$. The next theorem shows how to drive optimal Q_i for the rest of the tasks in $\mathcal{T}$. The results are optimal, meaning that if there is at least one set of Q_is under which $\mathcal{T}$ is RBR-Feasible, it will find them.

Theorem 5.4. *The optimal set of non-preemptive interval Q_is of tasks τ_i for $2 \leq i \leq n$ is given by:*

$$Q_i = min\{min\{\beta_j \mid j \in hp(\pi_i)\}, C_i\} \tag{5.15}$$

assuming that $\beta_j \geq 0$ for $j \in hp(\pi_i)$.

Proof. Increasing the length of Q_i for a task reduces the response time in two ways. First, from Equation 5.11, increasing Q_i reduces the start time of the job $S_{i,k}$ which reduces the

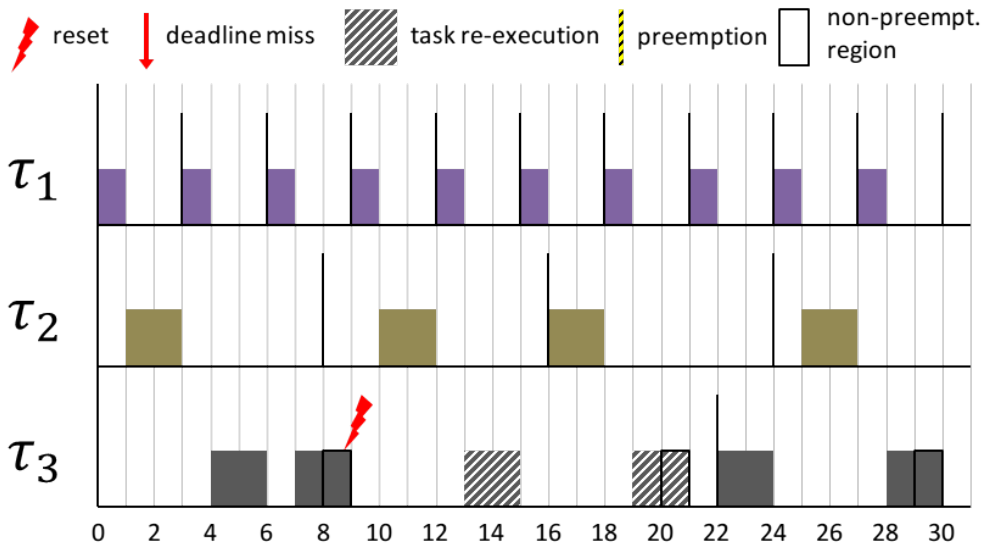


Figure 5.4: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_3 has a non-preemptive region of size $Q_3 = 1$. Restart occurs at $t = 9 - \epsilon$ ($C_r = 0$). The task set is schedulable with restarts.

finish time and consequently the response time of τ_i. Second, from Equation 5.14, increasing Q_i reduces the restart overhead $\mathcal{O}_i^{npe}$ on the task and lower priority tasks which in turn reduces the response time. Thus Q_i may increase as much as possible up to the worst-case execution time C_i; $Q_i \leq C_i$. However, the choice of Q_i must not make any of the higher priority tasks unschedulable. As a result, Q_i must be smaller than the smallest blocking tolerance of all the tasks with higher priority than π_i; $Q_i \leq min\{\beta_j | j \in hp(\pi_i)\}$. Combining these two conditions results in the relation of Equation 5.15.

5.4.2 Preemption Thresholds

In the previous section, we discussed non-preemptive endings as a way to reduce the length of the longest preemption chain and decrease the overhead of restarts. In this section, we discuss an alternative approach to reduce the number of tasks in the longest preemption chain and thus reduce the overhead of restart-based recovery.

To achieve this goal, we use the notion of preemption thresholds which has been proposed in [3]. According to this model, each task τ_i is assigned a nominal priority π_i and a preemption threshold $\lambda_i \geq \pi_i$. In this case, τ_i can be preempted by τ_h only if $\pi_h > \lambda_i$. At activation time, priority of τ_i is set to the nominal value π_i. The nominal priority is maintained as long as the task is kept in the ready queue. During this interval, the execution of τ_i can be delayed by all tasks with priority $\pi_h > \pi_i$, and by at most one lower priority

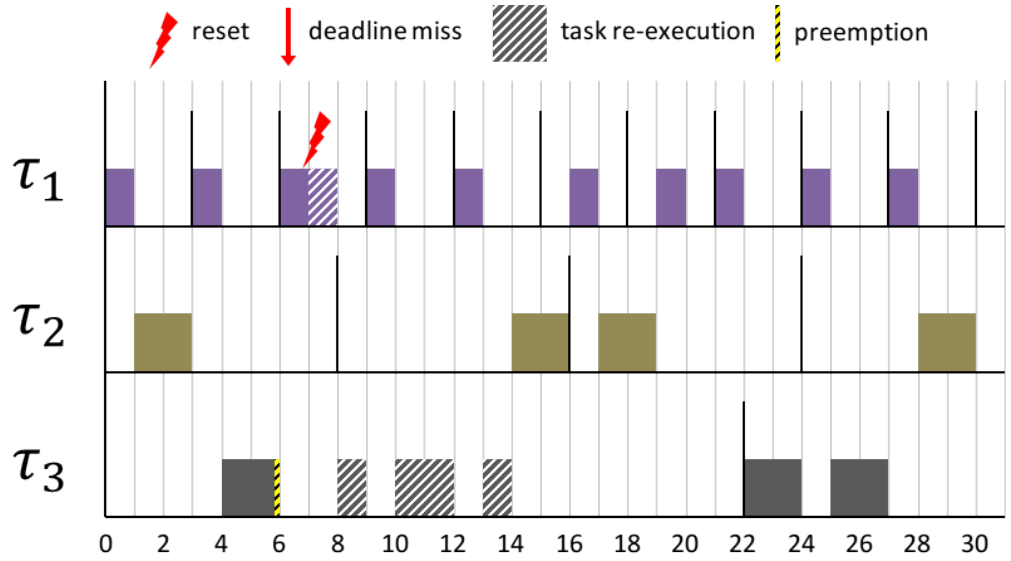


Figure 5.5: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_2 and τ_3 have a preemption threshold of $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. Restart occurs at $t = 7 - \epsilon$ ($C_r = 0$). In this case, the task set remains schedulable.

task with threshold $\lambda_l \geq \pi_i$. When all such tasks complete, τ_i is dispatched for execution, and its priority is raised to λ_i. During execution, τ_i can be preempted by tasks with priority $\pi_h > \lambda_i$. When τ_i is preempted, its priority is kept at λ_i.

Restarts may increase the response time of $\tau_{i,k}$ in one of two ways; A restart may occur after the arrival of the job but before it has started, delaying its start time $S_{i,k}$. Alternatively, the system can be restarted after the job has started. We use $\mathcal{O}_i^{pt,s}$ to denote the worst-case overhead of a restart that occurs before the start time of a job in task sets with preemption thresholds. And, $\mathcal{O}_i^{pt,f}$ is used to represent the worst-case overhead of a restart that occurs after the start time of a job in task sets with preemption thresholds.

In Figure 5.5, we consider a task set with the same parameters as in Figure 5.1 where in addition τ_2 and τ_3 have a preemption threshold equal to $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. This assignment is effective to prevent a long preemption chain, and the jobs do not miss their deadline when the restart occurs at $t = 7 - \epsilon$. Notice that, the task set is still not RBR-Feasible since if the restart occurs at $t = 9 - \epsilon$, some job will miss the deadline, as shown in Figure 5.6.

Theorem 5.5. *For a task set with preemption thresholds under fixed priority, the worst-case overhead of a restart that occurs after the start of the job $\tau_{i,k}$ is $\mathcal{O}_i^{pt,f} = C_r + \mathcal{WCWE}(i)$ where*

$$\mathcal{WCWE}(i) = C_i + \max\{\mathcal{WCWC}(j) \mid \tau_j \in hp(\lambda_i)\} \tag{5.16}$$

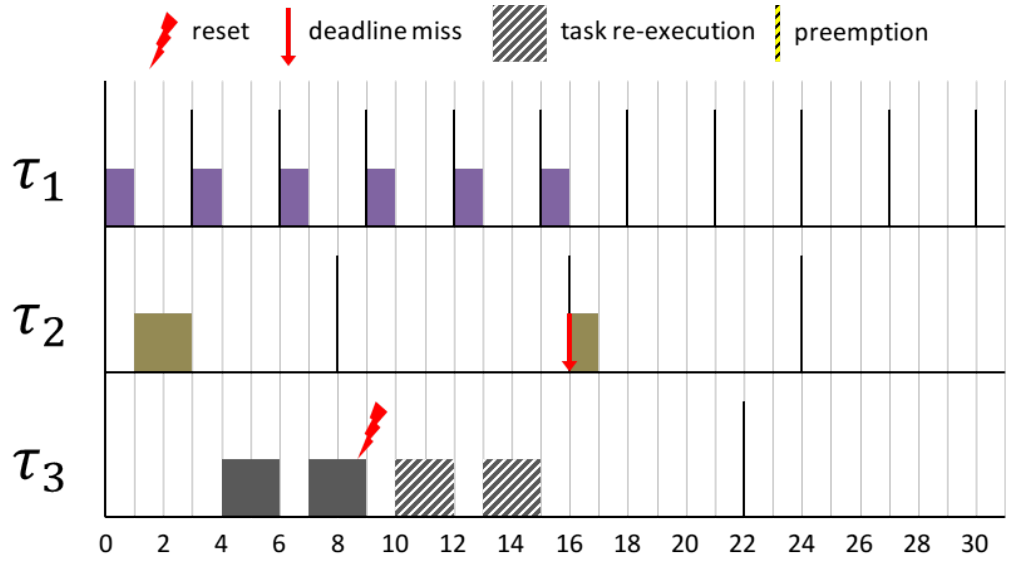


Figure 5.6: Example of system with 3 tasks $\tau_1 = (1, 3); \tau_2 = (2, 8); \tau_3 = (4, 22)$, where τ_2 and τ_3 have a preemption threshold of $\lambda_2 = 1$ and $\lambda_3 = 2$, respectively. Restart occurs at $t = 9 - \epsilon$ ($C_r = 0$). The task set is not schedulable.

Here, $\mathcal{WCWC}(1) = C_1$.

Proof. After a job $\tau_{i,k}$ starts, its priority is raised to λ_i. In this case, the restart will create the worst-case overhead if it occurs at the end of longest preemption chain that includes τ_i and any subset of the tasks with $\pi_h > \lambda_i$. Equation 5.16 uses a recursive relation to calculate the length of longest preemption chain consisting of τ_i and all the tasks with $\pi_h > \lambda_i$.

Theorem 5.6. *For a task set with preemption thresholds under fixed priority, a restart occurring before the start time of a job $\tau_{i,k}$, can cause the worst-case overhead of*

$$\mathcal{O}_i^{pt,s} = C_r + \max\{\mathcal{WCWE}(j) \mid \tau_j \in hp(\pi_i)\} \tag{5.17}$$

where $\mathcal{WCWE}(j)$ can be computed from Equation 5.16.

Proof. Start time of a task can be delayed by a restart impacting any of the tasks with priority higher than π_i. Equation 5.17 recursively finds the longest possible preemption chain consisting of any subset of tasks with $\pi_h > \pi_i$.

Due to the assumption of one fault per hyper-period, each job may be impacted by at most one of $\mathcal{O}_i^{pt,f}$ or $\mathcal{O}_i^{pt,s}$, but not both at the same time. Hence, we compute the finish time of the task once assuming that the restart occurs before the start time *i.e.,* $\mathcal{O}_i^{pt,f} = 0$, and another time assuming it occurs after the start time *i.e.,* $\mathcal{O}_i^{pt,s} = 0$. Finish time in these

two cases is referred respectively by $F^s_{i,k}$ (restart before the start time) and $F^f_{i,k}$ (restart after the start time).

We expand the response time analysis of tasks with preemption thresholds from [3], considering the overhead of restarting. In the following, $S_{i,k}$ and $F_{i,k}$ represent the worst case start time and finish time of job $\tau_{i,k}$. And, the arrival time of $\tau_{i,k}$ is $(k-1)T_i$. The worst-case response time of task τ_i is given by:

$$R_i = \max_{k \in [1,K_i]} \left\{ \max\{F^s_{i,k}, F^f_{i,k}\} - (k-1)T_i \right\} \tag{5.18}$$

Here, K_i can be obtained from Equation 5.4 by using $max(\mathcal{O}^{pt,f}_i, \mathcal{O}^{pt,s}_i)$ instead of $\mathcal{O}^{np}_i$. A task τ_i can be blocked only by lower priority tasks that cannot be preempted by it, that is:

$$B_i = \max_j \{C_j \mid \pi_j < \pi_i \leq \lambda_j\} \tag{5.19}$$

To compute finish time, $S_{i,k}$ is computed iteratively using the following equation [3]:

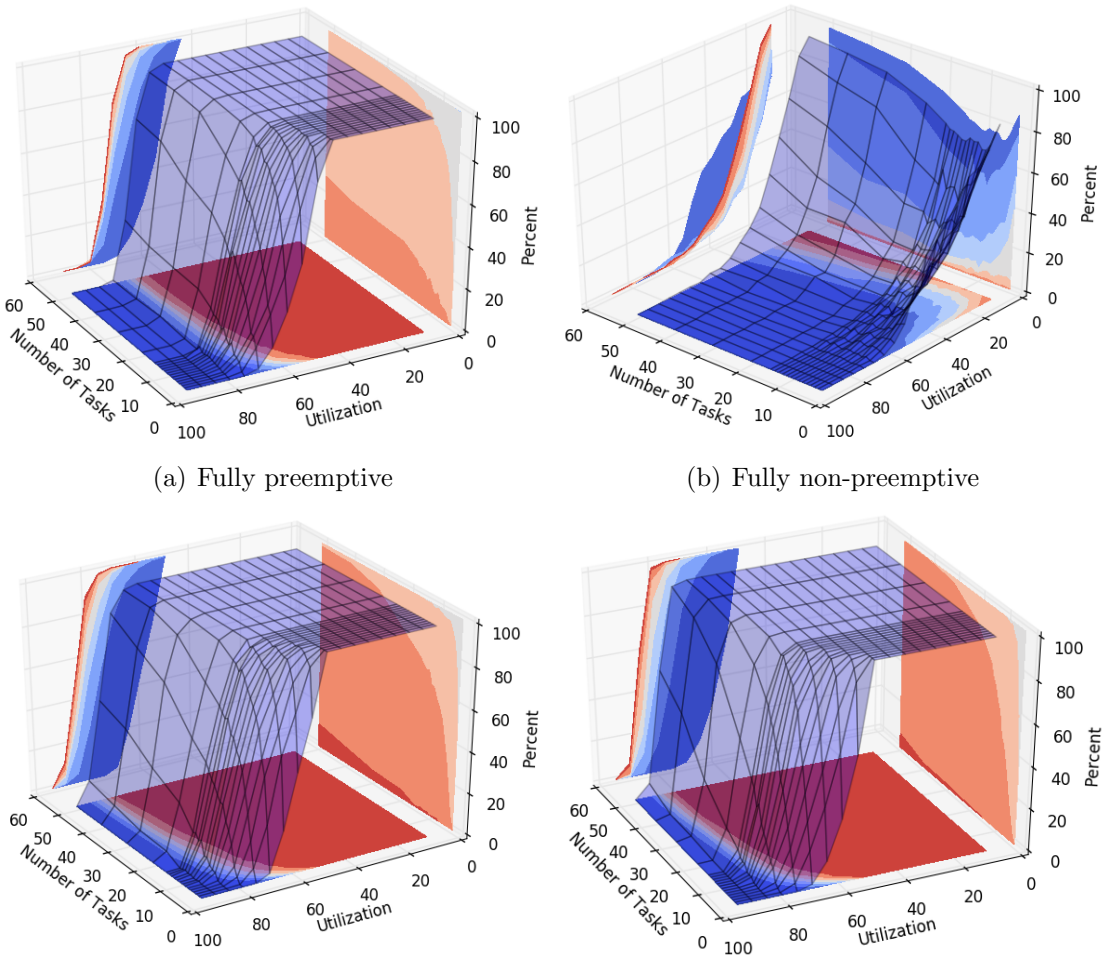
$$S^{(q)}_{i,k} = B_i + (k-1)C_i + \sum_{j \in hp(\pi_i)} \left(1 + \left\lfloor \frac{S^{(q-1)}_{i,k}}{T_j} \right\rfloor \right) C_j + \mathcal{O}^{pt,s}_i \tag{5.20}$$

Once the job starts executing, only the tasks with higher priority than λ_i can preempt it. Hence, the $F_{i,k}$ can be derived from the following:

$$F^{(q)}_{i,k} = S_{i,k} + C_i + \sum_{j \in hp(\lambda_i)} \left(\left\lceil \frac{F^{(q-1)}_{i,k}}{T_j} \right\rceil - \left(1 + \left\lfloor \frac{S_{i,k}}{T_j} \right\rfloor \right) \right) C_j + \mathcal{O}^{pt,f}_i \tag{5.21}$$

Task set $\mathcal{T}$ is considered RBR-Feasible if $\forall \tau_i \in \mathcal{T}, R_i \leq T_i$.

RBR-Feasibility of a task set depends on the choice of λ_is for the tasks. In this paper, we use a genetic algorithm to find a set of preemption thresholds to achieve RBR-Feasibility of the task-set. Although this algorithm can be further improved to find the optimal threshold assignments, the proposed genetic algorithm achieves acceptable performance, as we show in Section 5.5.



(a) Fully preemptive

(b) Fully non-preemptive

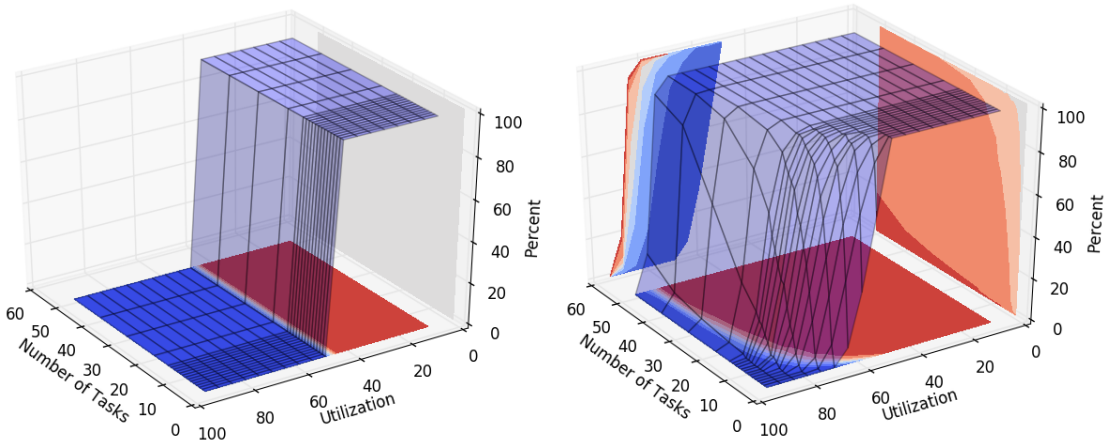
(c) Non-preemptive ending intervals.

(d) Preemption thresholds.

Figure 5.7: Minimum Period: 10, Maximum Period: 1000

5.5 EVALUATION

In this section, we compare and evaluate the four fault-tolerant scheduling strategies discussed in this paper. In order to evaluate the practical feasibility of our approach, we have also performed a preliminary proof-of-concept implementation on commercial hardware (i.MX7D platform) for an actual 3 degree-of-freedom helicopter. We tested logical faults, application faults and system-level faults and demonstrated that the physical system remained within the admissible region. Due to space constraints, we omit the description and evaluation of our implementation and refer to [81] for additional details.



(a) Fully preemptive

(b) Fully non-preemptive

(c) Non-preemptive ending intervals.

(d) Preemption thresholds.

Figure 5.8: Minimum Period: 900, Maximum Period: 1000

5.5.1 Evaluating Performance of Scheduling Schemes

In this section, we evaluate the performance of four fault-tolerant scheduling schemes that are discussed in this paper. For each data point in the experiments, 500 task sets with the specified utilization and number of tasks are generated. Then, RBR-feasibility of the task sets are evaluated under four discussed schemes; fully preemptive, fully non-preemptive, non-preemptive ending intervals, and preemption thresholds. In order to evaluate performance of the scheduling schemes, all the tasks in the analysis are assumed to be part of the critical workload. Priorities of the tasks are assigned according to the periods, so a task with shorter period has a higher priority.

The experiments are performed with two sets of parameters for the periods of the task sets. In the first set of experiments (Figure 5.7), task sets are generated with periods in the range of 10 to 1000 time units. In the second set (Figure 5.8), tasks have a period in the

range of 900 to 1000 time units. As a result, tasks in the first experiment have more diverse set of periods than the second one.

As shown in Figure 5.7(a) and 5.8(a), all the task sets with utilization less than 50% are RBR-feasible under preemptive scheduling. This observation is consistent with the results of [28] which considers preemptive task sets under rate monotonic scheduling with a recovery strategy similar to ours (re-executing all the unfinished tasks), and shows that all the task sets with utilization under 50% are schedulable.

Moreover, a comparison between Figure 5.7(a) and 5.8(a) reveals that fully preemptive setting performs better when tasks in the task set have diverse rates. To understand this effect, we must notice that the longest preemption chain for a task in preemptive setting, consists of the execution time of all the tasks with a higher priority. Therefore, under this scheduling strategy, tasks with low priority are the bottleneck for RBR-feasibility analysis. When the diversity of the periods is increased, lower priority tasks, on average, have much longer periods. As a result, they have a larger slack to tolerate the overhead of restarts compared to the lower priority tasks in task sets with less diverse periods. Hence, more task sets are RBR-feasible when a larger range of periods is considered.

On the contrary, when tasks have more diverse periods, non-preemptive setting performs worse (Figure 5.7(b) and 5.8(b)). This is because, with diverse periods, tasks with shorter periods (and higher priorities) experience longer blocking times due to low priority tasks with long execution times.

As the figures show, scheduling with preemption thresholds and non-preemptive intervals in both experiments yield better performance than preemptive and non-preemptive schemes. This effect is expected because the flexibility of these schemes allows them to decrease the overhead of restarts by increasing the non-preemptive regions, or by increasing the preemption thresholds while maintaining the feasibility of the task sets. Tasks under these disciplines exhibit less blocking and lower restart overhead.

Preemption thresholds and non-preemptive endings in general demonstrate comparable performance. However, in task sets with very small number of tasks (2-10 task), scheduling using non-preemptive ending intervals performs slightly better than preemption thresholds. This is due to the fact that, with small number of tasks, the granularity of the latter approach is limited because few choices can be made on the tasks' preemption thresholds. Whereas, the length of non-preemptive intervals can be selected with a finer granularity and is not impacted by the number of tasks.

5.6 SUMMARY AND DISCUSSION

In this chapter, we constructed and analyzed the required safety conditions under which all the critical tasks will meet their deadlines if the system restarts and the interrupted tasks re-execute. We analyzed the performance of these strategies for various task sets and proposed two techniques to improve their schedulability. We also implemented a proof-of-concept prototype and tested it against faults in the application-layer and RTOS.

The schedulability-based design differs from the controller-based fault-tolerant approach in two key areas. First, this approach focuses on the schedulability of the controller tasks and does not require any modification to the controller design (as long as it is constructed following the Simplex methodology). Whereas, in the controller-based design discussed in Chapter 4, the complexity of constructing the base controller increases at an exponential rate with regards to the number of dimensions of the system dynamics. Schedulability-based design, therefore, can be applied to certain high-dimensional physical plants where computing a base controller is computationally infeasible due to the high number of dimensions.

On the other hand, this approach is limited for CPS with longer restart times (relative to the execution frequency of the control tasks). The schedulability analyses of this chapter demonstrate that if the restart time of the system is larger than the minimum of all the periods of all the critical tasks, the task set is not schedulable. As the restart time of the platform increases, the feasibility of the task set decreases. The proposed solution in its current form, despite being useful for many platforms, may not suit platforms with a long restart time.

Another point worth mentioning is that the current restart time of many platforms is not optimal, mainly because reducing the reboot time of the platform has not been investigated. One future direction of research is creating a multi-stage booting solution for multi-core platforms to mitigate this problem. Our possible idea is to boot one core with the bare minimum requirements to execute the SC in the quickest possible time. The SC can keep the system safe, while the real-time or general purpose OS boots on the other cores. Once the boot process is complete, the control switches to the controllers running on the OS.

CHAPTER 6: SINGLE NODE CPS SECURITY

Some of the recent attacks on cyber-physical systems (CPS) are focused on causing physical damage to the plants. Such intruders make their way into the system using cyber exploits but then initiate actions that can destabilize and even damage the underlying (physical) systems. Examples of such attacks on medical pacemakers [82], or vehicular controllers [83] exist in literature. Any damage to such physical systems can be catastrophic – to the systems, the environment or even humans. The drive towards remote monitoring/control (often via the Internet) only exacerbates the safety-related security problems in such devices.

When it comes to security, many techniques focus on preventing the software platform from being compromised or detecting the malicious behavior as soon as possible and taking recovery actions. Unfortunately, there are always unforeseen vulnerabilities that enable intruders to bypass the security mechanisms and gain administrative access to the controllers. Once an attacker gains such access, all bets are off with regards to the safety of the physical subsystem. For instance, the control program can be prevented from running, either entirely or even in a timely manner, sensor readings can be blocked or tampered with, and false values forwarded to the control program and similarly actuation commands going out to the plants can be intercepted/tampered with, system state data can be manipulated, *etc.* These actions, either individually or in conjunction with each other, can result in significant damage to the plant(s). At the very least, they will significantly hamper the operation of the system and prevent it from making progress towards its intended task.

In this chapter, I *develop analytical methods that can formally guarantee the baseline safety of the physical plant even when the controller unit's software has been entirely compromised.* The main idea of our work in this chapter is to carry out consecutive evaluations of physical safety conditions, inside secure execution intervals, separated in time such that an attacker with full control will not have enough time to destabilize or crash the physical plant in between two consecutive intervals. We refer to these intervals as "Secure Execution Intervals, SEI". In this chapter, the time between consecutive SEIs is dynamically calculated in real time, based on the mathematical model of the physical plant and its current state. The key to providing such formal guarantees is to make sure that each SEI takes places before an attacker can cause any physical damage.

To further clarify the approach, consider a simplified drone example. The base-line safety for a drone is to not crash into the ground. Using a mathematical model of the drone, we demonstrate, in Section 6.2.2, how to calculate the shortest time that an adversary with full

control over all the actuators would need to take the drone into zero altitudes (an unsafe state) from its current state (*i.e.,* current velocity and height). The key is, once inside the SEI, to schedule the starting point of the upcoming SEI *before* the shortest possible time to reach the ground. During the SEI, depending on whether the drone was compromised or not, it will be either stabilized and recovered or, it will be allowed to resume its normal operation. With this design in place, despite a potentially compromised control software, the drone will remain above the ground (safe).

Providing formal safety guarantees, even for the simple example above is non-trivial and challenging. For instance, an approach is needed to compute the shortest time to reach the ground at run-time. Each SEI must be scheduled to take place at a state that not only is safe (before hitting the ground), but also such that the controller can still stabilize the drone from that velocity and altitude, considering the limits of drone motors. Mechanisms are needed to prevent attackers from interfering with the SEIs in any way possible. In this chapter, we address all the challenges required to provide safety.

One of the primary technical necessities for the proposed design is a trusted execution environment where the integrity of the executed code can be maintained. In this chapter, we utilize two different approaches to achieve this goal; *(i) restart-based implementation* that makes use of full system restarts and software reloads *(ii) TEE-based implementation* that utilizes Trusted Execution Environment (TEE) such as ARM TrustZone [5] or Intel's Trusted Execution Technology (TXT) [6] that are available in some hardware platforms.

Under the restart-based implementation, the control platform is restarted in each cycle and the uncompromised image of the controller software is reloaded from read-only storage. Restarting the platform enables us to *(i)* eliminate all the possible transformations carried out by the adversary during the previous execution cycle[1] and also *(ii)* provides a window for trusted computation in an untrusted environment that we use to compute the next SEI triggering time (Section 6.2.1). This design utilizes an external HW timer to trigger the restart at the scheduled times. This simple design prevents the adversary from interfering with the scheduled restarting event.

Another alternative approach (introduced in this chapter) is to enable the SEIs to use TEE features that are available in HW platforms. In particular, we use ARM TrustZone [5] and LTZVisor [84] – a hypervisor based on TrustZone (Section 6.3.1). The TEE-assisted implementation does not require the platform to be restarted in every SEI cycle. Thus,

[1]It is possible that the adversary launches a new instance of the attack after a restart. Yet, the plant is protected against each attack instance and *malicious states are not carried across restarts.* As a result, the proposed approach is able to prevent the attacker from damaging the system every time and guarantees safety of the entire system.

there is no restarting overhead and, additionally, the controller state is not lost with every SEI cycle. This design can significantly improve the applicability of our method to physical plants with faster dynamics. As we have shown in the evaluation section, the maneuverability region of the 3DOF plant is increased by 234 percent when the controller is implemented by the TEE-based method.

For some CPS applications, one of the above implementation options might be a more suitable choice than the other one. If the physical plant has high-speed dynamics – relative to the restart time of the platform – or if prior state of the controller is necessary to carry out the mission – *e.g.,* authentication with ground control – the TEE-based option the reasonable choice. On the other hand, restart-based implementation is feasible for low-cost micro-controllers whereas platforms equipped with TEE are generally more expensive. Furthermore, many of the CPS applications have physical plants with slow physical dynamics (compared to the restart time of their embedded platform) and the restart-based implementation will perform just as good as the TEE-based implementation (as shown in Section 6.4.4). For such cases, the restart-based implementation is a better choice and the TEE-assisted implementation might only unnecessarily increase the cost and complexity of the system.

In summary, the contributions of this chapter are:

1. We introduce a design method for embedded control platforms with *formal guarantees on the base-line safety of the physical subsystem* when the software is under attack.

2. We propose a restart-based design implementation that enables trusted computation in an untrusted environment using platform restarts and common-off-the-shelf (COTS) components, without requiring chip customizations or specific hardware features.

3. We propose an alternative design implementation using TEE features that eliminates the restarting overhead and enables the core safety-guarantees to be provided on more challenging physical plants.

4. We have implemented and tested our approach against attacks through a prototype implementation for a realistic physical plant and a hardware-in-the-loop simulation. We compare both design implementation options and illustrate their use cases.

6.1 APPLICATIONS, THREATS AND ADVERSARIES

This chapter focuses on end-point devices that control and drive a safety-critical physical plant *i.e.,* the plant has safety conditions that need to be respected at all times. Components

such as sensing nodes that do not directly control a physical plant are not in the scope of this work. Safety requirements of the plant are defined as an admissible region in a connected subset of the state space. If the physical plant reaches the states outside of the admissible region, it could damage itself as well as the surrounding environment. Thus, to preserve the physical safety, the plant must only operate within the admissible region.

6.1.1 Adversary and Threat Model

Embedded controllers of CPS face threats in various forms depending on the system and the goals of the attacker. The particular attacks that we aim to thwart in this work are those that target damaging the physical plant. In this chapter, we assume attackers require an *external interface* such as the network, the serial port or the debugging interface to intrude into the platform. We assume that the attackers do not have physical access to the platform. Once a system is breached, we assume the attacker has full control (root access) over the software (non-secure world), actuators, and peripherals.

The following assumptions are made about the platform and the adversary's capabilities:

i) *Integrity of original software image:* We assume that the original images of the system software *i.e.,* real-time operating system (RTOS), control applications, and other components are not malicious. These components, however, may contain security vulnerabilities that could be exploited to initiate attacks.

ii) *Read-only storage for the original software image*: We assume that the original trusted image of the system software is stored on a read-only memory unit (*e.g.,* E^2PROM). This content is not modifiable at runtime by anyone including adversary. Updating this image requires physical access and is completed off-line when the system is not operating[2].

iii) *Trusted Execution Environment (TEE)*: Hardware-assisted TEEs such as TrustZone partition the platform into a secure world and a non-secure world. Resources (*i.e.,* code and data) in the secure world are isolated from the non-secure world and are only accessible by the software running in the secure world. A compromise in the non-secure world may not affect the execution and data in the secure world. In this chapter, we assume that the software in the secure world is trusted from the beginning and may

[2]This is common for many safety-critical IoT systems such as medical devices and some components in automotive systems – to prevent from runtime malfunctioning due to unwanted firmware corruption at the time of update and well as to prevent the adversary from tampering with the system's image remotely)

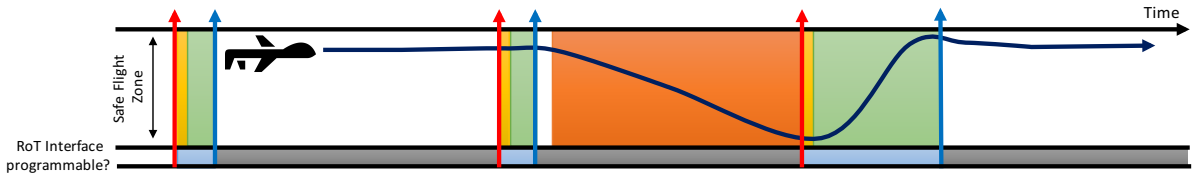


Figure 6.1: An example sequence of events for the restart-based implementation of the SEI. White: mission controller is in charge and platform is not compromised. Yellow: system is undergoing a restart. Green: SEI is active, SC and `find_safety_window` are running in parallel. Orange: adversary is in charge. Blue: RoT accepts new restart time. Gray: RoT does not accept new restart time. Red arrow: RoT triggers a restart. Blue arrow: SEI ends, the next restart time is scheduled in RoT, and the mission controller starts.

not be compromised (in our design, the secure world only interacts with sensors and actuators and does not have an exposed interface that can be a point of exploitation).

iv) Immediately after a reboot, as long as the external interfaces of the device (*i.e.,* network, debugging interface) remain disabled[3], software running on the platform is assumed to be uncorrupted.

v) *Integrity of Root of Trust (RoT)*: RoT – which is only necessary for the restart-based implementation – is an isolated hardware timer responsible for issuing the restart signal at designated times. As shown in Section 6.2.1, it is designed to be programmable *only* once in each execution cycle and *only* during an interval that we call the SEI.

Additionally, we assume that the system is not susceptible to external sensor spoofing or jamming attacks (*e.g.,* broadcasting incorrect GPS signals, electromagnetic interference on sensors *etc.*). An attacker may, however, spoof the sensor readings within the OS or applications. Our approach does not protect from data leak related attacks such as those which aim to steal secrets, monitor the activities, or violate the privacy. Our design does not protect from network attacks such as man-in-the-middle or denial-of-service attacks that restrict the network access. An attacker may enter the system via any external interface (*e.g.,* a telemetry channel, a network interface) and use known vulnerabilities such as buffer overflow or code injection to manipulate the system. However, as we show, the physical plant remains safe during such attacks.

6.2 METHODOLOGY

To explain our approach, let us assume that it is possible to create secure execution intervals (SEI) during which we can trust that the system is going to execute uncompromised software and adversary cannot interfere with this execution in any way. Under such assumption, we will show that it is possible to guarantee that a physical plant will remain within its admissible states as long as the following conditions remain true: *(i)* the timing between these intervals are separated such that, due to the physical inertia, the plant will not reach an inadmissible state until the beginning of the consequent SEI. *(ii)* The state of the plant at the beginning of the following SEI will be such that the SC can still stabilize the system. Under these conditions, the plant will be safe in between two SEIs (due to condition 1). If an adversary pushes the system close to the boundaries of inadmissible states, during the following SEI, we can switch to SC, and it can stabilize the plant (condition 2).

In the rest of this section, we present an analytical framework that shows how appropriately timed separations between the consequent SEIs guarantee the physical safety. Additionally, we show how these time values can be calculated in run-time. Finally, we discuss two different mechanisms – restart-based implementation and TEE-assisted implementation – to enable a trusted computation environment – SEI – during which the time intervals between SEI will be computed, without any adversarial interference.

6.2.1 Restart-based Secure Execution Intervals (SEI)

One essential element of the approach introduced in this chapter is the run-time computation of the time separation between consecutive executions of the safety-critical tasks – the tasks that evaluate the safety conditions (next section) and stabilize the plant if necessary. The ultimate safety guarantees of our approach depend on the integrity of these computations. To achieve safety, therefore, it is essential to have a means to completely protect these tasks from any adversarial interference – adversary should not be able to stop or delay the execution or, corrupt the results of the computations. In this work, we use the term Secure Execution Interval (SEI) to refer to execution intervals during which the integrity of the code is preserved.

One way to create SEIs in an untrusted environment is to rely on the *full platform restarts* and the *software reloads*. The procedure is as follows. For each SEI, the platform needs to restart entirely and then immediately load the clean software image from the read-only

[3]This is achieved by not initiating a socket connection, not reading/writing from/to any of the ports and not performing any of the hand shaking steps.

storage. Additionally, after the restart, all the external interfaces of the platform – those that might be an exploitation point for external adversaries – will remain disabled. As soon the platform boots, it can execute the safety-related tasks trustworthily and produce correct results. Once the execution of the critical tasks is finished, the time to trigger the following restart – the next SEI – is scheduled. Finally, the SEI ends, the external interfaces are activated, and the mission controller and other necessary components are launched.

An additional mechanism is necessary to schedule a restart and trigger it such that the adversary cannot prevent it. We designate a separate HW module, called root-of-trust (RoT) to do this. RoT is essentially an external timer that can send a restart signal to the HW restart pin of the controller board at the scheduled time. It has an interface that allows the main controller to set the time of the next restart signal. We refer to this interface by `set_SEI_trigger_time`. The only difference of RoT with a regular timer is that it allows the processor to call the `set_SEI_trigger_time` interface only *once* after each restart and ignores any additional calls to this interface until the timer expires. Once the RoT timer is configured, adversaries cannot disable it until it has expired and the platform is restarted. Figure 6.1 illustrates the sequence of events in the system.

6.2.2 Finding the Safety Window in Run-Time

During the SEI, platform executes two tasks in parallel: *(i)* `find_safety_window` task which calculates the time window in which the plant will remain safe due to its physical inertia and uses this result to set the triggering time of the next SEI. And, *(ii)* SC that keeps the plant stable while `find_safety_window` is computing. Figure 6.1 presents an example sequence of the system events. If no malicious activity had taken place during the previous execution cycle (first cycle of Figure 6.1), the next SEI triggering time is computed and scheduled quickly, and the mission controller resumes. However, if an attacker had been able to compromise the platform within the previous cycle and managed to push the plant close to the inadmissible states (second cycle of Figure 6.1), the SC will need some time to stabilize the plant – push it further into the recoverable region – and SEI will be longer.

The fundamental idea here is how should `find_safety_window` calculate the triggering time of the next SEI such that up to the beginning of the next SEI, the physical plant would not be able to reach an unsafe state and at the beginning of next SEI, the state would still be *recoverable* by the SC. The rest of this subsection answers this question.

Before we proceed, it is useful to define some notations. We use the notation of $\mathsf{Reach}_{=T}(x, C)$ to denote the set of states that are reachable by the physical plant from an initial set of states x after exactly T units of time have elapsed under the controller C.

$\mathsf{Reach}_{\leq T}(x, C)$ can be defined as $\bigcup_{t=0}^{T} \mathsf{Reach}_{=t}(x, C)$ *i.e.,* union of all the states reachable within all times t up to T time units. Also, we use SC to refer to the *safety controller* and UC to refer to an *untrusted controller*, *i.e.,* one that might have been compromised by an adversary. We use notation $\Delta(x_1, x_2)$ to represent the shortest time required for the physical plant to reach state x_2, starting from x_1.

Definition 6.1. *True Recoverable states are all the states from which the given SC can eventually stabilize the plant. Formally,* $\mathcal{T} = \{x \mid \exists \alpha > 0 : \mathsf{Reach}_{\leq \alpha}(x, SC) \subseteq \mathcal{S}$ *&* $\mathsf{Reach}_{=\alpha}(x, SC) \subseteq \mathcal{R}\}$*. The set of true recoverable states is represented with* $\mathcal{T}$*.*

Definition 6.2. $\mathcal{T}_\alpha$ *denotes the set of states from which the given SC can stabilize the plant within at most* α *time. Formally, we have* $\mathcal{T}_\alpha = \{x \mid \mathsf{Reach}_{\leq \alpha}(x, SC) \subseteq \mathcal{S}$ *&* $\mathsf{Reach}_{=\alpha}(x, SC) \subseteq \mathcal{R}\}$*. From definition it follows that* $\forall \alpha : \mathcal{T}_\alpha \subseteq \mathcal{T}$*.*

Let us call T_s, the switching time, and use it for referring to the time between the triggering time of the SEI until SEI is active and ready to execute tasks. For the restart-based SEI implementation, T_s is equal to the length of one restart cycle of the embedded platform[4]. Furthermore, let us use γ to represent the shortest time that is possible to take a physical system from its current state $x(t) \in \mathcal{T}$ to a state outside of $\mathcal{T}$. We can write

$$\gamma(x) = \min \{\Delta(x, x') \mid \text{ for all } x' \notin \mathcal{T}\} \tag{6.1}$$

It follows that

$$\text{If } x(t) \in \mathcal{T} \text{ then } x(t + \tau) \in \mathcal{T} \text{ where } \tau < \gamma(x(t)). \tag{6.2}$$

From Equation 6.2 we can conclude

$$\begin{aligned} &\mathsf{Reach}_{\leq \gamma(x(t)) - \epsilon}(x(t), UC) \subseteq \mathcal{S} \\ &\mathsf{Reach}_{= \gamma(x(t)) - \epsilon}(x(t), UC) \subseteq \mathcal{T} \end{aligned} \quad \text{where } \epsilon \to 0 \tag{6.3}$$

Equation 6.3 indicates that if it was possible to calculate $\gamma(x(t))$ in an SEI, we could have scheduled the consecutive SEI to be triggered at time $t + \gamma - T_s - \epsilon$. This process would have ensured that by the time the following SEI had started, the state of the plant was truly recoverable and admissible.

The value of $\gamma(x)$ depends on the dynamics of the plant and the limits of the actuators. Unfortunately, it is not usually possible to compute a closed-form representation for $\gamma(x)$. Because computing a closed-form representation for the $\mathcal{T}$ of the given SC is not a trivial

[4] T_s is the length of the interval from the triggering point of restart until the reboot is completed, filters are initialized and control application is ready to control the plant.

problem. Actuator limits is another factor that needs to be taken into account in the calculation of $\mathcal{T}$. Therefore, in many cases, finding γ would require performing extensive simulations or solving numerical or differential equations.

An alternative approach is to check the conditions of Equation 6.3 for a specific value of time, λ:

$$\mathsf{Reach}_{\leq\lambda}(x(t), UC) \subseteq \mathcal{S} \;\&\; \mathsf{Reach}_{=\lambda}(x(t), UC) \subseteq \mathcal{T}_\alpha \tag{6.4}$$

Fortunately, having a tool to compute the reachable set of states in run-time allows us to evaluate all the components of Equation (6.4). Real-time reachability can compute the reachable set of states up to the λ time with an untrusted controller UC to check the first part of the equation (6.4). To evaluate the second part, we use the calculated reachable set at time λ as the starting set of states to perform another reachability computation for α time under SC and check $\mathsf{Reach}_{\leq\alpha}(\mathsf{Reach}_{=\lambda}(x(t), UC), SC) \subseteq \mathcal{S}$ and $\mathsf{Reach}_{=\alpha}(\mathsf{Reach}_{=\lambda}(x(t), UC), SC) \subseteq \mathcal{R}$. These two conditions are equivalent to the second part of the equation above.

The λ that is calculated for the state $x(t)$ is a *safety window* of the physical system in state $x(t)$, that is the interval of time, starting from time t, that the plant will remain safe and recoverable, even if the adversary controls it. Hence, we can conclude that the time $t + \lambda - T_s$, is a point where the platform can be safely restarted – *i.e.,* the next SEI can be triggered. Algorithm 6.1, performs a binary search and tries to find the largest safety window of the plant from a given $x(t)$ within a bounded computation time, T_{search}. Given a large T_{search}, Algorithm 6.1 would calculate the the *maximum safety window* of the plant for that state. In run-time, however, T_{search} has to be limited and therefore choosing the initial candidate $\lambda_{\text{candidate}}$ is crucial. It is also possible to use an adaptive λ_{init} by dividing the state space into subregions and assigning a λ_{init} to each region. At runtime, choose the λ_{init} associated with the state and initialize the Algorithm 6.1.

Note that the real actions of the adversary are unknown ahead of the time. As a result, in the conditions of Equation (6.4), the reachability of the plant under *all* possible control values need to be calculated. Consequently, the computed reachable set under UC ($\mathsf{Reach}(x, UC)$) is the largest set of states that might be reached from the given initial state, *within the specified time*. The real-time reachability tool in [30] allows this sort of computation due to the usage of a box representation for control inputs. Control inputs are set to the full range available to the actuators. As a result, the computed set the states that might be achieved under all of the actuator values. Notice that this procedure does not impact the time required for reachability computation.

Algorithm 6.1: Finding physical safety window from state x. Here, $T_{\text{eq-6.4}}$ refers to the time required to evaluate the conditions of Equation 6.4. We can compute the exact value of $T_{\text{eq-6.4}}$ because the reachability computation time is capped (one of the important features of [30]) and, in total, there are 4 Reach operations to be performed.

`find_safety_window`(x, λ_{init})

1: `startTime` := currentTime()
2: $\lambda_{\text{candidate}} := \lambda_{\text{init}}$
3: `RangeStart` := T_s; `RangeEnd` := $\lambda_{\text{candidate}}$
4: **while** currentTime() - `startTime` $< T_{\text{search}} - T_{\text{eq-6.4}}$ **do**
5: **if** conditions of Equation (6.4) are true for $\lambda_{\text{candidate}}$ **then**
6: $\lambda_{\text{safe}} := \lambda_{\text{candidate}}$
7: `RangeStart` := λ_{safe}; `RangeEnd` := $2\lambda_{\text{safe}}$
8: **else**
9: `RangeEnd` := $\lambda_{\text{candidate}}$
10: **end if**
11: $\lambda_{\text{candidate}} := (\texttt{RangeStart} + \texttt{RangeEnd})/2$
12: **end while**
13: **return** -1

When an intelligent adversary compromises the system, it can quickly push the plant towards the inadmissible states and very close to the boundary of the unsafe region. When operating close to the inadmissible states, there is a very narrow margin for misbehavior. If the adversary takes over again, they can violate the physical safety. Therefore, when SEI starts and the plant is in states very close to the boundary of the unsafe region, safety controller would need to execute for longer than usual until the plant is sufficiently pushed into the safe area. Deciding on how long the SC needs to run automatically happens based on the result of `find_safety_window` as presented in Algorithm 6.2. If the plant's state is too close to the boundary of the unrecoverable region, the safety window of the plant will be very short, and `find_safety_window` will most likely return -1. In Algorithm 6.2, this will force the while loop and consequently the SC to continue running for another cycle. This cycle will continue until SC has sufficiently distanced the plant from the unsafe region. At this point, `find_safety_window` will be able to compute a safety window and the SEI will end.

It's worth noting that what real-time reachability yields is a superset of the actual reachable set of states. Therefore, the calculated λ ensures that the system always remains within the safe region.

Algorithm 6.2: One operation cycle with restart-based SEI

1: Start Safety Controller. */* SEI begins */*
2: $\lambda_{\text{safe}} = \lambda_{\text{init}}$ /*Initializing the safety window*/
3: **repeat**
4: `start_time` := systemTime()
5: x := obtain the most recent state of the system from Sensors
6: λ_{safe} :=`find_safety_window`(x, λ_{safe})
7: `elapsed_time` := systemTime() - `start_time`
8: **until** $\lambda_{\text{safe}} \neq -1$ and $\lambda_{\text{safe}} > T_s +$ `elapsed_time`
9: Send $\lambda_{\text{safe}} -$ `elapsed_time` $- T_s$ to RoT. */* Set the next restart time. */*
10: Activate external interfaces. */* SEI ends. */*
11: Terminate SC and launch the mission controller.
12: When RoT sends the restart signal to hardware restart pin:
13: Restart the platform
14: Repeats the procedure from beginning (from Line 1)

6.3 TEE-ASSISTED DESIGN IMPLEMENTATION

The restart-based approach to enable SEIs requires a restart in each operation cycle and imposes two main types of overheads on the system: *(i)* restart-time and *(ii)* memory erasure due to the restarts. Implementing this approach on some CPSs can be challenging especially if the platform restart time is not negligible compared to the speed of the dynamics of the plant. Another issue with this design implementation arises from the fact that the system restarts erase the platform memory. For some applications, such frequent memory erasures can be problematic. For instance, to establish a remote connection, the controller might need to perform handshaking steps and store the state in the memory. If the system is frequently restarted, the controller may not be able to establish a reliable communication.

To mitigate some of these issues, we propose an alternative implementation where we use ARM TrustZone technology [5] and in particular LTZVisor [84] – which is a lightweight TrustZone assisted hypervisor with real-time features for embedded systems[5]. Here, instead of relying on the platform restarts to create SEIs, we exploit the isolated execution environments that are attainable through TrustZone.

In the rest of this section, we present some background on TrustZone and LTZVisor, and then we discuss the implementation of the approach.

[5]In this work, we have used TrustZone and LTZVisor. Nevertheless, other available Trusted Execution Environment (TEE) technologies such as Intel's Trusted Execution Technology (TXT) [6] can be employed to achieve the same goal.

6.3.1 Background on TrustZone and LTZVisor

TrustZone [5] hardware architecture can be seen as a dual-virtual system, partitioning all system's physical resources into two isolated execution environments. A new 33rd processor bit, the Non-Secure (NS) bit, indicates in which world the processor is currently executing, and is propagated over the memory and peripherals buses. An additional processor mode, the monitor mode, is added to store the processor state during the world switch. TrustZone security is extended to the memory infrastructure through the TrustZone Address Space Controller (TZASC) that can partition the DRAM into different memory regions. Secure world applications can access non-secure world memory, but the reverse is not possible. Additional enhancements in TrustZone provide the same level of isolation in cache and system devices.

LTZVisor [84] is a lightweight hypervisor that allows the consolidation of two virtual machines (VMs), running each of them in an independent virtual world (secure and non-secure). It exploits TrustZone features in the platforms to provide memory segmentation, cache-level isolation, and device partitioning between the two VMs. LTZVisor dedicates timers to each VM that enables each one to have a distinctive notion of system time. Additionally, it provides an API for communication between the two VMs.

LTZVisor manages the secure and non-secure world interrupts in a way that meets the requirements of the hard real-time systems. All the implemented interrupts can be individually defined as secure and non-secure. If the secure VM is executing, all the secure interrupts are redirected to it without hypervisor interference. If a non-secure interrupt arises during secure VM execution, it will be queued and processed as soon as non-secure side becomes active. On the other hand, if the non-secure VM is executing and a secure interrupt arises, it will be immediately handled in the secure world. This design prevents a denial-of-service attack on the secure-side applications.

LTZVisor implements a scheduling policy that guarantees that the non-secure guest OS is only scheduled during the idle periods of the secure guest OS, and the secure guest OS can preempt the execution of the non-secure one. This scheduling policy resolves one of the well-known real-time scheduling problems in virtual environments known as hierarchical scheduling and makes LTZVisor an excellent choice to meet real-time requirements of the tasks in the secure VM. Besides, creators of LTZVisor show that the overhead of switching from secure VM to non-secure VM and vice versa is small and deterministic [84]. Thus, secure VM is ideal for running a real-time operating system (RTOS) whereas, non-secure VM can run general purpose operating systems like Linux.

6.3.2 TEE-enabled SEIs

In this design, to protect the SC and `find_safety_window` tasks, they execute in the secure VM, and everything else runs in the non-secure VM. The SC and `find_safety_window` are executed, and before they finish, they schedule their next execution time *i.e.,* the next SEI. Mission controller and any other component start running as soon as all the tasks in the secure VM have yielded. LTZVisor guarantees that the non-secure VM cannot interfere with the execution of the tasks in the secure VM.

Each task inside the secure VM, once executed, can choose to yield and set the future time when its status will change to ready again. In LTZVisor, the secure VM has a higher priority than the non-secure VM. Consequently, the non-secure VM tasks will execute only when there are no secure tasks that are ready to execute. Similarly, as soon as one of the secure VM tasks becomes ready, LTZVisor pauses the non-secure VM, stores the necessary registers and executes the secure task. The scheduling policy in each VM determines the priorities and execution details for the tasks of that VM.

The operation cycle of the system during the SEI is very much the same as described in Algorithm 6.2 except instead of setting the RoT and the restarting step, secure tasks schedule their next wake up time using the secure platform timer or the OS of the secure VM. SC and `find_safety_window` tasks execute in parallel. As soon as `find_safety_window` finds a valid safety window, both tasks set their next wake up time and yield the execution. At this point, LTZVisor resumes the execution of the non-secure VM until it is time for the SC and `find_safety_window` to wake up.

Note that, due to the isolation provided by TrustZone, non-secure VM cannot interfere with the execution of secure tasks when they are ready to execute. This protection eliminates the need for the RoT timer which was a necessary component to implement the restart-based SEI.

6.3.3 Optional Recovery Restart

The safety guarantees that the TEE-based implementation provides are precisely the same guarantees as restart-based SEI implementation. Nevertheless, there is a significant difference. When the system is being restarted in every cycle, if it gets compromised, the malicious components will only last until the following restart, and then the software will be restored. When using TrustZone, if the non-secure world gets compromised, it will remain compromised. Although the adversary cannot violate the safety of the plant, it can seriously prevent the system from making any progress.

There are two possible mechanisms to mitigate this problem. One arrangement is to introduce rare, randomized restarts into the system[6]. Another mitigation is to monitor the platform, during the SEI, for potential intrusions and malicious activities and restart the platform after the malicious behavior is detected[7]. Note that with the optional recovery restarts described in this section, a well-behaving system that is not under attack will rarely restart. The platform will be restarted only after it is deemed malicious or when the random function requires it to do so. Whereas, with the restart-based implementation of SEIs, the platform has to be restarted before every SEI.

Deciding whether the platform needs to restart or not takes place at the beginning of the SEI – either based on a randomized policy or a detection mechanism. If it is decided to restart, the steps to perform the recovery are presented in Algorithm 6.3. One crucial point in restarting the system is the fact that the platform restart must take place only when the plant is in a state where it will sustain the safety throughout the restart and will end up in a recoverable state – according to Defintion 3.2 – after the restart has completed. This requirement is satisfied if the conditions of Equation 6.4 are met.

Under these steps, SC continues to push the plant towards the center of the safe region. In parallel, the `find_safety_window` function is executed in a loop and checks if the plant at its current state meets the conditions of safe restarting in Equation 6.4 for the length of platform restart time. Once the `find_safety_window` confirms the safety conditions for the current plant state, the recovery restart is initiated. In other words, the system is restarted when the plant has enough distance from the boundaries of the recoverable states and unrecoverable states.

Algorithm 6.3: Steps to perform a recovery restart.

1: SC starts and is periodically invoked in parallel to the next steps.
2: $\lambda_{\text{Recovery}} = T_{\text{restart}} + T_{\text{eq-6.4}} + \epsilon$
3: **repeat**
4: $x :=$ obtain the most recent state of the system from Sensors
5: **until** conditions of Equation (6.4) are true for $\lambda_{\text{Recovery}}$
6: (optional) Store sensor reading in the non-volatile storage
7: Restart the system
8: /*Following steps are executed after the restart*/
9: (optional) Load the pre-restart sensor data from storage into the memory

[6]The rationale behind randomized system restarts – also known in the literature as software rejuvenation – is that there are no perfect intrusion detection mechanisms. Also, there will always exist malicious activities that will remain undetected. In another work [34], we have analyzed the impact of restart-based recovery on the availability of a system under attack.

[7]In this work, we do not propose any particular intrusion detection algorithm. There is a variety of such techniques that the system architects can choose from.

6.3.4 Carrying Sensor State Between Restarts

Some control applications might need the prior-to-restart sensor readings for improved performance or higher quality output. For instance, low-pass filters use the past sensor readings to remove noise from the sensors. TEE-assisted implementation can accommodate this requirement. In this design, restarts are always initiated within the secure VM and, the secure VM is always the first to execute after the restart. Immediately prior to the restart, the secure VM can store any data on the non-volatile storage, and load it back into the memory after the restart. Note that the non-secure VM is not able to interfere with this process at all.

It is worth mentioning that the above procedure can be used to carry any values, including the variables or states in the non-secure VM, and make them available after the restart. However, we strongly advise avoiding a design where the CPS relies on the prior-to-restart state of the non-secure VM to carry out its essential mission mainly because the platform is restarted only when the non-secure VM is deemed compromised. At this point, all the states in the non-secure VM must be assumed corrupted. Passing the corrupted values across restarts can propagate the adversarial effect across the restarts and defeat the purpose of recovery restarts.

6.4 EVALUATION AND FEASIBILITY STUDY

In this section, we evaluate the protections provided by our approach and measure the feasibility of implementing it on real-world CPSs. We choose two physical plants for this study: a 3-degree of freedom helicopter [85] and a warehouse temperature management system [86]. For both plants, the controller is implemented using both restart-based and TEE-assisted approaches on a ZedBoard [87] embedded system.

6.4.1 Test-Bed Description

Warehouse Temperature Management System:

This system consists of a warehouse room with a direct conditioner (heater and cooler) to the room and another conditioner in the floor [86]. The safety goal for this plant is to keep the room temperature, T_R, within the range of $[20°C, 30°C]$. Following equations describe the heat transfer between the heater and the floor, the floor and the room, and the room and outside space. The model assumes constant mass and volume of air and heat transfer

only through conduction.

$$\dot{T}_F = -\frac{U_{F/R}A_{F/R}}{m_F Cp_F}(T_F - T_R) + \frac{u_{H/F}}{m_F Cp_F}$$

$$\dot{T}_R = -\frac{U_{R/O}A_{R/O}}{m_R Cp_R}(T_R - T_O) + \frac{U_{F/R}A_{F/R}}{m_R Cp_R}(T_F - T_R) + \frac{u_{H/R}}{m_R Cp_R}$$

Here, T_F, T_R, and T_O are the temperature of the floor, room and outside. m_F and m_R are the mass of floor and the air in the room. $u_{H/F}$ is the heat transferred from the floor heater to the floor and $u_{H/R}$ is the heat transferred from the room heater to the room both of which are controlled by the controller. Cp_F and Cp_R are the specific heat capacity of floor (in this case concrete) and air. $U_{F/R}$ and $U_{R/O}$ represent the overall heat transfer coefficient between the floor and room, and room and outside.

For this experiment, the walls are assumed to consist of three layers; the inner and outer walls are made of oak and isolated with rock wool in the middle. The floor is assumed to be quadratic and consists of wood and concrete. The parameters used are as following[8]: $U_{R/O} = 539.61$ J/hm^{2}K, $U_{F/R} = 49920$ J/hm^{2}K, $m_R = 69.96$ kg, $m_F = 6000$ kg, floor area $A_{F/R} = 25$ m^2, wall and ceiling area $A_{R/O} = 48$ m^2, thickness of rock wool, oak and concrete in the wall and floor respectively 0.25 m, 0.15 m and 0.1 m. Maximum heat generation capacity of the room and floor conditioner is respectively 800 J/s and 115 J/s. And, the maximum cooling capacity of the room and the floor cooler is -800 J/s and -115 J/s.

3-Degree of Freedom Helicopter:

3DOF helicopter (displayed in figure 6.2) is a simplified helicopter model, ideally suited to test intermediate to advanced control concepts and theories relevant to real-world applications of flight dynamics and control in tandem rotor helicopters, or any device with similar dynamics [85]. It is equipped with two motors that can generate force in the upward and downward direction, according to the given actuation voltage. It also has three sensors to measure elevation, pitch, and travel angle as shown in Figure 6.2. We use the linear model of this system obtained from the manufacturer manual [85] for constructing the safety controller and calculating the reachable set in run-time. Due to the lack of space, the details of the model are included in our technical report [88].

For the 3DOF helicopter, the safety region is defined in such a way that the helicopter fans do not hit the surface underneath, as shown in Figure 6.2. The linear inequalities describing

[8]For the details of calculation of $U_{F/R}$ and $U_{R/O}$ and the values of the parameters refer to Chapter 2 and 3 of [86].

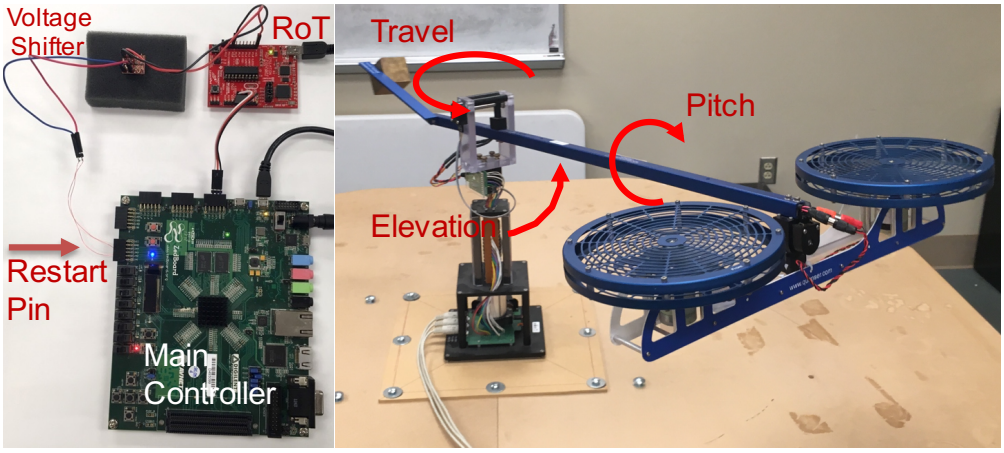


Figure 6.2: 3DOF helicopter and the ZedBoard controller.

the safety region are $-\epsilon + |\rho|/3 \leq 0.3$, $\epsilon \leq 0.4$, and $|\rho| \leq \pi/4$. Here, variables ϵ, ρ, and λ are the elevation, pitch, and travel angles of the helicopter. Limitations on the motor voltages of the helicopter are $|v_l| \leq 4V$ and $|v_r| \leq 4V$ where v_l and v_r are the voltage for controlling left and right motors.

6.4.2 Restart-Based Implementation of SEI

In this section, we discuss the implementation of the controllers of the 3DOF platform and the temperature management system using the restart-based SEI approach (Section 6.2). In our technical report [88], more details are provided about the hardware and software implementation of the controller. Due to the limited access to a real warehouse, the controller interacts with a simulated model of the physical plant running on a PC (Hardware-in-the-loop simulation).

RoT Module:

The RoT module is implemented using a low-cost MSP430G2452 micro-controller on a MSP-EXP430G2 LaunchPad board [89]. To enable restarting, pin P2.0 of the micro-controller is connected to the restart input of the main controller. Internal Timer A of the micro-controller is used for implementing the restart timer. It is a 16-bit timer configured to run at a clock rate of 1 MHz (*i.e.,* 1μs per timer count) using the internal, digitally controlled, oscillator. A counter inside the interrupt handler of Timer A is used to extend the timer with an adjustment factor, in order to enable the restart timer to count up to the

required range based on the application's needs.

The I^2C interface is adopted for the main controller to set the restart time on the RoT module. After each restart, during the SEI, the RoT acts as an I^2C slave waiting for the value of the restart time. As soon as the main controller sends the restart time, RoT disables the I^2C interface and activates the internal timer. Upon expiration of the timer, an active signal is set on the restart pin to trigger the restart event and the I^2C interface is activated again for accepting the next restart time.

Main Controller:

The controller is implemented on a Zedboard [87] which is a development board for Xilinx's Zynq-7000 series all programmable SoC. It contains an XC7Z020 SoC, 512 MB DDR3 memory, and an onboard 256 MB QSPI Flash. The XC7Z020 SoC consists of a processing system (PS) with dual ARM Cortex-A9 cores and 7-series programmable logic (PL). The processing system runs at 667MHz. In our experiments, only one of the ARM cores is used, and the idle cores are not activated. The I^2C and *UART* interfaces are used for connecting to the RoT module and the actuators of the plant. Specifically, two multiplexed I/Os, MIO14 and MIO15, are configured as SCL and SDA for I^2C respectively. We use UART1 (MIO48 and MIO49 for UART TX and RX) as the main UART interface.

The reset pin of Zedboard is connected to RoT module's reset output pin via a BSS138 chip, an N-channel voltage shifter. It is because the output pin on RoT module operates at 3.3 volts while the reset pin on Zedboard accepts 1.8 volts. The entire system (both PS and PL) on Zedboard is restarted when the reset pin is pulled to the low state. The boot process starts when the reset pin is released (returning to the high state). A boot-loader is first loaded from the onboard QSPI Flash. The image for PL is then loaded by the boot-loader to program the PL which is necessary for PS to operate correctly. Once PL is ready, the image for PS is loaded, and the operating system will take over the control of the system.

The platform runs *FreeRTOS* [73], a preemptive real-time operating system. Immediately after the reboot, `safety_controller` and `find_safety_window` tasks are created and executed. `safety_controller` is a periodic task with the period of 20 ms (50 Hz) and the execution time of 100 μs and has the highest priority in the system. Safety controller itself is designed using the method described in Section 3.1. Each invocation of this tasks obtains the values of sensors and sends the control commands to the actuators. `find_safety_window` executes a loop and only breaks out when a valid safety window is calculated. It executes at all times except when it is preempted by `safety_controller`. When `find_safety_window` computes a valid safety window, it sends the value minus the

elapsed time (Algorithm 6.2) to the RoT module via the I^2C interface, sets a global variable in the system, and terminates. Based on this global variable, `safety_controller` task terminates, and the mission controller task is launched. `find_safety_window` is implemented based on the Pseudo-code described in Algorithm 6.1. Execution time of each cycle of the loop in this function is capped at 50 ms (*i.e.,* $T_{\text{search}} := 50$ ms). In `find_safety_window`, to calculate the reachability of the plant from a given state, we used the implementation of our real-time reachability tool [30]. All the code for the implementation can be found in the GitHub repository [88].

3DOF Helicopter Controller: ZedBoard platform interfaces with the 3DOF helicopter through a PCIe-based Q8 data acquisition unit[75] and an intermediate Linux-based machine. The PC communicates with the Zedboard through the UART interface. Mission controller is a PID controller whose goal is to navigate the 3DOF to follow a sequence of set points. Control task has a period of 20 ms (50 Hz), and at every control cycle, the control task receives the sensor readings (elevation, pitch, and travel angles) from PC and sends the next set of voltage control commands for the motors. The PC uses a custom Linux driver to communicate with the 3DOF sensors and motors. In our implementation, the restart time of the ZedBoard with FreeRTOS is upper-bounded at $390ms$.

Warehouse Temperature Controller: Due to the lack of access to the real warehouse, we used a hardware-in-the-loop approach to perform the experiments related to this plant. Here, the PC simulates the temperature based on the heat transfer model Described in Section (6.4.1). The mission controller is a PID that adjusts the environment temperature according to the time of the day. The controller is implemented on the ZedBoard with the same components and configurations as the 3DOF controller – RoT, serial port connection, I^2C interface, $50Hz$ frequency, and the same restart time. Control commands are sent to the PC, applied to the simulated plant model and the state is reported back to the platform.

6.4.3 TrustZone-Assisted SEI implementation

Our prototype implementation uses LTZVisor on the ZedBoard which provides two isolated execution environments, secure VM and non-secure VM. LTZVisor can only use one of the ZedBoard cores, and the other cores are not activated. Similar to the previous section, ZedBoard is connected to the physical plant sensors and actuators through *UART* interface. The configuration of the *UART* pins and PL are the same as the previous section.

`safety_controller` and `find_safety_window` are compiled as one bare metal application and executed in the secure VM[9]. The functionality of these components is identical to

[9]LTZVisor also provides support for FreeRTOS on the secure VM and Linux on the non-secure VM.

what was described in the previous section. Using the platform timer, we ensure that the `safety_controller` function is called and executed every 20 ms while, `find_safety_window` is being executed for the rest of the time. Once the state of the plant reaches a state where a safety window is available, `find_safety_window` returns the results, the application yields the processor and sets the next invocation point to the current time plus computed safety window minus the computation time – Section 6.3.2. At this point, LTZVisor restores the execution of the mission controller application in the non-secure VM until the secure VM application is invoked again. We use the `YIELD` function, provided by the LTZVisor on the secure VM, which suspends the execution of the application and invokes it after the specified interval of time.

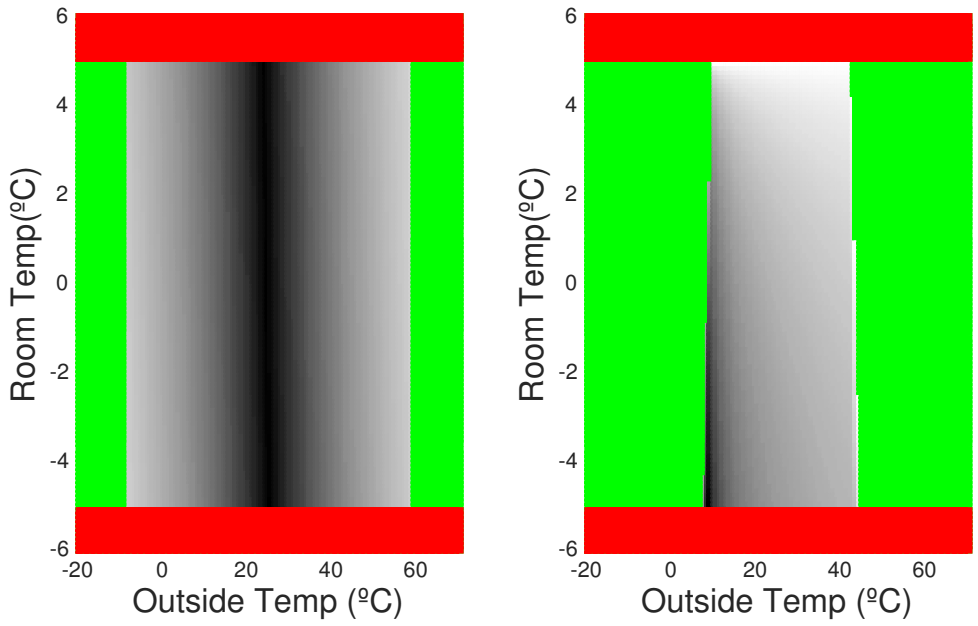
In our prototype, recovery restarts are initiated based on a randomized scheme. We use a pseudo-random number generator function that returns a value between 0 and 1. if the values are less than 1/1000, we restart the platform – the probability of 0.1 percent. Otherwise, the execution proceeds to the normal SEI. The mechanism to trigger the restarts is through system-level watchdog timer. This is an internal 24-bit counter that on timeout outputs a system reset to the Processing System (all the cores and system registers) and Program Logic (the FPGA fabric in the ZedBoard). To trigger a restart, the timer is enabled and set to expire on the shortest time allowed by the resolution. The timer expires immediately and restarts the platform.

6.4.4 Safety Window of the Physical Plants

At the end of each SEI, the triggering point of the next SEI needs to be computed and scheduled. Two main factors determine the distance between consecutive SEIs; *(i)* how stable the dynamics of the plant is and *(ii)* the proximity of the current state of the plant to the boundaries of the inadmissible states. In figures 6.3 and 6.4, the absolute maximum safety window of the physical plant is plotted from various states for the plants under consideration. These values are computed using Algorithm 6.1 except for clarification, the lower end of the search in this algorithm, `RangeStart`, is set to 0. In these plots, the red region represents the inadmissible states, and the plant must never reach those states. If the plant is in a state that is marked green, it is still undamaged. However, at some future time, it will reach an inadmissible state, and the safety controller may not be able to prevent it from coming to harm. The reason is that actuators have a limited physical range. In the green states, even actuators operating with the maximum capacity, may not be able to cancel the momentum

However, at the time of this writing, the code enabling these features is not publicly released yet. That is why these components are implemented as bare-metal applications.

and prevent the plant from reaching unsafe states. The gray and yellow highlighted regions are the operational region of the plant – states where the safety window of the plant is larger than zero and mission controller can execute. In the gray area, the darkness of the color is the indicator of the length of the safety window in that state. Darker points indicate a larger value for the safety window.

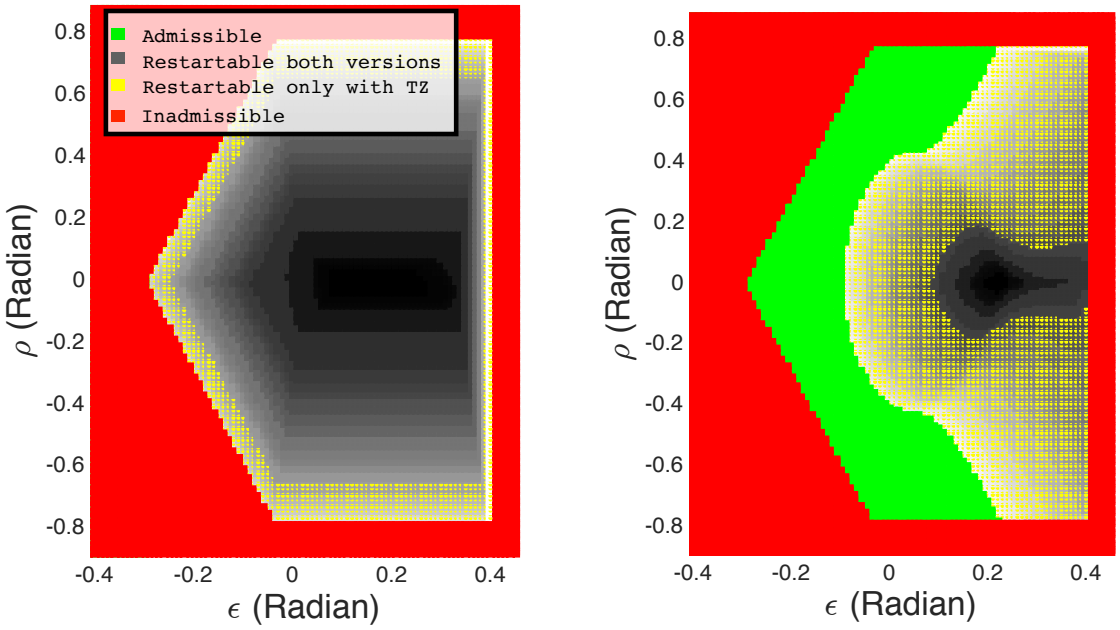


(a) Projection into $T_F = 25^oC$ (b) Projection into $T_F = 29^oC$

Figure 6.3: Safety window values for the warehouse temperature. Largest value of the safety window – the darkest region – is $6235s$.

Figures 6.3(a) and 6.3(a), plot the calculated safety windows for the warehouse temperature management system. For this system, when the outside temperature is too high or too low, the attacker requires less time to take the temperature beyond or below the safety range. Note that if an adversary take over the platform at $T_F = 25^oC$, $T_R = 40^oC$, and $T_O = 26^oC$ – top part of Figures 6.3(a) – and runs the heaters at their maximum capacity, plant will remain safe for $6235s$. Intuitively, due to high conductivity between the floor and the room as well the high heat capacity of the floor, the rate of heat transfer from room to the floor is larger than the transfer rate from the heater to the room. Due to the same reason, when the floor temperature is $T_F = 29^oC$, the safety window of the plant is almost zero near the boundary of the $T_R = 40^oC$ – top part of Figure 6.3(a).

In Figure 6.4, the safety window for the 3DOF helicopter are plotted – projection into the 2D plane. The darkest point, have the largest safety window which is $1.23s$. As seen



(a) Projection of the state space into the plane $\dot{\epsilon} = 0$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3\text{Radian}/s$

(b) Projection of the state space into the plane $\dot{\epsilon} = -0.3\text{Radian}/s$, $\dot{\rho} = 0$, $\lambda = 0$, and $\dot{\lambda} = 0.3\text{Radian}/s$

Figure 6.4: Safety window values for the 3DOF helicopter. Largest value of the safety window – the darkest point – is $1.23s$.

in this figure, safety window is largest in the center where it is farthest away from the boundaries of the unsafe states. In Figure 6.4(b), the angular velocity of 3DOF elevation is $\dot{\epsilon} = -0.3\text{Radian}/s$ which means that the helicopter is heading towards the bottom surface at a rate of 0.3 Radian per second. As seen in the figure, with this downward velocity, the plant cannot be stabilized from the lower elevation levels (*i.e.,* the green region). It can also be seen that in the states with elevation less than 0.1 Radians, the safety window is shorter in Figure 6.4(b) compared to Figure 6.4(a). Intuitively, for the adversary, crashing the 3DOF helicopter is easier when the plant is already heading downward.

As we mentioned earlier, the temperature management system has higher inertia and slower dynamics than the 3DOF helicopter. The above figures reflect this effect, very clearly. As the computed safety windows for the former plant are orders of magnitudes larger than the latter – 6235 s is the largest safety window for warehouse temperature versus 1.23 s for the 3DOF helicopter. In this system, the rate of the change of the temperature even when the heater/coolers run at their maximum capacity is slow, and adversary needs more time to force the state into unsafe states.

Now, we will discuss the difference between the gray and yellow regions. The mission

controller can operate in the yellow states only with the TEE-assisted implementation of the SEIs and not with the restart-based implementation of the SEIs. This is due to the following reason. In run-time, computed safety windows are used to set the triggering point of the next platform SEI. However, the next SEI can be scheduled only if the safety window is larger than the switching time of the platform, T_s, as presented in Algorithm 6.2. With the restart-based implementation of the SEIs, the switching time is equal to the restart time of the platform (390 ms for RTOS on the ZedBoard) whereas, with the TEE-assisted implementation, switching time is the timing overhead of the context switching from secure VM to non-secure VMs and vice versa (less than 12 μs for ZedBoard at 667 MHz as presented in [84]). States marked with the yellow color are those that the computed safety window is shorter than the platform restart time. At these states, with the restart-based SEI, the mission controller cannot be activated.

As a result of using TrustZone-assisted implementation, we measured a 234 percent increase in the size of the operational region of the 3DOF plant – the yellow vs. the gray area – across the 6-dimensional state space. However, note that this measurement is very specific to this particular platform and this specific plant. The expected improvement highly depends on the platform restart time and the speed of the plant dynamics. Not every CPS can be expected to gain significant benefits from adopting TrustZone for implementing the SEIs. For instance, if the restart time of the platform were shorter, the size of the gray area in Figure 6.4 would have been larger, and the overall improvement of the operable states – as a result of using TrustZone – would have been smaller. Comparison between the size of the yellow region for the 3DOF vs. the temperature management system is another clear implication of this point. The platform restart time compared to the length of the safety windows of the warehouse plant is almost negligible. That is why implementing the SEIs using TrustZone does not yield any noticeable improvements and the yellow region in Figure 6.3 is non-visible.

6.4.5 Impact on Controller Availability

Every CPS has a mission that is the primary goal of the system to accomplish. The main component that drives the system towards this goal is the mission controller. Therefore, every process that interrupts the execution of the mission controller results in the slow progress of the CPS mission. Thus, one of the consequences of our design is that the SEIs and the platform restarts stop the execution of the mission controller and reduce its availability. In this section, we measure the impact of each one of the two implementations of our design, on the average availability of the mission controller.

The exact "availability" of the mission controller is the ratio of time that the mission controller is executing (all the times that the system is not in the SEI and is not going through a restart) to the total time of the operation. In every restart cycle, availability is defined as $\delta_{\text{mc}}/(\delta_{\text{mc}}+T_{\text{SEI}}+T_s)$. Here, δ_{mc} is the duration of mission controller execution, T_{SEI} is the length of SEI, and T_s is the switching time. With the restart-based implementation of the SEIs, T_s is equal to the restart time of the platform, whereas, for the TrustZone-assisted SEI implementation, T_s is the upper bound of the time required for switching from non-secure VM to secure VM and vice versa. The exact availability of the mission controller is specific to the particular trajectory that the plant takes. To get a better sense of this metric, for each implementation, we compute the average availability of the mission controller across all the states where the safety window is longer than the switching time, T_s, which is 390 ms for restart-based SEI and 12 μs for the TrustZone-assisted SEI implementation.

For the 3DOF system, with the restart-based implementation, the calculated average availability of the mission controller is %51.2. As seen in the Figure 6.4, safety windows of the 3DOF plant are in the range of 0 s to 1.23 s. The platform has a restart time of 390 ms which is significant relative to the values of safety windows and it notably reduces the availability of the mission controller. On the other hand, with the TrustZone-assisted SEIs, the average availability of the mission controller is %85.1. When TrustZone is utilized, T_s is negligible – $12\mu s$ which explains the %35 improvement in the availability. It can be seen that despite the negligible switching switching overhead, the mission controller does not reach %100 availability. This is because of the time required to evaluate the safety conditions and execute `find_safety_window` in the loop inside Algorithm 6.2. In the states near the unsafe/safe state boundary, the platform might need to execute the loop cycle more than once – longer SEI allows the safety controller to create enough distance from the unrecoverable states.

For the temperature management system, the average availability of the mission controller is %99.9 with both restart-based and TrustZone-assisted implementations of the SEIs. Due to the slow dynamics of this plant, safety windows are much longer than the T_s and T_{SEI} under both implementations – as illustrated in Figure 6.3. Hence, the mission controller is almost always available. Due to the same reason, reduced switching time that is achieved when the controller is implemented using TrustZone instead of the restarts does not notably improve the average availability of the mission controller.

The above results show that the impact of our approach on the temperature management system is negligible under both implementation schemes. In fact, the restart-based implementation is the most suitable choice for this plant and many other high-inertia plants. On the other hand, integrating our design into the controller of the 3DOf helicopter comes

with a considerable impact on the availability of the helicopter controller. Even though the TrustZone considerably reduces the overhead and improves the availability, but still the control performance will noticeably suffer. Note that, the helicopter system is among the most unstable systems and therefore, one of the most challenging ones to provide *guaranteed* protection. As a result, the calculated results for the helicopter system can be considered as an approximate upper bound on the impact of our approach on the controller availability. In the next section, we demonstrate that, despite the reduced availability, the helicopter and warehouse temperature remain safe and the plants make progress. Reduced availability of the controller is the cost to pay to achieve guaranteed safety and can be measured ahead of time by designers to evaluate the trade-offs.

6.4.6 Attacks on the Embedded System

To evaluate the effectiveness of our proposed design, we perform three attacks on the controllers of the 3DOF helicopter (with the actual plant) and one attack on the hardware-in-the-loop implementation of the temperature management system. All the attacks are performed on both versions of the controller implementation. In these experiments, our focus is on the actions of the attacker after the breach into the system has taken place. Hence, the breaching mechanism and exploitation of the vulnerabilities are not a concern of these experiments. An attacker may use any number of exploits to get into the controller device.

In the first experiment, the mission controller of the temperature management system was attacked in the following way. The outside temperature was set to 45° C, and initial room temperature was set to 25° C. Immediately after the SEI was finished, the malicious controller forced both of heaters to increase the temperature with their maximum capacity. Under the restart-based SEI, we observed that the platform was restarted before the temperature reached 30° C and after the restart, SC was able to lower the temperature. Similar behavior was observed with the TrustZone-assisted implementation. A switch to the secure VM was triggered before the temperature reached an unrecoverable value, the SC was able to lower the temperature.

Second attack experiment was performed on the 3DOF helicopter. Here, the attacker, once activated, killed/disabled the mission controller. Under the restart-based SEIs, in every operation cycle, the restart action reloads the software and revives the mission controller. Therefore, the attack was activated at a random time after the end of the SEI in each cycle. Under the TrustZone-assisted SEI implementation, once the mission controller is killed, it

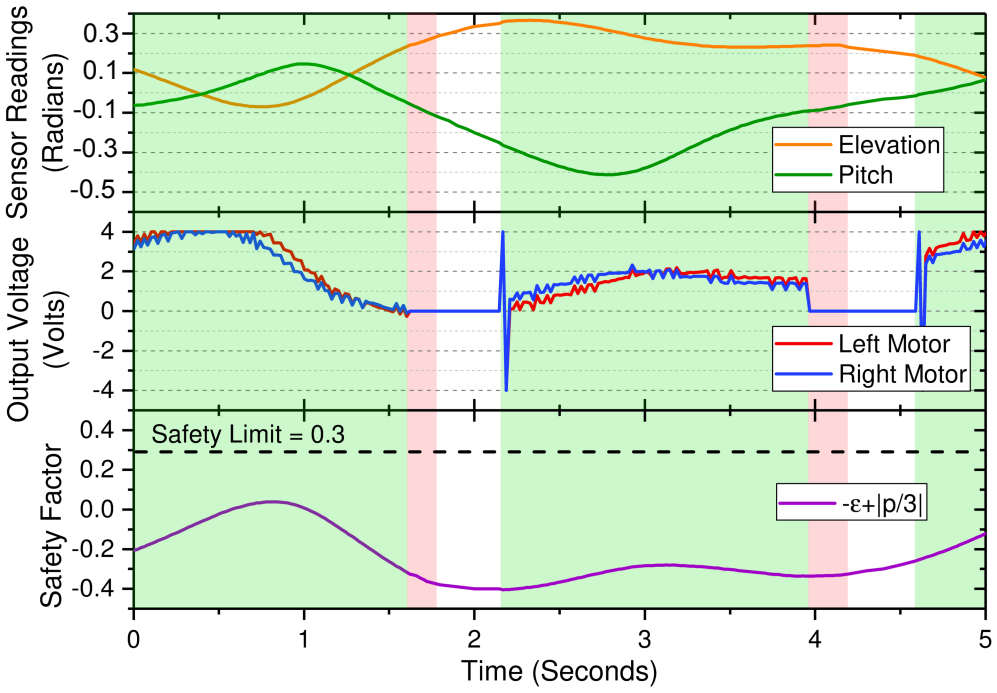


Figure 6.5: 3DOF Helicopter trace under restart-based implementation during two cycles when the system is under worst-case attack (where attacker is active immediately after the SEI). Green: SEI, red: mission controller (in this case attacker), white: system reboot.

will only be recovered when a randomized recovery restart is performed[10]. We used a random value to activate the attack at a random operation cycle – with a probability of 1 percent. After the recovery restart, mission controller was revived and controlled the plant until the next attack was triggered. During these experiments, we observed that the 3DOF helicopter did not hit the surface *i.e.,* it always remained within the admissible set of states.

In the third experiment, the attacker corrupts the sensor readings and feeds the corrupted values in the mission controller logic. To evaluate the safety under an extreme case, the attack is activated immediately after the end of SEI. In both implementations of the controller, the attack is active during all the non-SEI and non-restart times of the system. Similar to the first attack experiment, it was observed that the 3DOF helicopter remained safe throughout the attack.

In the last attack experiment, we investigate the effectiveness of our design against an attacker that is active immediately after the SEI, replaces the original controller with a

[10]Note that in our prototype implementation, we did not implement a detection mechanism. However, one could deploy the logic to monitor the mission controller and restart the platform as soon as the controller is disabled.

malicious process that turns off the motors/fans of the helicopter, and forces the plant to hit the surface. During the operation of the malicious controller, the elevation of the helicopter was reduced. However, in every cycle, before a crash, the safety controller will take over, push the helicopter and increase the elevation. Throughout this experiment, we observed that the plant tolerated the adversarial behavior and did not hit the surface.

A trace of the states of 3DOF helicopter during two consequent restart cycles, with the restart-based implementation of SEIs, is plotted in Figure 6.5. This trace is recorded from the sensor readings of the real physical plant when the plant is under the last attack experiment. The figure depicts elevation, pitch, actuator control inputs (voltages of the motor), and the safety factor. The safety factor is obtained from the safety conditions for the 3DOF as described in Section 6.4.1. From the figure, it can be seen the controller spends most of the time in SEI (green region) and reboot (white region) state. This is because this extreme-case attack is activated immediately after each SEI and destabilizes the helicopter. By the time that the reboot completes (end of the white region), the system is close to unsafe states. Hence, SEI becomes longer so that the SC can stabilize the helicopter. Under this very extreme attack model, the system did not make any progress towards its designated path, yet it remained safe which is the primary goal in this situation.

6.5 SUMMARY AND DISCUSSION

In this chapter, we presented an *attack-tolerant design for embedded control devices* that protects the safety of physical plants in the presence of adversaries. Due to physical inertia, pushing a physical plant from a given (potentially safe) state to an unsafe state – even with complete adversarial control – is not instantaneous and often takes finite (even considerable) time. We leveraged this property to calculate a *safe operational window* and combined it with the effectiveness of *system-wide restarts* (or Trusted Execution Environments such as TrustZone) to protect the safety of the physical system. We evaluated our approach on realistic systems and demonstrate its feasibility.

Some limitations need to be considered before deploying this design for a physical plant or platform. The restart-based implementation is most suitable for CPSs where the platform restart time is much smaller than the speed of the plant dynamics. Many embedded systems have reboot times that range from tens of milliseconds [90] to tens of seconds which can be considered insignificant for many applications such as temperature/humidity management in storage/transportation industries, process control in chemical plants, pressure control in water distribution systems and oxygen level management in patient bodies. The main advantage of the restart-based implementation of SEIs is that it can be deployed on

the cheapest, off-the-shelf micro-controllers that are still widely used in many industrial applications. Also, the deployed application must be designed to operate within the system's safety boundaries. Otherwise the operation of the system is trivially unsafe and the safety controller is unusable.

On the other hand, using the restart-based design on the physical plants with high-speed dynamics will require very frequent restarts and will significantly reduce the control performance and the progress of the system. Frequent reboots may also pose implementation challenges. For instance, the control device may need time to re-establish a connection over the Internet or to authenticate with the ground control. Such actions might not be possible if the device has to restart frequently. These types of applications will significantly benefit from the TrustZone-assisted implementation that eliminates the overhead associated with restarting. As a future direction, we are exploring a multicore implementation of TrustZone-assisted design where the SEI runs in parallel to the mission controller and has minimal impact on the mission controller's performance.

While a restart clears an instance of an attack it does not mean that the adversary is eliminated. It is possible that the adversary attempts to compromise and damage the system after each restart. However, even attack states cannot be carried across multiple attack instances due to the restarts. Each attack instance is contained by the proposed approach since the system restarts before it reaches the unsafe region. As a result, safety of the entire system is guaranteed.

One question that may arise is why not implement all the controllers using TrustZone? Platforms equipped with TrustZone or other TEEs are more expensive. Many control applications are deployed on very low-cost micro-controllers where only restart-based approach is feasible. Furthermore, many high-inertia physical plants will not gain any notable benefit if they are implemented via TrustZone – as shown for temperature management system in the evaluation section. In those cases, the TrustZone-based implementation only unnecessarily complicates the design and implementation of the CPS. Both of these techniques are in line with our goal, as mentioned in the Introduction Chapter, to provide a low-cost and quick to iterate framework that is safe-by-design.

It should be noted that restart-based SEI design is only suitable for stateless controllers (*e.g.,* mission controller) where the control command is generated based on the *current* state of the plant and environment. Such a design is useful for some applications but cannot be utilized with stateful controllers. In fact, this was one of the main incentives to extend our approach with the TEE-based SEI. One question that comes into mind is about the compatibility of a stateful controller with TEE-based SEI implementation and recovery restarts. Note that with TEE-based SEI approach, the system is restarted only when it

is detected to be compromised. Under the assumptions of our threat model, an adversary can maliciously modify all the state on the memory and disk (except read-only storage). In other words, even before the restart, the actual state of the system is already lost and the stateful mission controller cannot continue to operate. Restarting the system at this point only loses the untrustworthy and hence unusable state.

Another important point to mention is that, under both restart-based and TEE-based implementations of SEI, the safety controller has to be a stateless controller so that it can safely stabilize the plant without the knowledge of its past states. This is the main reason that even with the TEE-based SEI design approach, only mission-controller, which is not critical for the safety, can be stateful. In this case, due to the loss of states after the compromise, system will inevitably suffer a performance loss, but the safety will not be violated. This can be another limiting factor on the type of systems or the kind of safety constraints imposed on it that needs to be considered when using our approach.

CHAPTER 7: SAFETY IN DISTRIBUTED CPS

In the previous chapters, we looked into the problem of maintaining the safety of single node CPS under faults and adversarial attempts. We defined a single node CPS as a system that consists of a controller actuating a physical plant and attempting to guide the plant's state into a desired state. The safety premise was implied as a set of restrictions on the physical state of the plant with regards to the environment.

In this chapter, we investigate the safety of distributed CPSs. The safety premise of each node is defined with regards to the environment as well as the state of other nodes in the system. Distributed CPSs combine network communications along with interactions with the physical world. In particular, we consider a CPS scenario consisting of several embedded computing components each interacting and sensing the physical world and communicating with a central coordinator over an unreliable channel, such as wireless or the Internet. A distributed CPS is considered globally safe if and only if *all* the nodes are safe. These low-level controllers attempt to accomplish some task in a coordinated fashion. Since the physical world is being manipulated, it is essential that the supervisory control logic is carefully designed and satisfies strict safety requirements. For example, autonomous vehicles may use wireless to communicate their positions and alter their future routes but vehicles should never collide despite the potential for an unbounded number of message drops. This system is difficult to reason about because both (1) the communication layer can experience unbounded message delays and drops and (2) the dynamics of the physical world are represented by interacting relationships in a continuous space.

An example of a distributed CPS is autonomous coordinated vehicle motion. A set of vehicles is moving through a shared physical space, and the user would like to be able to make run-time changes to the routes of the vehicles while guaranteeing that vehicles will not collide. Since messages may be lost over wireless, any new route information may arrive at some vehicles but not at others. Acknowledgments will not solve this problem since they may also sometimes be lost. As in distributed systems with lossy communication, it is impossible to achieve consensus in this system [91]. Despite this inherent limitation, I propose an approach that can ensure the safety invariant that collisions are avoided. If the communication channel eventually delivers packets, we can also provide the notion of progress that gives the vehicles the ability to safely modify their routes at run-time.

In the context of a distributed CPS, a designer is typically interested in two properties: *safety* and *progress*. A proof of safety will guarantee that the system will never enter an undesirable state. We formally specify safety as a predicate on the variables of the agents of

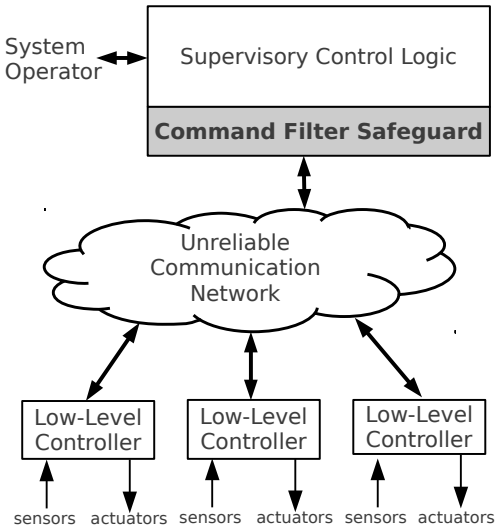


Figure 7.1: A Runtime Command Monitor ensures safety for the distributed cyber-physical system.

the distributed CPS which is true at all times (a safety invariant). The notion of progress that we consider is that, roughly speaking, all the agents will receive and follow a desired goal command in finite time. The ultimate guarantee that we provide is that the system will remain safe at all times (even if the network fails), while being able to meet the progress property as long as the communication network is functioning.

The scope of the work presented in this chapter will be the verification of the high-level control logic of the distributed CPS and not the verification of the individual controllers. We will therefore assume that the implementation of the individual low-level controllers is correct and bug-free. For example, upon receiving a command message, a low-level controller will follow that command as intended. Ensuring this is also non-trivial, but it is likely a more tractable problem for formal design approaches since each low-level system contains less variables than the composed system. Additionally, techniques such as the Simplex architecture [2, 92] may be used to guarantee certain behavior properties for low-level controllers, even if the complete controller is not directly verifiable. For guaranteeing progress, we further require that low-level controllers are locally exponentially stable.

An overview of the type of distributed CPS we consider is shown in Figure 7.1. Notice that our system uses a communication network where every controller can communicate with every other controller. The key enabler of our safety result is the realization that, if the network is assumed to be unreliable, individual low-level controllers need be able to maintain global safety even in that case that packets do not arrive. Safety here means that a given predicate on the state space will evaluate to true over all time. We propose a *Runtime*

Command Monitor interposed between the supervisory control logic and the network (as shown in the figure). If the supervisory control logic attempts to send control commands which, for any amount of message delay, can lead to a system state that violates the safety predicate, the Runtime Command Monitor will reject commands that could lead to unsafe states. We show that this design results in a fail-safe system.

The main technical challenge that is to determine the exact behavior for the Runtime Command Monitor for a particular distributed CPS system. In the defined approach, the runtime command monitor's decision depends upon a possible expensive online reachability computation. However, in order to make the technique more practical, we provide a theory for *reachability reduction transformations* that can simplify the reachability operation and allow us to perform *state and input enumeration* offline. The offline results can then quickly be applied at runtime to guide the behavior of the Runtime Command Monitor.

Since the network is unreliable, control commands that are sent from the supervisory logic may never arrive at a low-level controller. In order to be able to update the system behavior based on run-time information (progress), therefore, a stronger requirement must be imposed upon the network. As long as messages eventually arrive, we also provide a means to guarantee system progress while maintaining safety. To do this, we must ensure that commands that are sent out maintain the safety invariant both in the case where the command arrives and the new control strategy is used, and in the case where the command is indefinitely delayed. This notion of safe potential divergence is captured as *compatible actions*. We show that system progress properties can be guaranteed by constructing finite chains of compatible actions that end at the final desired system state.

The main contributions are as follows:

- We prove that run-time properties provide necessary and sufficient conditions for safety in a distributed CPS system. By encoding these checks into a Runtime Command Monitor, a fail-safe system can be developed (Section 7.1).

- The proposed run-time properties require computing the reachability of the system online that can be an expensive operation. Through a combination of *reachability reduction transformations* and *input and state enumeration*, we perform this operation offline (Section 7.3).

- We provide sufficient conditions for providing progress guarantees. This requires constructing a chain of compatible actions, as well as a network which eventually delivers packets that are sent. (Section 7.2)

- We apply the approach to a simulated system of vehicles moving in a shared

environment, where the runtime command monitor prevents vehicle collisions. (Section 7.4)

7.1 PROVIDING SAFETY

In this section, we use hybrid input/output automata to formalize the notion of a distributed networked control system with arbitrary delays and packetloss. We then prove a general theorem which is both a necessary and sufficient condition for the safety of such systems. We then apply the theorem by stating the run-time checks in order to maintain system invariants, which will be encoded into the Runtime Command Monitor in the proposed architecture.

7.1.1 Hybrid I/O Automata

Hybrid input/output automata are general models for systems consisting of discrete and continuous states, where the discrete states are governed by transition rules, and the continuous states evolve according to differential equations. There is also input and output in these systems, which allows easy composition of different components into a larger system.

Rather than explaining the full semantics for hybrid I/O automata, we provide a brief overview of only the most important aspects here, and refer an interested reader to a more comprehensive review [93, 94].

A n-dimensional hybrid I/O automaton consists of four parts: variables, transitions, trajectories, and actions. **Variables** are the discrete or continuous entities of an automaton, for example velocity or mode. A state of an automaton is a specific valuation of the variables. The state space of an automaton is $X = L \times \mathcal{X}$, where L is the set of possible discrete states (also called locations), and $\mathcal{X} \in \mathbb{R}^n$ is the set of possible continuous states. **Transitions** provide the behavior of the discrete variables in the system. These have an enabling precondition (a predicate on the continuous states) and an effect (a mapping on the continuous states). The state after the effect is applied is called the post state of the transition. Preconditions specify when transitions can occur, but generally automata are not forced to take a transition, which can create nondeterminism. **Trajectories** give the behavior of the continuous variables in the system as time passes, typically using differential equations, and systems can also have nondeterministic dynamics described by nondeterministic differential equations. The conditions under which time cannot advance are given as stop conditions, which can be used to force an enabled transition to occur. Finally, **actions** indicate the interaction points for external communication with other automata.

An action will always have a corresponding transition in the automaton. An action can occur when both automata that have the action satisfy the corresponding transitions' preconditions.

Time passes for a hybrid automata when a trajectory is acting upon the continuous variables. During the execution, there can be discrete jumps in state caused by the transitions. For two hybrid I/O automata with compatible actions, say A and B, we denote their composition using $A||B$.

7.1.2 System Definition

We model our supervisory control system as a network of communicating hybrid I/O automata. In this network, there is an automaton describing the behavior of each of the N agents in the system, $A_1, A_2, \ldots, A_N$, and an automaton which models the communication channel. This model is slightly more general than the one discussed earlier with an explicit supervisory controller. Here, we could arbitrary choose one of the agents to be the supervisor.

In this section, we are concerned with verifying that a predicate is a safety invariant for a system. That is, we are provided with a safety predicate on the states of the agent automata. The predicate is an invariant if it evaluates to true for all reachable states of the system from a given initial state (an unsafe state cannot be reached). A system is a composition of the agent hybrid I/O automata and the communication automaton.

For our unreliable network, we consider a communication automaton with weak guarantees about message delivery, named C_{weak}, which can delay each message arbitrarily long, or drop it. Such an automaton matches the communication properties of many networked or wireless communication systems. The automaton description for C_{weak} is given in Figure 7.2. Here, there are two possible send transitions, either of which can be applied when a message is sent out. The first one assigns a real-valued arrival time greater than the current time. The second one silently drops the packet. We also will consider two other communication scenarios, C_{drop} and C_{strong}. In C_{drop}, the first send transition of C_{weak} is omitted so all messages get dropped. In C_{strong}, the second send transition is omitted, so that all messages can only be arbitrarily delayed, but never dropped. A communication automaton would be composed with each of the agent automata by connecting the receive transition with destination i to Agent A_i. All the agents would invoke the same send transition.

```
automaton CommWeak(M : Type)
type Packet = tuple of message: M, delay: Real, dest : Nat
  variables
  internal bag : Set[Packet] := [],
    now : Real := 0
  actions
    send(m: M, dest: Nat),
    receive(m: M, dest: Nat)
  transitions
    send(m, dest) // not in CommDrop
      effect
        bag := insert([m,now+rand(),dest]
    send(m, dest) // not in CommStrong
      effect
        /* dropped */
    receive(m, dest)
      precondition
        contains(bag, [m,0,dest])
      effect
        remove(bag, [m,0,dest])
  trajectories
    stop when
      ∃p: Packet p ∈ bag ∧ (now = p.deadline)
    evolve
      d(now) = 1
```

Figure 7.2: The C_{weak} communication automaton assigns messages arbitrary delays and can drop messages. Here, rand() returns a nonnegative real number.

7.1.3 Safety Theorem

In order to prove a predicate P is an invariant for a system given a definition for each agent automaton and the communication automaton, a standard approach is to check that the invariant is satisfied for every transition and every trajectory. During this process, the invariant may need to be strengthened in order for the proof to follow.

The standard approach for proving invariants, however, can be difficult to apply. Since reasoning is done ahead of time, the analysis must be applicable to all states which can be encountered for each rule.

Here, we present an alternative approach for creating invariant-satisfying systems. Here, we will use a combination of static reasoning done ahead of time along with run-time checks. With this approach, we can sometimes guarantee an invariant in an easier manner than by using the normal, static-only approach. Rather than reasoning over sets of possible values, we instead move part of the checking to run-time, and can therefore use a specific value in a specific message. In order to do this, however, we need to prove a theorem which provides an equivalent condition for verifying invariants.

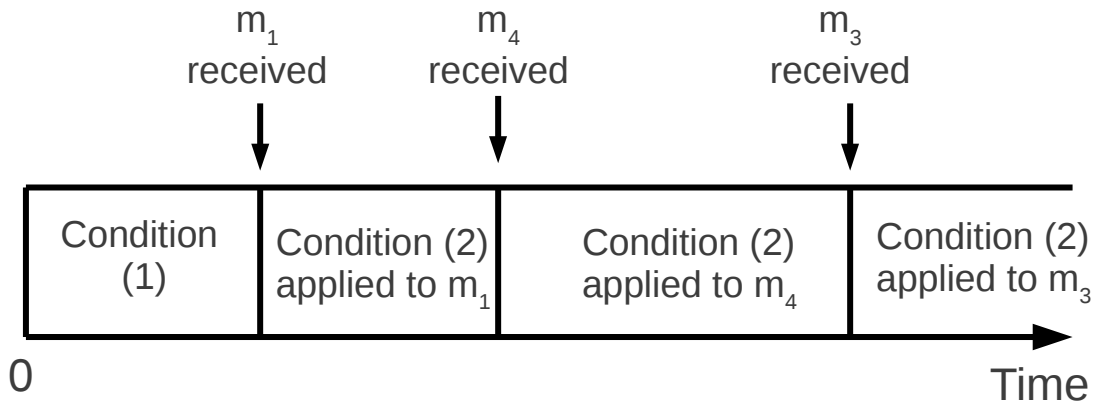


Figure 7.3: For every trace, at each time instant, either no message has been received in the system, or there is a most-recently received message.

A system is described by a composition of the automaton for each of the agents ($A^N = A_1||A_2||\ldots||A_N$) and the automaton for the communication channel. A property P is the predicate we are trying to show is an invariant, and is a predicate on the states of the agents, P: $A^N \rightarrow$\{true, false\}.

Theorem 7.1. *A predicate P is an invariant for a system $S = A^N||C_{weak}$ if and only if (1) P is an invariant for the system $S' = A^N||C_{drop}$, and (2) from any post state of a* `receive` *transition in S, P is preserved by the system $A^N_{post}||C_{drop}$, where A^N_{post} is the composed agent automata A^N starting in the post state of the* `receive` *transition.*

Proof. First we show the direction that if conditions (1) and (2) hold, the invariant is satisfied by the original system.

The proof of this statement is based on the observation that at every point in time, either no messages have been received, or there is a most-recently received message by one of the agents. As shown in Figure 7.3, for every possible trace there will be some amount of time where no messages have been received by any of the agents in the system, followed by a intervals of time where there is a most-recently received message.

Our proof proceeds by contradiction. Assume t_i is the first time at which P is evaluated to false in S. If t_i occurs before the first message is received, this means that P would also evaluate to false in S' at time t_i, since up to this point the behavior of S and S' is identical. This violates condition (1).

Therefore t_i occurs at or after a message has been received and processed. Let t_m be the time of the most-recently processed message before time t_i (the time at which the `receive` transition was invoked in C_{weak}). We apply condition (2) of the theorem at time t_m and

take A^N_{post} as the composed agent automata in the post state of the `receive` transition in S. Since in S, P evaluates to false before any further messages are received after t_m, this would mean it also evaluates to false for the system with agent automata A^N_{post} and a communication automaton which does not receive any messages. This is exactly the case checked by condition (2).

Next we show the other direction, that if a predicate P is an invariant for S, conditions (1) and (2) will hold. Again, we proceed by contradiction.

Assume condition (1) does not hold but P is an invariant of S. The behaviors of C_{drop} can be exactly simulated by C_{weak}, which means that P cannot be an invariant for S.

Next, assume the second case that condition (2) does not hold but P is an invariant for S. In the context of the false case of condition (2), let time t_m be the time at which the `receive` transition is invoked. Now consider a communication automaton which produces an identical behavior as S until t_m and then no longer receives messages. This behavior can also be exactly simulated by C_{weak} (by taking the dropping `send` transition for messages which would originally have an arrival time after t_m), which means that P cannot be an invariant for S.

Since both cases yield contradictions, if an invariant is satisfied in the original system, conditions (1) and (2) must also hold.

The two conditions of the theorem are therefore both necessary and sufficient for proving an invariant is satisfied for a system with unreliable communication over all time.

7.1.4 Application of Theorem to Runtime Command Monitor

From a static-time analysis perspective, the theorem does not gain us very much since condition (2) needs to be evaluated every time any message can be received, which is difficult to reason about. However, at run-time, condition (2) may be easier to verify. This is the approach advocated, to check condition (1) at system design time and condition (2) at run-time, which by the theorem will guarantee that P is an invariant of the system.

One challenge of this approach is that the necessary run-time analysis needs to be automated in software, which is done in our architecture in the Runtime Command Monitor. Since there may be nondeterminism from the dynamics of the agents, and since in general this may involve an infinite-time reachability computation, this may be easy or hard depending on the specific system.

In terms of applicability, one main concern that we will evaluate further in our case study in Section 7.4 is the run-time overhead of the approach, which is application-specific. If we consider a typical case of time-invariant systems where low-level controllers are stable from

a control-theoretic sense, and the commands are new set points, then the potential area the agent may enter given some unknown delay consists of the states it will encounter while transitioning from the old set point to the new one, projected over all future time (since delay is unknown). To check condition (2), this transition area would be computed and checked with the future states the other agents may enter against the safety predicate.

Another consideration is to specify the action to take if the analysis for the specific message indicates condition (2) is *not* satisfied at run-time. The system cannot be allowed to take action based on the message, since it may lead to a state which violates the invariant. In our proposed design, these messages are filtered (never sent out) by the Runtime Command Monitor. This preserves condition (2) for the system (since no messages will be sent out unless (2) is satisfied) which guarantees that P will continue to be an invariant for the system. Of course dropping messages can adversely affect system progress, but it will only be done to maintain safety (if the predicate captures a notion of safety). In Section 7.2, we present sufficient conditions to guarantee progress which require, among other things, a stronger communication automaton, where messages can be delayed arbitrarily but not dropped.

Since the Runtime Command Monitor drops messages at send time, it needs to reason about possible system states when the packet will be received (since condition (2) deals with the system state upon message reception, not sending). This also may be challenging because, for unrestricted systems, it involves reasoning about which messages may be sent out in the future before the arrival time of the message, and possible message reordering. For example, in Figure 7.3, message m_4 arrives before message m_3. The run-time analysis at the send time of message m_3 needs to take this possible reordering into account. Also, in an unrestricted system, these messages can be sent from and arrive at different agents (for example m_3 may be from Agent 1 to Agent 2, while m_4 is from Agent 3 to Agent 4). For specific systems, however, this analysis may be simpler. For example, systems which maintain sequence numbers in messages and only take actions on the most-recent messages received, do not have to consider reordering. Systems like the supervisory control system we are considering have a single entity which sends command messages, and therefore we do not need to reason about command messages exchanged between other agents. As matches our intuition, having guaranteed orders of packet delivery produces systems that are easier to predict and prove correct, whether using the standard static-time approach or our run-time technique. Condition (2) of the theorem demonstrates this, while, at the same time, tells us what would need to be checked for the more general case.

7.2 GUARANTEEING PROGRESS

We will now describe a manner in which we can guarantee a time-insensitive notion of safe system progress. We assume a more specific CPS model here where each agent is running a stable closed-loop controller.

First, we discuss the distributed control system architecture that we consider more specifically in Section 7.2.1. Section 7.2.2 defines the notion of compatible actions in the context of the distributed control system and proposes methods of checking compatibility. In Section 7.2.3, we then show scheme of coordinated control that guarantees safety according to our earlier result from Section 7.1. Finally, Section 7.2.4 proves progress of the system under a stronger assumption of the communication layer.

7.2.1 Controller Architecture

As before, we consider a distributed control system consisting of a collection of N agents with a central coordinator. We assume that each agent receives commands only from the central coordinator. Each Agent A_i has a *local controller* and a *current set point*. The current set point of Agent A_i can be changed through communication with the central coordinator. In this section we will assume the current set point to be is a single goal position of A_i. That is, the local controller of Agent A_i drives the agent's continuous variables to move towards the current set point. When agent A_i reaches an ϵ-ball around the set point (for some fixed ϵ), agent A_i will report its arrival to the central coordinator by sending a progress update message. The central coordinator will then, upon receiving arrival messages from all the agents, send each agent its next set point. Each agent, thus, receives a sequence of set points. The kth set point in this sequence for agent A_i is written as $\mathbb{S}_i[k]$. An execution of Agent A_i can be viewed as a hybrid sequence $\eta_i = wait_i[0] \frown receive[1] \frown \tau_i[1] \frown send \frown wait_i[1] \frown receive[2] \frown \tau_i[2] \frown send \frown wait_i[2] \ldots$, where (i) each $\tau_i[k]$ is a trajectory moving towards set point $\mathbb{S}_i[k]$, (ii) *send* is discrete action where the Agent sending the progress update message, (iii) $wait_i[k]$ is a trajectory when waiting for next set point, where agent A_i stays within the ϵ-ball of $\mathbb{S}_i[k]$, and (iv) $receive[k]$ is a discrete action invoked by the central coordinator's send action, during which the set point of agent A_i is changed from $\mathbb{S}_i[k-1]$ to $\mathbb{S}_i[k]$. In each trajectory $\tau_i[k]$, the initial state and the final state of the trajectory are within ϵ-balls of successive set points of A_i. A global set point is defined as a collection of the local set points for each of the N agents, and is denoted as $\mathbb{S}^N$.

In this section we are concerned with progress, but the progress must be made cognizant of safety. As in Section 7.1, safety is defined in terms of a predicate P_S. The progress property

is defined using a global set point $\mathbb{S}^N_{final}$. The formal notion of progress we prove is that each agent will, in finite time, reach within an ϵ-ball around its set point in $\mathbb{S}^N_{final}$, while always having P_S evaluate to true.

7.2.2 Compatibility and Stability

Section 7.1 showed that in order to ensure safety, the central coordinator needs to reason about future states of A_i, and will therefore issue set points according to the states A_i can reach. Reasoning about future states of A_i can be done using reachability analysis. We denote $Reach_i[k]$ as the set of reachable states of A_i under trajectory $\tau_i[k]$. The reachable set of the global system (the composed behavior of all the agents) is denoted as $Reach^N$. For safety of the system, we need to verify that $Reach^N$ satisfies the safety predicate P_S. Recall that a trajectory $\tau_i[k]$ of A_i depends on two set points, $\mathbb{S}_i[k-1]$ and $\mathbb{S}_i[k]$, of A_i. For a specific set point $\mathbb{S}_i[k]$, we check whether P_S remains true over the composed $Reach^N[k]$ by computing the reachable set of states for each of the other agents. This property of safety for a new global set point captures the notion of *compatible actions*.

Definition. $\mathbb{S}^N[k]$ and $\mathbb{S}^N[k+1]$ are said to be pairwise **compatible actions** if the global state $x^N \in Reach^N[k]$ always satisfies P_S when every A_i moves along a trajectory defined by $\mathbb{S}_i[k]$ and $\mathbb{S}_i[k+1]$.

The notion of compatible actions can also be generalized to n-way compatible actions. That is, given n collections of set points, we can say they are n-way compatible if the global state always satisfies P_S when every agent moves along a trajectory defined by any pair of the set points. Due to the extra requirements, however, it is generally easier to construct chains of pairwise compatible actions. For this work, we use pairwise compatibility, and perhaps investigate applications of n-way compatible action chains in future research.

7.2.3 Safety Guaranteed Run-Time Checking

We assume low-level controllers which are *locally exponentially stable*, and start from a safe global set point.

Definitions. A controller is said to be **locally exponentially stable** with respect to a set point, if there exists a neighborhood of the set point such that any trajectories starting from any state in a neighborhood of the set point, converges towards to the set point. In addition, the distance between the trajectory and the set point decays exponentially over time. The neighborhood is also called the **region of attraction**, which defines the maximum region from where the set point will be reached. Even though the distance between a trajectory

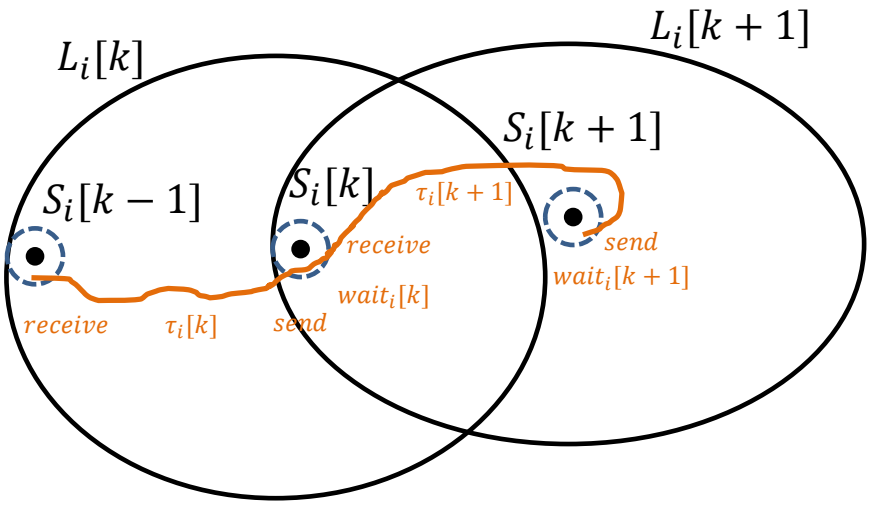


Figure 7.4: An execution trace (orange) in which agent A_i received set points $\mathbb{S}_i[k-1]$, $\mathbb{S}_i[k]$, and $\mathbb{S}_i[k+1]$ in sequence. Initially, the agent is near set point $\mathbb{S}_i[k-1]$. Since it is inside the Lyapunov region of attraction when set point $\mathbb{S}_i[k]$ is used (indicted by $L_i[k]$), control can be switched to use the new set point, $\mathbb{S}_i[k]$. An epsilon ball around this set point (the dotted blue line) will be reached in finite time, because the controller is exponentially stable. When all agents have arrived at their new set point, the central coordinator sends new set points and the process repeats for the next one, $\mathbb{S}_i[k+1]$, with a region of attraction indicated by $L_i[k+1]$.

and a set point is exponentially decaying, the set point may never be reached *exactly* in a finite amount of time. However, any ϵ-ball around the set point will be guaranteed to be reached in finite time. We call the state of the system where each agent is within an ϵ-ball of its corresponding setpoint is called an ***ϵ-stable*** state.

An initial safe global set point is, formally, an *ϵ-stable* state where the union of each of the agent's ϵ-balls contains no unsafe states.

The behavior of the supervisory control logic is:

(1) Until receiving progress report updates from all the agents, indicating that each agent is within an ϵ-ball of the current set point $\mathbb{S}^N[k]$, the central coordinator will not send any new set points. This means progress is not made until an ϵ-stable state is reached.

(2) Once an ϵ-stable state is reached and progress updates are received from each agent, the server issues a new collection of set points $\mathbb{S}^N[k+1]$ following conditions below, and sends them to the corresponding agents.

(2a) The global set point $\mathbb{S}^N[k+1]$ should be compatible with the global set point $\mathbb{S}^N[k]$. That is, the all reachable states of the system do not violate the predicate P_S.

(2b) For each agent, the ϵ-ball of its set point in $\mathbb{S}_i[k]$ should be contained by the region of attraction of its set point in $\mathbb{S}_i[k+1]$, which guarantees that the next set point will be reached by the low-level controller.

This situation is illustrated for one agent in Figure 7.4. In the figure, $L_i[n]$, $n \in \mathbb{N}$ represents the Lyapunov region of attraction for a controller with set point $\mathbb{S}_i[n]$.

Lemma 7.1. *Safety predicate P_S is an invariant of the system which uses the above-described supervisory control logic.*

Proof. Recall the safety theorem from Section 7.1, which requires checking two conditions to show a predicate is an invariant.

Condition (1) holds because if all packets get dropped at the beginning, the set points never change. Since the initial state is assumed ϵ-stable and the union of the ϵ-balls is safe, and P_S remains true.

Condition (2) requires that P_S remains true in the system $A^N_{post}||C_{drop}$, where A^N_{post} is the system after receiving any packet. Suppose that after a packet gets delivered, all follow-up packets get dropped. This causes the server will stop sending new set points since not all reports will be received for the next step (part 1 of the supervisory control logic described above). This is also true in the case that some packets were dropped earlier before the packet being considered. In this case, the system can only be conservative in that it might not advance to the next global set point (which is safe). Since the central coordinator will not send out any new set points, the agents will remain using the current set point for all future time. Agent i's states will therefore remain in the pair-wise compatible reach set, and by pair-wise compatibility, P_S will remain true.

Since both parts of the antecedent of theorem are satisfied, we can conclude the consequent, that P_S is an invariant of the system.

7.2.4 Progress Guarantee

We will now discuss a sufficient condition to guarantee system progress. Formally, we want the system reach a target global set point $\mathbb{S}^N_{final}$ in some finite amount of time.

To guarantee progress, we require three requirements. First, messages in the network can only get delayed arbitrarily long, but cannot be dropped. For this assumption we will use automaton C_{strong}, as described in Section 7.1.2. In practice, this such a communication model is achievable, for example, by having a low-level network layer which keeps resending packets until an acknowledgment is received, assuming the connection will eventually get reestablished. Second, there is a finite chain of pairwise compatible actions (which we call a *compatible action chain*) from the current state to the target global set point $\mathbb{S}^N_{final}$. Third, the local controllers for each agent are exponentially stable for each set point in the compatible action chain.

Theorem 7.2. *The system $A^N||C_{strong}$ will, in finite time, get to a state where each of the agents A^N is within an ϵ-ball of their corresponding set point in the target $\mathbb{S}^N_{final}$.*

Proof. Recall that agent A_i's execution is a hybrid trace $\eta_i = wait_i[0] \frown receive[1] \frown \tau_i[1] \frown send \frown wait_i[1] \frown receive[2] \frown \tau_i[2] \frown send \frown wait_i[2] \ldots$. First, $\tau_i[k]$ is a trajectory starting from an ϵ-ball of $\mathbb{S}_i[k-1]$ to an ϵ-ball of the $\mathbb{S}_i[k]$. Since we assumed the local controller is exponentially stable, the distance between the continuous state of A_i and the set point is exponentially decaying. Thus, any ϵ-ball of the set point will be reached in a finite time. The exact amount of time can be computed from the constant value of the exponential in the stability property of the controller. Second, a send action through C_{strong} takes finite delivery time to invoke a receive action of the coordinator. Since this is true for all agents, the coordinator will receive all the reports of progress in a finite time. At this point the next set point will be sent back to A_i. This sending also takes a finite time since it is done by C_{strong}. Due to this, the $wait_i[k]$ trajectory where A_i is waiting for a new set point has a finite duration. Finally, since by the second requirement the chain of pair-wise compatible actions is finite, the target $\mathbb{S}^N_{final}$ is reachable through finitely many of these steps. By this reasoning, we conclude that the execution of η_i will reach $\mathbb{S}^N_{final}$ in a finite amount of time.

A system designer may want a stronger guarantee of progress that the final set point will be reached by all agents after some exact amount of time (rather than just finite). In order to prove these stronger progress properties, we can adapt the same proof as above, while imposing limits on each of the steps which were previously only required to take finite time. If the network guarantees packet delivery with a worst-case transmission time, and we know the exponential constants of our locally exponentially stable controllers, we can compute the maximum amount of time it can take for the system to go from one known set point $\mathbb{S}_i[k-1]$ to the next known set point $\mathbb{S}_i[k]$. We can compute the maximum amount of time for the system to complete the entire compatible action chain by summing up the maximums for each pair of compatible actions. In this way, the maximum amount of time that can elapse before reaching the final set point can also be calculated.

A last note about compatible action chains is that their construction is application-specific and may be nontrivial, or even impossible (some systems cannot safely make progress under our communication assumptions). This is because the safety predicate P_S depends on the application, and different safety predicates may required different schemes to create compatible action chains. For our progress guarantee, we assume that there is some means to construct a compatible action.

7.3 ELIMINATING RUNTIME REACHABILITY

Reachability is a potentially expensive operation which may be too slow to perform at runtime. In this section, we provide methods which move key aspects of the reachability computation to analysis time. Specifically, we rely on two strategies for doing this: finding *reachability reduction transformations*, and *input and state enumeration*.

7.3.1 Reachability Reduction Transformations

In our proposed framework, we propose running many reachability computations offline, prior to the execution of the system, and then applying their results at runtime. The number of these computations, however, may be excessive such that even offline analysis is intractable. In order to alleviate this problem somewhat, we examine the system dynamics and attempt to extract redundancies in the computation which allow us to reduce the amount of computation that is necessary. We first introduce an example vehicle system to guide the intuition of our approach. Later, in the case study in Section 7.4, we will consider several vehicles with the dynamics shown below, and combine the approach in this section with the theory developed in the previous sections.

Consider an example system consisting of a single vehicle moving in 2-D space. The vehicle has 1 location and 4 continuous variables which are (i) the x-coordinate, (ii) the y-coordinate, (iii) the traveling speed v, and (iv) the heading angle θ. The vehicle inputs are (i) an acceleration/deceleration rate a, and (ii) a turning curvature ρ. The motion of the vehicle (the trajectories of the automaton) behaves according to the following set of differential equations:

$$\begin{cases} \dot{x} &= v \cdot \cos\theta \\ \dot{y} &= v \cdot \sin\theta \\ \dot{v} &= a \\ \dot{\theta} &= v \cdot \rho \end{cases} \tag{7.1}$$

Additionally, as with a real system, we bound the acceleration/deceleration rate, the allowed turning curvature, and the velocity. That is, a vehicle can not stop right away, turn in place, or accelerate indefinitely. In order to analyze the future possible states, we compute over-approximations of reachability.

With this system, it is intuitive that any trajectory starting from a point (x, y) is identical to the trajectory starting from the origin $(0, 0)$ shifting by a vector of (x, y). Some rotational invariant property can also be observed in this model: a trajectory with initial heading θ is identical to the trajectory with initial heading 0 rotated by a angle of θ. These observations

intuitively mean that, by transforming a reach set from one specific configuration, we can derive reach sets from a number of other configurations. In this way, the space of offline reach set computations can be largely reduced.

We now formalize and generalize the above intuition. Recall that for a specific location, the continuous state space is written as $\mathcal{X}$. We denote the continuous reach set in the location under analysis as a mapping $Reach : \mathcal{X} \times \mathbb{R}_{\geq 0} \to 2^{\mathcal{X}}$. For any state $x \in \mathcal{X}$ and time $t \in \mathbb{R}_{\geq 0}$, the function $Reach(x, t)$ maps to a subset of the state space that is reachable in the given location from state x within time t[1]. We define a reachability reduction transformation as follows.

Definition 7.1. *Let F, G be two functions defined on the continuous state space of a location of an n-dimensional hybrid automaton $F, G : \mathcal{X} \to \mathcal{X}$. The pair (F, G) is called a* reachability reduction transformation *if for every state $x \in \mathcal{X}$ for any time $t \in \mathbb{R}_{\geq 0}$, the following holds:*

$$Reach(F(x), t) = G(Reach(x, t)).$$

With a reachability reduction transformation (F, G), we can compute the reach set from state $F(x)$ by simply applying function G to the reach set from x, which is typically faster than computing another reach set. This approach becomes more powerful if we can find a parameterized set of such reachability reduction transformations $\{(F_i, G_i)\}_{i \in \mathbb{R}^m}$ for some $m \leq n$ the number of dimensions of the automaton. We will give an example of such set of reachability reduction transformations for the above vehicle dynamics.

Example 7.1. *For the vehicle dynamics presented in Equation (7.1), for each vector $i \in \mathbb{R}^3 = (x_i, y_i, \theta_i)$, the following pair (F_i, G_i) is a reachability reduction transformation.*

$$F_i\left(\begin{bmatrix} x \\ y \\ v \\ \theta \end{bmatrix}\right) = \begin{bmatrix} x + x_i \\ y + y_i \\ v \\ \theta + \theta_i \end{bmatrix}, \; G_i\left(\begin{bmatrix} x \\ y \\ v \\ \theta \end{bmatrix}\right) = \begin{bmatrix} (x + x_i) \cdot \cos\theta_i - (y + y_i) \cdot \sin\theta_i \\ (x + x_i) \cdot \sin\theta_i + (y + y_i) \cdot \cos\theta_i \\ v \\ \theta + \theta_i \end{bmatrix}.$$

Proof. For state $s = (0, 0, v, 0)$, for any time t, the reach set $Reach(s, t)$ from s can be

[1]For a *set* of states $x' \subseteq X$, $Reach(x', t)$ denotes the range of function $Reach$ on the set x'.

computed by integrating Equation (7.1):

$$\begin{array}{rcl} x(t) & = & \int_0^t v\cos\theta dt, \\ y(t) & = & \int_0^t v\sin\theta dt, \\ v(t) & = & v + \int_0^t a dt, \\ \theta(t) & = & \int_0^t v\rho dt. \end{array} \tag{7.2}$$

Let $F(s) = (x_0, y_0, v, \theta_0)$ be an arbitrary start state, we take $i = (x_0, y_0, \theta_0)$. The reach set $Reach(F_i(s), t)$ then has the form:

$$\begin{array}{rcl} F_i x(t) & = & x_0 + \int_0^t v\cos(\theta+\theta_0)dt, \\ F_i y(t) & = & y_0 + \int_0^t v\sin(\theta+\theta_0)dt, \\ F_i v(t) & = & v + \int_0^t a dt, \\ F_i \theta(t) & = & \theta_i + \int_0^t v\rho dt. \end{array} \tag{7.3}$$

We can apply triangular identities

$$\cos(\theta+\theta_0) = \cos\theta\cos\theta_0 - \sin\theta\sin\theta_0, \text{ and } \sin(\theta+\theta_0) = \sin\theta\cos\theta_0 + \cos\theta\sin\theta_0$$

to Equation (7.3). By comparing the above derived equation and Equation (7.2), we can conclude that $Reach(F(s), t) = G(Reach(s, t))$.
□

Notice that, through this analysis, we did not get any closed-form solution of the reach set (Equation (7.2) is in integral form). To make use of the reachability reduction transformation, therefore, we still need a method to compute the reach set from a state $s = (0, 0, v, 0)$.

7.3.2 Input and State Enumeration

In the safety theorem provided in Section 7.1, reachability needs to be computed from a specific state. We want to move this computation to analysis time, however, we do not know what this state will be at runtime. One way to resolve this is problem to perform several reachability computations ahead of time, each from a subset of the possible states the system may be in, and then select the appropriate result at runtime (state enumeration).

Formally, if $D \subseteq X$ are the possible states we may encounter at runtime, we precompute the reach set from each set of a finite covering of states we may encounter$d_1, d_2, \ldots, d_m$, where $d_1 \cup d_2 \cup \ldots \cup d_m \supseteq D$. There is an inherent trade off with this approach, since having

a larger m will result in a more precise reachability result (at runtime we're in a single state), at the cost of having to do more computation work ahead of time.

A second problem, is that the input for the automaton is also unknown ahead of time. This is potentially more problematic, since the inputs are functions of time, and it may be difficult to enumerate all possibilities. Notice, however, that in the context of the Runtime Command Monitor's check, the reachability computed will be the system composed with C_{drop}, that is, the communication system which drops all packets. It is more reasonable that the actions to be performed after communication stops are limited, and therefore we may also be able to enumerate the possible inputs (input enumeration).

Formally, if the set of possible input functions is I, as before, we precompute the reach set from each set of a finite covering of inputs we may encounter $i_1, i_2, \ldots, i_{m'}$, where $i_1 \cup i_2 \cup \ldots \cup i_{m'} \supseteq I$.

The end result is that we enumerate over the possible states and inputs we may encounter at runtime, and then, at runtime, select the corresponding result set. This enumeration may be quite large. However, by applying reachability reduction transformations, and adjusting the parameters m and m', we can tune the computation time needed against the pessimism experienced due to using a larger-than-necessary start state set and input bounds.

7.4 VEHICLES IN A SHARED ENVIRONMENT

In order to demonstrate the effectiveness of reachability reduction transformations and input and state enumeration in the context of a distributed CPS, we now discuss a case study using the approach.

In the considered system, several independent vehicles are moving on a 2-D plane, trying to reach a destination while avoiding collisions. The simulated vehicles are mobile nodes that can accelerate forward and backwards and turn similar to cars. This means they cannot rotate in-place and instead there is a minimum turning radius. Since the acceleration and deceleration is limited in a range, a sudden change of speed is not possible. The vehicles use the dynamics described previously in Section 7.3. Additionally each vehicle has a radius r, and a collision occurs if the centers of any two vehicles is less than twice the vehicle radius.

Similar to a real car, each vehicle moves based on two inputs variables, an acceleration and a turning curvature input, which we refer to as a *command*. Each vehicle communicates with supervisory control logic called the *central coordinator*. Vehicle commands are initially generated by each of the agents, sent to the central coordinator which can either accept or reject the message, and then the decision is sent back to the originating agent. If no new

commands are accepted one second after a command is applied, the vehicle stops turning and slows to a stop.

If the network works properly, and all the input commands are compatible with global safety (collision avoidance), then the control strategy is equal to one where there is some time delay due to the round trip time of the network. However, at any point in time the central coordinator may reject a command, in which case the vehicle would eventually slow to a stop.

In Section 7.1.3, when proving the system will remain safe, the reasoning consisted of command messages coming from the central coordinator being sent through an unreliable network. In the system here, however, the command messages are being sent to the central coordinator, which initially appears to be a different type of system. This sending of the command requests to the central coordinator can be considered as happening before the reasoning done in the theorems. After all, the way in which command messages are generated in Section 7.1.3 is not discussed. In this case they are generated from requests from the vehicles. After generating the command messages, the central coordinator will then send the response to the agent (accept or reject) which is ultimately what the agents will act upon (their behavior does not change based solely on the requests). In this way, the distributed CPS architecture is similar to that discussed in the previous sections and the proofs can be applied.

7.4.1 Design-Time Computation

There are N agents (vehicles) in our system, each with the vehicle dynamics as described previously in Section 7.3. We refer to the continuous components of the agent i state using $A_i.x, A_i.y, A_i.v, A_i.\theta$. Global safety in our system is that the vehicles are collision free. Formally, the safety predicate P is that the distance between every pair of vehicles A_i, A_j, where $i \neq j$, must be more than twice the vehicle radius; that is $P = \forall_{i,j|i\neq j}(A_i.x - A_j.x)^2 + (A_i.y - A_j.y)^2 > (2r)^2$.

The central coordinator maintains a reachable set of states for each vehicle under the current command strategy. Since the initial state of the vehicles is stationary and collision free, the first condition of the theorem in Section 7.1 which requires P to be an invariant for the system $S' = A^N||C_{drop}$, is satisfied.

When a command request is received, the central coordinator must check the second condition which involves computing the reachability of $A^N_{post}||C_{drop}$. This computation uses the reach set for the two-command combination of the current command being applied, and the command being requested. These two-command combination reach sets are

Parameter	Value
$[v_{min}, v_{max}]$	[-2, 5]
$[a_{min}, a_{max}]$ (if $v \geq 0$)	[-5, 2]
$[a_{min}, a_{max}]$ (if $v < 0$)	[-2, 5]
$[\rho_{min}, \rho_{max}]$	[-0.2, 0.2]
$[v_{step,}a_{step}, \rho_{step}]$	[0.5, 0.5, 0.05]

Table 7.1: The design-time computation of reachability using state and input enumeration to produce reachability results which can be used at runtime.

computed offline using input and state enumeration, and then online reachability reduction transformations are applied in order to support any arbitrary combinations of two commands.

As described in Section 7.3.2, the central coordinator will enumerate the state space for each vehicle and possible inputs. For a single command packet, the vehicle behavior is to apply a given acceleration value for one second, and then slow to a stop. While this command is being applied (prior to slowing down), a new command can be computed by the controller and sent to the central coordinator for approval. In this way, control is possible, while safety is still guaranteed if either communication fails or if the new command is not compatible with what the other agents are doing.

As explained in Section 7.3.1, we apply reachability reduction transformations to x, y, and θ so that the enumeration only needs iterate through values of the state variable v for each command of the two-command combination reach set. In terms of input enumeration, in each command request packet, the values of a and ρ are constant and bounded, so we can enumerate those values as well. We therefore precomputed the reachability for all combinations of the current command values v, a, and ρ, and the desired command values a', and ρ', using the parameters given in Table 7.1. These parameters offer a trade off between computation time and accuracy, where using smaller step sizes will yield tighter reach sets, at the cost of longer enumeration. In our case, we adjusted the parameters to be as small as possible while maintaining a tractable computation time (more details in Section 7.4.3). For each combination, a file was produced containing the reach set for that set of parameters, which could be loaded at runtime by the central coordinator as needed.

Each file contained the reach set for a range of values that we were enumerating, each of size $[v_{step,}a_{step}, \rho_{step}]$. To actually compute reachability of a hybrid automaton, a number of different tools exist [95, 96, 97]. We used a modified version of the HyCreate tool [98] to compute the reachability for this system, due to (1) familiarity with the tool, (2) the tool allowing us define ranges for the input values, (3) having the capability to use nonlinear dynamics in each mode which is needed since the derivatives of x and y contain sine and

cosine. Notice that reachable set of states for each command could be reduced by decreasing v_{step}, a_{step}, and ρ_{step}, at the cost of having to do a larger number of computations to cover the state and input space.

7.4.2 Runtime Evaluation

In order to demonstrate effectiveness of proposed approach, we created a Matlab simulation of the system where each vehicle has an initial position and destination. The vehicles each had a controller that generated commands towards getting them to the destination. The overall goal was to prevent collisions between the vehicles while still allowing motion within the shared space.

We performed a set of experiments using the simulator in order to validate the approach and evaluate its scalability. We now describe each of the experiments, their results, and then provide a short analysis about the scalability of the method.

Experiments

For all the experiments, the initial position and final destination of the vehicles was assigned randomly in a 20x20 meter area. For measuring each data point, the simulator was run a hundred times and all target variables were measured at each execution.

The first experiment measured the time required to perform the safety check for a single command request from a vehicle. These values were measured varying the number of vehicles (Figure 7.5).

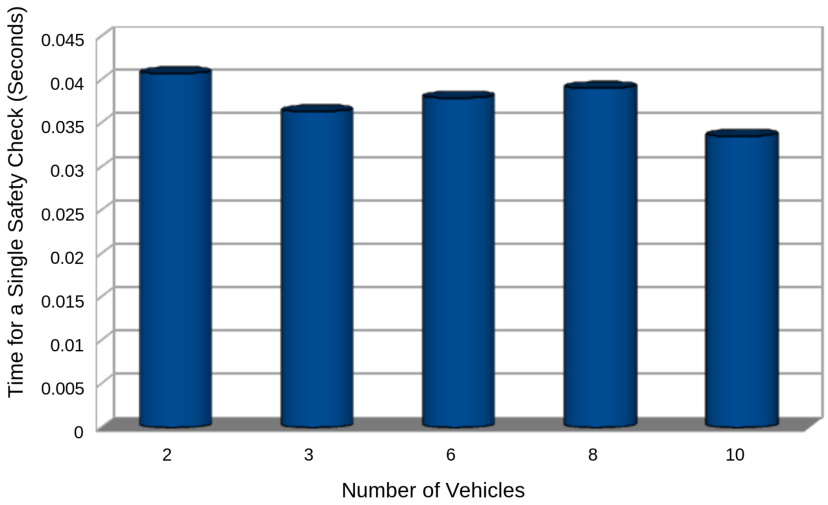


Figure 7.5: The single safety check time is independent of the number of vehicles.

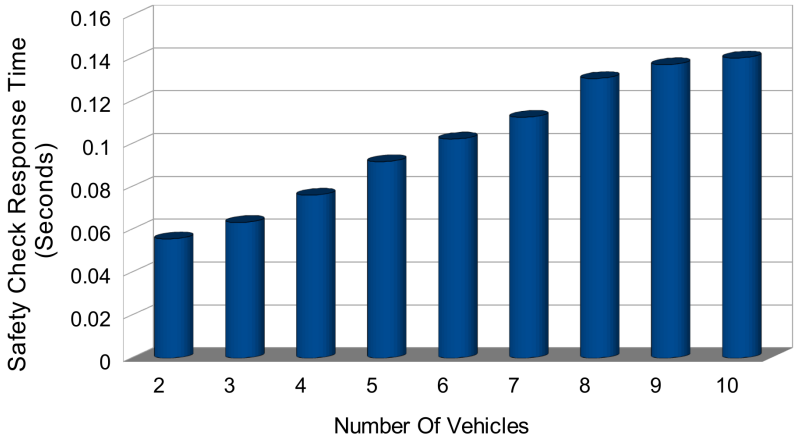


Figure 7.6: The average response time increases linearly in relation to the number of agents in the system.

The measured time for a single safety check execution on the central coordinator is independent from number of vehicles. This is because we did the reachabilty computation offline, and the online check only requires loading the computation from disk and checking for compatibility with existing commands being followed in the system. Although the number of existing commands increases as we have more agents, the check time is dominated by the reachable set load time from disk.

The second experiment measured the average response time for a vehicle's safety check request. Response time is defined as the interval of time between the moment that a command is sent to central coordinator for a safety check until the safety check result (permission granted or not granted) is received by the vehicle. Measurements for this experiment are shown in Figure 7.6.

As the number of vehicles increases, the number of simultaneous requests being processed also increases linearly. As a result, some of the requests get queued in the central coordinator, which is implemented as a single thread in our implementation. This could be slightly parallelized by loading reachability results in separate threads, and then using a mutex to handle updates to the global state once the reach set has been loaded. Nonetheless, due to the contention for shared global state, a worst-case linear performance is expected for this step, although through parallelization we could significantly reduce the constant.

Additionally, the measurements here represent a particularly poor case for the number of vehicles considered. This is because clocks are synchronized in the simulation, and all vehicles have the same command-send frequency. Therefore, all the commands from all the vehicles arrive at once. In practice, we would expect the request times to be not so precisely

synchronized and therefore likely to be spread out more, resulting in a lower experienced delay. The safety result from Section 7.1 is still applicable to this system when messages have varying delays.

In the third experiment, the effect of packet loss in the communication channel was measured against the operation time of the system. We measured the time interval for three vehicles to go the distance between their starting and ending points. Here, the start and end points were fixed for all the experiments. Cases during which vehicles stopped to prevent a collision were discarded from the results (the system could continue to progress after vehicles initially stopped by changing the desired command to one that was compatible with the actions of other vehicles, although for measurements we did not devise such a system). Measured times are presented in Figure 7.7.

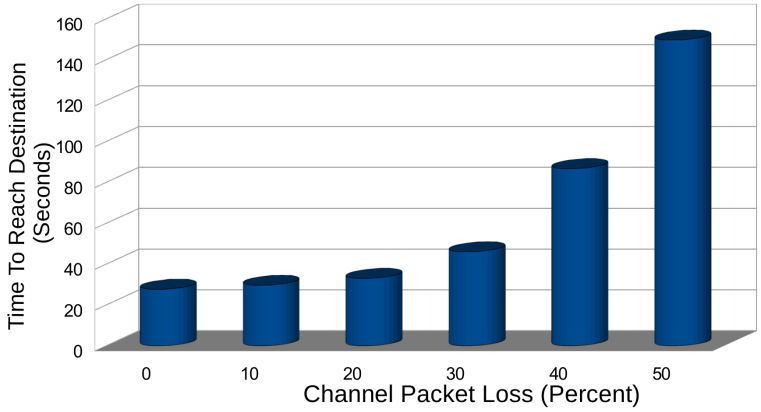


Figure 7.7: As packet loss increases, the vehicles need more time to reach their destination.

In general, increasing packet loss would lead to unnecessary vehicle stops due to the network, rather than to avoid conflicts. These unnecessary stops mean that, as we increase packetloss, the performance of the distributed CPS decreases. However, safety is always guaranteed. This is the trend observed in the figure.

The final experiment validated the safety aspects of the proposed approach and implementation. Here, we created scenarios where a collision would occur if there was no central coordinator. The scenarios we tested are shown in Figures 7.8, 7.9 and 7.10.

In all of tested cases, collisions are prevented by the central coordinator when the vehicles get too close.

However, notice that safety in this implementation comes at the cost of stopping the vehicles when necessary. In an alternative implementation of the proposed approach, if the central coordinator rejected a command, nodes might have sufficient time to send an

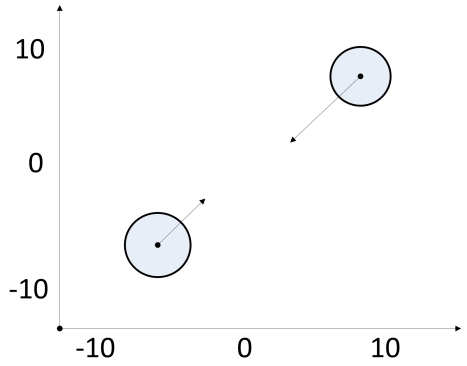


Figure 7.8: Two vehicles

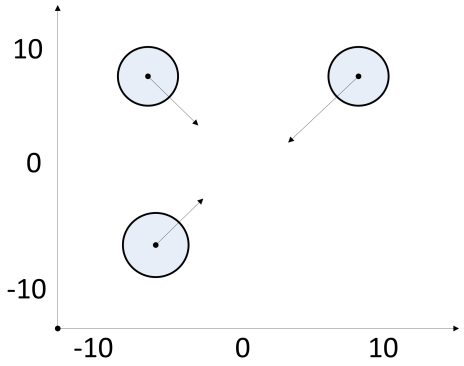


Figure 7.9: Three vehicles

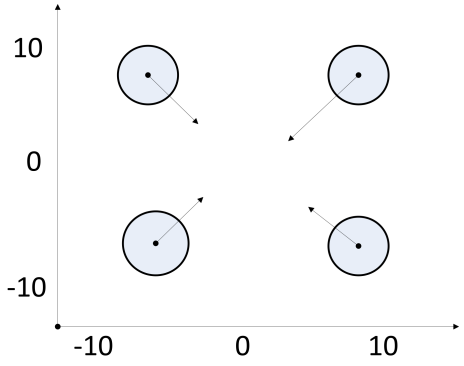


Figure 7.10: Four vehicles

alternate command before having to slow to a stop. Alternatively, we could apply the compatible action chain idea from Section 7.2 if a progress guarantee was needed.

Implementation Scalability

The measurements in the first experiment (average time for a single safety check) are a good indicator of implementation scalability. The control interval of the vehicles, as defined before, is the interval of time during which the vehicle must be granted permission to apply its proposed command, otherwise it will go into the stopping mode as it does not have any safe command to execute next. If the number of vehicles increases such that the total processing time is more than the control interval, some vehicles' response times will go above control interval. Those vehicles would be forced to stop even though their commands could have been safe to execute. In other words, due to lack of enough processing capacity on central coordinator some vehicles would be forced to stop unnecessarily.

As a result of this analysis, we suggest that the number of vehicles (agents) should be smaller than $\frac{T_c}{RTT+T_s}$, where, T_c is the control interval, RTT is the network round trip time

and T_s is the time required to perform a safety check for a single command of a vehicle.

7.4.3 Resource Requirements

The applicability of the method is dependent on the online and offline resources which are necessary to implement the approach.

In terms of offline resources, reachability reduction transformations are applied to the system, and then the input and state enumeration process proceeds to generate reachable sets of states for every configuration. The reachability reduction transformation process is done once manually based on the system dynamics, and serves to reduce the size of the enumeration. The enumeration itself depends on the number of coverings of the sets of states needed, as described in Section 7.3.2.

For the parameters in our case study (Table 7.1), the computation required 175616 individual reachability computations to be done, which took three days using a single system with an Intel i7-2670QM CPU (2.20GHz) with four cores and 12 GB RAM. The final combined size of the files, was 4.96 GB. This computation is an embarrassingly parallel workload (each of the 175616 computations are independent) and we made use of all four cores on the system. It could likely be run on a parallel computing cluster if further reductions in offline computation time were desired.

In terms of online resources, the resource requirements consists of storage size and access speed. On disk, 4.96 GB would be required to store the reachable sets of states. For an embedded system, this could be problematic, but note that these files only need to be located on the central coordinator, not the individual agents. For this reason, we believe the storage requirement is acceptable.

The other key measurement is the access speed to load the reachability result. Each of the 175616 reachability files was in the tens of KB and only a single file needs to be read per check. Based on the experiments performed (Figure 7.5), this load and check time is approximately 40ms, which is acceptable for the time scales we are considering (recall that the vehicles needs to receive a response within a second before beginning to stop).

Other measures of scalability include considering a larger number of vehicles, larger area, or more complicated systems.

Reachability reduction transformations also allow us to compute the reachable sets of states for a single agent, and soundly compose them at runtime (since the agent automata are independent). They also allows us to translate and rotate the resultant reach sets into place as needed. This means that no additional offline computation would be necessary in order to consider more agents or a larger area. The online translation and rotation of the

reach set takes constant time.

For more complicated systems with additional state variables, the scalability will depend upon the scalability of the underlying tool used compute reachability. Some state-of-the-art hybrid automata reachability tools, for example Flow* [96], have been used for systems with up to ten variables with non-linear dynamics. With linear dynamics, SpaceEx [97] has analysed systems including a 28 variable helicopter system, and even systems with up to 200 variables (filter benchmarks). The specifics, of course, depend on the system under consideration and the length of time needed to be analysed. This length of time, in turn, depends on the behavior of the low-level controller. For systems with complex safety actions in the absence of commands, for example, airplanes which cannot stop in place, the reachability tool would need to be able to detect a fixpoint in its computation in order to ensure it has output the entire set of reachable states, which adds complication to the underlying reachability algorithms.

7.5 SUMMARY AND DISCUSSION

In this chapter, we have described an approach to increase the resilience of a cyber-physical system from errors in the high-level control logic. Our approach of monitoring run-time commands in order to maintain a safety invariant is general and powerful but comes at the cost of performing part of the checking at run time. Since this may be impractical to do online, we then went on to show, through a combination of reachability reduction transformations and input and state enumeration, how to perform this operation offline. A case study was created to evaluate the effect of this operation and measurements showed the technique both performed as expected (vehicles did not collide), and scalability could be achieved with this type of framework.

As future work, we intend to investigate more flexible approaches for proving safe progress guarantees. For example, the use of n-way compatible action chains (Section 7.2.2) could be investigated to allow agents to take multiple steps in a compatible action chain before needing to hear back from the central coordinator. Also, issues dealing with agent failure and recovery have not been considered here but would likely need to be investigated for practical use. Additionally, we could consider more complicated notions of safety rather than just invariants, such as temporal logic properties defined using LTL or CTL. Finally, we would like to relax the architectural requirement of a central coordinator and, instead, allow distributed agents to send commands to one another as needed while still maintaining safety and a notion of progress.

CHAPTER 8: FAIL-SAFE DESIGN PATTERNS

In this work thus far, we have discussed a few design patterns that each provide resiliency against various types of fault that may occur in different layers of a CPS. Every design has its limitations on what sorts of guarantees and protections it can provide. There are additional faults that our designs, in their current form, will not be effective against them. In this short chapter, we aim to list some of these categories and briefly review some mitigation strategies. These strategies do not necessarily provide the same level of safety and progress guarantees that we have discussed so far in this work. Instead, they should be viewed as last resort techniques that might limit the damage and possibly aid in recovery of CPS.

When using our restart-based fault-tolerant design (chapters 4 and 5), it is possible that a fault in the complex controller itself or malfunction in other applications could trigger system restarts. In theory, under the guarantees provided by our design, the physical plant will remain safe even under such repeated restarts. However, continued operation under such conditions will put the physical components through constant stress and eventually lead to failure due to overuse.

One possible way to reduce this effect is to limit the number of restarts that can happen within an interval of time and take actions to ensure that restarts rates do not increase beyond the threshold. For instance, a malfunctioning software component such as an application may be the cause of the restarts. One strategy to contain such situations is to disable various non-critical software modules either one-by-one or all at once. While not guaranteed, this might enable the system to continue normal operation. One can employ more sophisticated disabling policies. For instance, system modules could be disabled one-by-one to isolate the root cause. It is even possible to construct a dependency tree between applications and modules and disable/terminate them starting from the leaves of the tree towards the root of the tree trying to limit the impact of disablement on the overall system functionality. This procedure can be even further extended by disabling unnecessary kernel modules and services. If the root cause lies among the components that are necessary for the safety controller to function, the only solution would be to restart the platform whenever the problem occurs.

A system that has implemented the restart-based secure architecture (Chapter 6) can be subject to repeated attacks by an adversary. As we mentioned earlier, restarting the system only recovers the plant from that instance of the attack and it does not fix the vulnerability that allowed the exploit to happen in the first place. In theory, the plant remains safe under repeated attacks; however, in reality, such attacks will cause extreme wear and tear

on the physical plant and will eventually lead to failure. One possible mitigation is to limit the number of times that the secure platform can re-establish network connectivity. After the limit has passed, the secure architecture will completely isolate the system from the external network and operate using the safety controller – this is a fail-safe mode. After a certain amount of time passes, depending on the actual system requirements, network connectivity can be established again. Another approach is to increase the difficulty of performing repeated attacks by diversifying the software after each restart. For instance, one could compile multiple versions of the system software image where, on each version, modules are loaded into different static addresses. After every restart, one image will be picked randomly and loaded into the memory. This would stop the adversary from using the same parameters of the previous attack in the case of attacks that rely on static addresses *e.g.,* buffer overflow attacks. Additionally, diversifying other parameters of the system (such as the assignment of port numbers) after each restart can further increase the complexity of launching attacks on the system.

Another category of faults that our designs did not cover is that of physical faults. A physical fault refers to a situation in which the behavior of the physical components, and consequently the plant dynamics, significantly deviates from the original model that was used to design the controller. In a passive fault-tolerant control system, deviations of the plant parameters from their true values or deviations of the actuators from their expected position may be effectively compensated by a fixed robust feedback controller. However, if these deviations become excessively large and exceed the robustness bound, actions need to be taken. Our restart-based techniques rely heavily on the physical dynamics of the system in the process of constructing the restart-tolerant controller. If the physical plant is not compatible with the model, the base controller may not be able to stabilize the plant and the safety guarantees will not stand any longer.

There is a series of work based on robust fault-tolerant control that provide high levels of performance and robustness in the presence of physical failures. Examples of such work include input-to state stable control [99], disturbance-observer control [100] and internal model-based control [101]. There is also work that relies on L1 adaptive control theory to accommodate the deviations of the physical system. Wang *et al.* [102] propose a new simplex based architecture that can adapt to the physical failures and will guarantee safety under such failures. The authors suggest a monitoring system to detect physical failures and examine the scale of the uncertainty in the system caused by them. They use adaptive control theory to design a high-assurance (safety) controller that can stabilize the plant in spite of the deviations.

Faults in sensors and actuators is another category that I did not address in my designs.

All techniques discussed in prior chapters assume that the sensors operate correctly, i.e., they measure the target property within a reasonable noise level on their specifications. Without access to the physical properties of the plant, in our designs, the decision module and safety controller cannot maintain the safety. There is a large body of work on sensor and actuator faults in the CPS. A recent study [103] proposed a complete error detection, fault diagnosis and system recovery architecture for a coaxial octorotor. Some researchers have developed an analytical redundancy-based approach to detecting and isolating sensor, actuator and component (i.e., plant) faults in dynamical systems [104, 105]. Detecting simultaneous actuator and sensor faults is a difficult problem. Hajiyev *et al.* [106] proposed a method based on two types of Kalman filters: a conventional linear Kalman filter that estimates the states of the plant and a two-stage Kalman filter that estimates the loss of effectiveness and faults in actuators. Redundancy of sensors and actuators plays a crucial role in the fault-tolerance level of these systems. Once a faulty sensor is detected, it is ignored if the system can continue operating without it (for instance if other sensors measure the same property directly or indirectly) or the measured physical property is estimated using the readings of other available sensors. Much of the work in this area also provides a tolerance bond on the maximum number of simultaneous failures that can be tolerated by a system.

Radiation-induced transient faults are another category that was not in the scope of this work. This type of fault is usually caused by an environmental factor such as radiation or electromagnetic field. One way to mitigate this type of fault is to take advantage of hardware redundancy. In such cases, multiple processing units, preferably of different types, run the same computational workload in parallel. Redundancy significantly reduces the probability of being affected by transient faults. Any of the units impacted by a transient fault will be detected due to the mismatch and the correct output can be fed into the next critical layer [107, 108]. Another conceptually similar approach is to use the idea of software logical redundancy where each task is executed multiple times on the same hardware platform. This requires the additional execution load to be taken into account beforehand and make sure that all replicas of the task finish before their deadline. For the proposed designs presented in this work, we can utilize logical redundancy for critical components such as decision module or safety controller to mitigate the effect of transient faults. Many of the studies in this area also calculate a bound on the maximum number of transient faults that can be tolerated within a time window by their specific design [109].

Handling each fault category poses new challenges and to mitigate them requires extra software or hardware components that eventually translate into higher costs. System designers need to carefully analyze the expected operational environment, the necessary level of fault-resiliency and their budget and design the system accordingly.

CHAPTER 9: CONCLUDING REMARKS

In the modern complex systems, faults are a norm rather than an abnormality and safety-critical systems need to be designed with faults in mind. In this dissertation, I presented multiple low-cost techniques to make use of full system restarts and create safe-by-construct designs that guarantee the safety of CPS.

There exist many problems that can be further studied to improve our designs. Restart time of the platform is a bottleneck in many of our approaches. One future direction of research is creating a multi-stage booting solution for multi-core platforms to mitigate this problem. Our possible idea is to boot one core with the bare minimum requirements to execute the SC in the quickest possible time. The SC can keep the system safe, while the real-time or general purpose OS boots on the other cores. Once the boot process is complete, the control switches to the controllers running on the OS.

Another challenge is the loss of state in case of full system restarts. TEE based solutions that we provided in Chapter 6, significantly mitigate this problem but do not eliminate it. Even with TEE, the restarted system lacks a reliable method to resume its mission in a meaningful way. One could look into means of protecting the state in the presence of an attacker or enabling the capability to detect if the data is corrupted or not.

Another problem that can be further investigated is how to enable concurrent execution of SEIs and normal-world applications in platforms with TEE capabilities. This will allow the normal world tasks to execute with the interruption of SEI, and consequently, the impact of adding security features to the platform will be negligible. The primary challenge would be to use only commercial-off-the-shelf platforms without additional hardware customizations

REFERENCES

[1] S. M. Sulaman, A. Orucevic-Alagic, M. Borg, K. Wnuk, M. Höst, and J. L. de la Vara, "Development of safety-critical software systems using open source software–a systematic map," in *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, 2014, pp. 17–24.

[2] L. Sha, "Using simplicity to control complexity," *IEEE Software*, vol. 18, no. 4, pp. 20–28, Jul 2001.

[3] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*. IEEE, 1999.

[4] S. Baruah, "The limited-preemption uniprocessor scheduling of sporadic task systems," in *17th Euromicro Conference on Real-Time Systems (ECRTS'05)*, July 2005, pp. 137–144.

[5] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, "Implementing embedded security on dual-virtual-cpu systems," *IEEE Design Test of Computers*, vol. 24, no. 6, pp. 582–591, Nov 2007.

[6] Intel Corp, "Intel trusted execution technology," https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/trusted-execution-technology-security-paper.pdf, 2018, accessed: July 2018.

[7] G. Candea, J. Cutler, and A. Fox, "Improving availability with recursive microreboots: A soft-state system case study," *Perform. Eval.*, vol. 56, no. 1-4, pp. 213–248, Mar. 2004. [Online]. Available: http://dx.doi.org/10.1016/j.peva.2003.07.007

[8] G. Candea and A. Fox, "Recursive restartability: Turning the reboot sledgehammer into a scalpel," in *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on*. IEEE, 2001, pp. 125–130.

[9] G. Candea and A. Fox, "Crash-only software," in *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems*, 2003, pp. 67–72.

[10] G. Candea, E. Kiciman, S. Zhang, P. Keyani, and A. Fox, "Jagr: An autonomous self-recovering application server," in *Autonomic Computing Workshop. 2003. Proceedings of the*. IEEE, 2003, pp. 168–177.

[11] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot- a technique for cheap recovery," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04, 2004, pp. 31–44.

[12] K. Vaidyanathan and K. S. Trivedi, "A comprehensive model for software rejuvenation," *Dependable and Secure Computing, IEEE Transactions on*, vol. 2, no. 2, pp. 124–137, 2005.

[13] S. Garg, A. Puliafito, M. Telek, and K. S. Trivedi, "Analysis of software rejuvenation using markov regenerative stochastic petri net," in *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on.* IEEE, 1995, pp. 180–187.

[14] Y. Huang, C. Kintala, N. Kolettis, and N. D. Fulton, "Software rejuvenation: Analysis, module and applications," in *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on.* IEEE, 1995, pp. 381–390.

[15] F. Abdi, R. Mancuso, S. Bak, O. Dantsker, and M. Caccamo, "Reset-based recovery for real-time cyber-physical systems with temporal safety constraints," in *IEEE 21st Conference on Emerging Technologies Factory Automation (ETFA 2016)*, 2016.

[16] L. Sha, "Dependable system upgrade," in *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE.* IEEE, 1998, pp. 440–448.

[17] L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, vol. 1. IEEE, 1996, pp. 335–346.

[18] D. Seto and L. Sha, "An engineering method for safety region development," 1999.

[19] T. L. Crenshaw, E. Gunter, C. L. Robinson, L. Sha, and P. Kumar, "The simplex reference model: Limiting fault-propagation due to unreliable components in cyber-physical system architectures," in *Real-Time Systems Symposium, 2007. RTSS 2007. 28th IEEE International.* IEEE, 2007, pp. 400–412.

[20] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE.* IEEE, 2009, pp. 99–107.

[21] S. Mohan, S. Bak, E. Betti, H. Yun, L. Sha, and M. Caccamo, "S3a: Secure system simplex architecture for enhanced security and robustness of cyber-physical systems," in *Proceedings of the 2nd ACM international conference on High confidence networked systems.* ACM, 2013, pp. 65–74.

[22] P. Vivekanandan, G. Garcia, H. Yun, and S. Keshmiri, "A simplex architecture for intelligent and safe unmanned aerial vehicles," in *2016 IEEE 22nd International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA).* IEEE, Aug 2016, pp. 69–75.

[23] F. Liberato, R. Melhem, and D. Mosse, "Tolerance to multiple transient faults for aperiodic tasks in hard real-time systems," *IEEE Transactions on Computers*, vol. 49, no. 9, pp. 906–914, Sep 2000.

[24] S. Punnekkat, A. Burns, and R. Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, 2001.

[25] G. Lima and A. Burns, *Scheduling Fixed-Priority Hard Real-Time Tasks in the Presence of Faults*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 154–173.

[26] R. M. Pathan and J. Jonsson, "Exact fault-tolerant feasibility analysis of fixed-priority real-time tasks," in *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, Aug 2010, pp. 265–274.

[27] C.-C. Han, K. G. Shin, and J. Wu, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Transactions on Computers*, vol. 52, no. 3, pp. 362–372, March 2003.

[28] M. Pandya and M. Malek, "Minimum achievable utilization for fault-tolerant processing of periodic tasks," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1102–1112, Oct 1998.

[29] M. A. Haque, H. Aydin, and D. Zhu, "Real-time scheduling under fault bursts with multiple recovery strategy," in *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, April 2014.

[30] S. Bak, T. T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *Real-Time Systems Symposium (RTSS), 2014 IEEE*. IEEE, 2014, pp. 138–148.

[31] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandboxing controllers for cyber-physical systems," in *Proceedings of the 2011 IEEE/ACM Second International Conference on Cyber-Physical Systems*, ser. ICCPS '11. Washington, DC, USA: IEEE Computer Society, 2011. [Online]. Available: http://dx.doi.org/10.1109/ICCPS.2011.25 pp. 3–12.

[32] F. Abdi, R. Tabish, M. Rungger, M. Zamani, and M. Caccamo, "Application and system-level software fault tolerance through full system restarts," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*, ser. ICCPS '17. New York, NY, USA: ACM, 2017. [Online]. Available: http://doi.acm.org/10.1145/3055004.3055012 pp. 197–206.

[33] M.-K. Yoon, B. Liu, N. Hovakimyan, and L. Sha, "Virtualdrone: virtual sensing, actuation, and communication for attack-resilient unmanned aerial systems," in *Proceedings of the 8th International Conference on Cyber-Physical Systems*. ACM, 2017, pp. 143–154.

[34] F. Abdi, M. Hasan, S. Mohan, D. Agarwal, and M. Caccamo, "ReSecure: A restart-based security protocol for tightly actuated hard real-time systems," in *IEEE CERTS*, 2016, pp. 47–54.

[35] M.-K. Yoon, S. Mohan, J. Choi, J.-E. Kim, and L. Sha, "Securecore: A multicore-based intrusion detection architecture for real-time embedded systems," in *Real-Time and Embedded Technology and Applications Symposium (RTAS), 2013 IEEE 19th.* IEEE, 2013, pp. 21–32.

[36] N. Santos, K. P. Gummadi, and R. Rodrigues, "Towards trusted cloud computing." *HotCloud*, vol. 9, no. 9, p. 3, 2009.

[37] R. Perez, R. Sailer, L. van Doorn et al., "vtpm: virtualizing the trusted platform module," in *Proc. 15th Conf. on USENIX Security Symposium*, 2006, pp. 305–320.

[38] R. Sailer, T. Jaeger, E. Valdez, R. Caceres, R. Perez, S. Berger, J. L. Griffin, and L. Van Doorn, "Building a mac-based security architecture for the xen open-source hypervisor," in *null.* IEEE, 2005, pp. 276–285.

[39] P. Wilson, A. Frey, T. Mihm, D. Kershaw, and T. Alves, "Implementing embedded security on dual-virtual-cpu systems," *IEEE Design & Test of Computers*, vol. 24, no. 6, 2007.

[40] J. Winter, "Trusted computing building blocks for embedded linux-based arm trustzone platforms," in *Proceedings of the 3rd ACM workshop on Scalable trusted computing.* ACM, 2008, pp. 21–30.

[41] X. Ge, H. Vijayakumar, and T. Jaeger, "Sprobes: Enforcing kernel code integrity on the trustzone architecture," *arXiv preprint arXiv:1410.7747*, 2014.

[42] A. M. Azab, P. Ning, J. Shah, Q. Chen, R. Bhutkar, G. Ganesh, J. Ma, and W. Shen, "Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security.* ACM, 2014, pp. 90–102.

[43] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comput. Syst.*, vol. 20, no. 4, pp. 398–461, Nov. 2002. [Online]. Available: http://doi.acm.org/10.1145/571637.571640

[44] P. E. Veríssimo, N. F. Neves, and M. P. Correia, "Intrusion-tolerant architectures: Concepts and design," in *Architecting Dependable Systems.* Springer Berlin Heidelberg, 2003, pp. 3–36.

[45] P. Veríssimo, "Future directions in distributed computing," A. Schiper, A. A. Shvartsman, H. Weatherspoon, and B. Y. Zhao, Eds. Berlin, Heidelberg: Springer-Verlag, 2003, ch. Uncertainty and Predictability: Can They Be Reconciled?, pp. 108–113. [Online]. Available: http://dl.acm.org/citation.cfm?id=1809315.1809338

[46] P. Sousa, N. F. Neves, and P. Veríssimo, "Proactive resilience through architectural hybridization," in *Proceedings of the 2006 ACM Symposium on Applied Computing*, ser. SAC '06. New York, NY, USA: ACM, 2006. [Online]. Available: http://doi.acm.org/10.1145/1141277.1141435 pp. 686–690.

[47] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Highly available intrusion-tolerant services with proactive-reactive recovery," *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 4, pp. 452–465, April 2010.

[48] J. Song, S. Han, A. Mok, D. Chen, M. Lucas, M. Nixon, and W. Pratt, "Wirelesshart: Applying wireless technology in real-time industrial process control," in *RTAS '08: Proceedings of the 2008 IEEE Real-Time and Embedded Technology and Applications Symposium*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 377–386.

[49] Honeywell, "Onewireless network - isa100.11a-compliant wireless mesh network," https://www.honeywellprocess.com/en-US/explore/products/wireless/OneWireless-Network/pages/default.aspx, 2012.

[50] J. Yao, X. Liu, G. Zhu, and L. Sha, "Netsimplex: Controller fault tolerance architecture in networked control systems," *Industrial Informatics, IEEE Transactions on*, vol. PP, no. 99, p. 1, 2012.

[51] C. Kim, M. Sun, S. Mohan, H. Yun, L. Sha, and T. F. Abdelzaher, "A framework for the safe interoperability of medical devices in the presence of network failures," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ser. ICCPS '10. New York, NY, USA: ACM, 2010, pp. 149–158.

[52] L. Bu, Q. Wang, X. Chen, L. Wang, T. Zhang, J. Zhao, and X. Li, "Toward online hybrid systems model checking of cyber-physical systems' time-bounded short-run behavior," *SIGBED Rev.*, vol. 8, no. 2, pp. 7–10, June 2011.

[53] S. Bak, T. Johnson, M. Caccamo, and L. Sha, "Real-time reachability for verified simplex design," in *Real-Time Systems Symposium (RTSS), 2014 IEEE 35th*, 2014.

[54] J. N. Tsitsiklis, "On the stability of asynchronous iterative processes," *Mathematical systems theory*, vol. 20, no. 1, pp. 137–153, Dec 1987. [Online]. Available: https://doi.org/10.1007/BF01692062

[55] K. M. Chandy, S. Mitra, and C. Pilotto, "Convergence verification: From shared memory to partially synchronous systems," in *Formal Modeling and Analysis of Timed Systems*, F. Cassez and C. Jard, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 218–232.

[56] A. K. Agogino and K. Tumer, "A multiagent approach to managing air traffic flow," *Autonomous Agents and Multi-Agent Systems*, vol. 24, no. 1, pp. 1–25, Jan 2012. [Online]. Available: https://doi.org/10.1007/s10458-010-9142-5

[57] D. Seto and L. Sha, "A case study on analytical analysis of the inverted pendulum real-time control system," DTIC Document, Tech. Rep., 1999.

[58] D. Seto, E. Ferreira, and T. F. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," DTIC Document, Tech. Rep., 2000.

[59] E. D. Sontag, *Mathematical control theory: deterministic finite dimensional systems*. Springer Science & Business Media, 2013, vol. 6.

[60] "PCA9685: 16-channel, 12-bit PWM Fm+ I2C-bus LED controller," https://goo.gl/FMnOQT, 2016, accessed: Oct. 2016.

[61] G. Reißig, A. Weber, and M. Rungger, "Feedback refinement relations for the synthesis of symbolic controllers," *IEEE TAC*, vol. 62, 2017.

[62] M. Althoff and B. H. Krogh, "Reachability analysis of nonlinear differential-algebraic systems," *IEEE Transactions on Automatic Control*, vol. 59, no. 2, pp. 371–383, Feb 2014.

[63] E. Asarin, T. Dang, and A. Girard, "Reachability analysis of nonlinear systems using conservative approximation," in *International Workshop on Hybrid Systems: Computation and Control*. Springer, 2003, pp. 20–35.

[64] P. Tabuada, *Verification and control of hybrid systems: a symbolic approach*. Springer Science & Business Media, 2009.

[65] C. Baier and J. P. Katoen, *Principles of model checking*. MIT press, 2008.

[66] G. Reissig, "Computing abstractions of nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 56, no. 11, pp. 2583–2598, Nov 2011.

[67] M. Rungger and M. Zamani, "Scots: A tool for the synthesis of symbolic controllers," in *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control*. ACM, 2016, pp. 99–104.

[68] M. Zamani, I. Tkachev, and A. Abate, "Towards scalable synthesis of stochastic control systems," *Discrete Event Dynamic Systems*, vol. 27, no. 2, pp. 341–369, 2017.

[69] M. Zamani and M. Arcak, "Compositional abstraction for networks of control systems: A dissipativity approach," *IEEE Transactions on Control of Network Systems*, vol. PP, no. 99, pp. 1–1, 2017.

[70] F. Blanchini and S. Miani, *Set-theoretic methods in control*. Springer, 2008, pp. 156–163.

[71] M. Rungger and P. Tabuada, "Computing robust controlled invariant sets of linear systems," *CoRR*, vol. abs/, 2016. [Online]. Available: http://arxiv.org/abs/1601.00416

[72] Q. Inc., "3 dof helicopter."

[73] "FreeRTOS ," http://www.freertos.org, 2016, accessed: Sep. 2016.

[74] https://github.com/abditag2/reset-based-recovery, 2017.

[75] Quanser Inc., "Q8 data acquisition board," http://www.quanser.com/products/q8, 2016, accessed: September 2016.

[76] A. Inc., "Arm trustzone," https://www.arm.com/products/security-on-arm/trustzone, 2016, accessed: September 2016.

[77] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, vol. 20, no. 1, pp. 46–61, 1973.

[78] R. I. Davis, A. Burns, R. J. Bril, and J. J. Lukkien, "Controller area network (can) schedulability analysis: Refuted, revisited and revised," *Real-Time Systems*, vol. 35, no. 3, pp. 239–272, 2007.

[79] R. J. Bril, J. J. Lukkien, and W. F. J. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption revisited," in *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, July 2007, pp. 269–279.

[80] G. C. Buttazzo, M. Bertogna, and G. Yao, "Limited preemptive scheduling for real-time systems. a survey," *IEEE Transactions on Industrial Informatics*, vol. 9, no. 1, pp. 3–15, Feb 2013.

[81] F. Abdi, R. Mancuso, R. Tabish, and M. Caccamo, "Achieving system-level fault-tolerance with controlled resets," University of Illinois at Urbana-Champaign, Tech. Rep., April 2017. [Online]. Available: http://rtsl-edge.cs.illinois.edu/reset-based/reset_sched.pdf

[82] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *2008 IEEE Symposium on Security and Privacy (sp 2008)*, May 2008, pp. 129–142.

[83] K. Koscher, A. Czeskis, F. Roesner, S. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham et al., "Experimental security analysis of a modern automobile," in *IEEE Symposium on Security and Privacy*. IEEE, 2010, pp. 447–462.

[84] S. Pinto, J. Pereira, T. Gomes, A. Tavares, and J. Cabral, "LTZVisor: TrustZone is the Key," in *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), M. Bertogna, Ed., vol. 76. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017. [Online]. Available: http://drops.dagstuhl.de/opus/volltexte/2017/7153 pp. 4:1–4:22.

[85] Quanser Inc., "3-DOF helicopter reference manual," document Number 644, Revision 2.1.

[86] S. H. Trapnes, "Optimal temperature control of rooms for minimum energy cost," M.S. thesis, Institutt for kjemisk prosessteknologi, Norway, 2013.

[87] AVNET, "Zedboard hardware user's guide," http://zedboard.org/sites/default/files/documentations/ZedBoard_HW_UG_v2_2.pdf, accessed: Apr. 2017.

[88] https://github.com/emsoft2017restart/restart-based-framework-demo, 2017.

[89] Texas Instruments, "Msp-exp430g2 launchpad development kit," http://www.ti.com/lit/ug/slau318g/slau318g.pdf, 2016, accessed: April 2017.

[90] Make Linux, "Super fast boot of embedded linux," http://www.makelinux.com/emb/fastboot/omap, 2017, accessed: June 2017.

[91] J. Turek and D. Shasha, "The many faces of consensus in distributed systems," *Computer*, vol. 25, no. 6, pp. 8–17, June 1992.

[92] S. Bak, D. K. Chivukula, O. Adekunle, M. Sun, M. Caccamo, and L. Sha, "The system-level simplex architecture for improved real-time embedded system safety," in *RTAS '09: Proceedings of the 2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, 2009.

[93] S. Mitra, "A verification framework for hybrid systems," Ph.D. dissertation, Massachusetts Institute of Technology, 2007.

[94] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager, *The Theory of Timed I/O Automata (Synthesis Lectures in Computer Science)*. Morgan & Claypool Publishers, 2006.

[95] G. Frehse, "Phaver: Algorithmic verification of hybrid systems past hytech." Springer, 2005, pp. 258–273.

[96] X. Chen, E. Abraham, and S. Sankaranarayanan, "Taylor model flowpipe construction for non-linear hybrid systems," in *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, 2012, pp. 183–192.

[97] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Proc. 23rd International Conference on Computer Aided Verification (CAV)*, ser. LNCS, S. Q. Ganesh Gopalakrishnan, Ed. Springer, 2011.

[98] S. Bak and M. Caccamo, "Computing reachability for nonlinear systems with hycreate," in *Demo and Poster Session, ACM/IEEE 16th International Conference on Hybrid Systems*, 2013.

[99] E. D. Sontag and Y. Wang, "On characterizations of the input-to-state stability property," *Systems and Control Journal*, vol. 24, no. 5, pp. 351–359, 1995.

[100] H. Shim and N. H. Jo, "An almost necessary and sufficient condition for robust stability of closed-loop systems with disturbance observer," *Automatica*, vol. 45, no. 1, pp. 296 – 299, 2009. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0005109808003749

[101] L. Harnefors and H. . Nee, "Model-based current control of ac machines using the internal model control method," *IEEE Transactions on Industry Applications*, vol. 34, no. 1, pp. 133–141, Jan 1998.

[102] X. Wang, N. Hovakimyan, and L. Sha, "L1simplex: fault-tolerant control of cyber-physical systems," in *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*. ACM, 2013, pp. 41–50.

[103] M. Saied, B. Lussier, I. Fantoni, H. Shraim, and C. Francis, "Fault diagnosis and fault-tolerant control of an octorotor uav using motors speeds measurements," *IFAC-PapersOnLine*, vol. 50, no. 1, pp. 5263–5268, 2017.

[104] E. C. Larson, B. E. Parker, and B. R. Clark, "Model-based sensor and actuator fault detection and isolation," in *Proceedings of the 2002 American Control Conference (IEEE Cat. No. CH37301)*, vol. 5. IEEE, 2002, pp. 4215–4219.

[105] J. Lee and J. Lyou, "Fault diagnosis and fault tolerant control of linear stochastic systems with unknown inputs." *Systems Science*, vol. 27, no. 3, pp. 59–76, 2001.

[106] C. M. Hajiyevt and F. Caliskan, "Integrated sensor/actuator fdi and reconfigurable control for fault-tolerant flight control system design," *The Aeronautical Journal*, vol. 105, no. 1051, pp. 525–533, 2001.

[107] A. Damm, J. Reisinger, W. Schwabl, and H. Kopetz, "The real-time operating system of mars," *ACM SIGOPS Operating Systems Review*, vol. 23, no. 3, pp. 141–157, 1989.

[108] G. K. Saha, "Approaches to software based fault tolerance–a review," *Computer Science*, vol. 13, no. 3, p. 39, 2005.

[109] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors, "Plr: A software approach to transient fault tolerance for multicore architectures," *IEEE Transactions on Dependable and Secure Computing*, vol. 6, no. 2, pp. 135–148, 2009.