Application and System-Level Software Fault Tolerance Through Full System Restarts

Fardin Abdi
Dept. of Computer Science, University of Illinois at Urbana-Champaign
abditag2@illinois.edu

Rohan Tabish
Dept. of Computer Science, University of Illinois at Urbana-Champaign
rtabish@illinois.edu

Matthias Rungger
Dept. of Electrical and Computer Engineering, Technical University of Munich, Germany
matthias.rungger@tum.de

Majid Zamani
Dept. of Electrical and Computer Engineering, Technical University of Munich, Germany
zamani@tum.de

Marco Caccamo
Dept. of Computer Science, University of Illinois at Urbana-Champaign
mcaccmo@illinois.edu

ABSTRACT

Due to the growing performance requirements, embedded systems are increasingly more complex. Meanwhile, they are also expected to be reliable. Guaranteeing reliability on complex systems is very challenging. Consequently, there is a substantial need for designs that enable the use of unverified components such as real-time operating system (RTOS) without requiring their correctness to guarantee safety.

In this work, we propose a novel approach to design a controller that enables the system to restart and remain safe during and after the restart. Complementing this controller with a switching logic allows the system to use complex, unverified controller to drive the system as long as it does not jeopardize safety. Such a design also tolerates faults that occur in the underlying software layers such as RTOS and middleware and recovers from them through system-level restarts that reinitialize the software (middleware, RTOS, and applications) from a read-only storage. Our approach is implementable using one commercial off-the-shelf (COTS) processing unit. To demonstrate the efficacy of our solution, we fully implement a controller for a 3 degree of freedom (3DOF) helicopter. We test the system by injecting various types of faults into the applications and RTOS and verify that the system remains safe.

CCS CONCEPTS

•Computer systems organization →Embedded software; Reliability;

KEYWORDS

Cyber-Physical Systems, Fault-Tolerance, Fault-Recovery, Runtime Restart, Embedded Systems, Reliability

ACM Reference format:
Fardin Abdi, Rohan Tabish, Matthias Rungger, Majid Zamani, and Marco Caccamo. 2017. Application and System-Level Software Fault Tolerance Through Full System Restarts. In *Proceedings of The 8th ACM/IEEE International Conference on Cyber-Physical Systems, Pittsburgh, PA USA, April 2017 (ICCPS'17),* 10 pages.
DOI: http://dx.doi.org/10.1145/3055004.3055012

1 INTRODUCTION

With the increased use of embedded systems in various safety-critical environments, these systems are expected to provide both reliability and high-performance. On the one hand, delivering high-performance drives the need for more complex systems[1]. On the other hand, the high complexity increases the possibilities for errors and makes formal verification more challenging.

In these complex control systems where one or more control applications run on top of an RTOS and share the resources, two root causes may lead to safety violations[2]. First, the control application may issue a set of unsafe commands due to the incorrect logic (bugs) or fail to generate any commands at all (*i.e.,* application-level faults). Second, even with a bug-free control application, faults in underlying software layers such as the RTOS can disrupt the execution of the controller and jeopardize the safety (*i.e.,* system-level faults). Ideally, all the components of these systems including the RTOS must be formally verified to ensure they are fault-free. Due to the high complexity, formal verification of the entire platform is tough. Therefore, designing platforms and architectures that enable the system designers to utilize components such as RTOS and vendor drivers without requiring to prove their correctness to guarantee safety is important.

In this work, a new approach is proposed to provide *fault-tolerance* and *liveliness* in the presence of application-level faults (faults in application logic and its implementation) as well as system-level faults (faults in the underlying software such as RTOS and middleware) *using only one COTS processing unit.* This paper explains the procedure to design a controller that enables the entire computing

[1]Such systems usually use the operating system (OS) primitives to perform I/O or to set up concurrent threads [19], utilize vendor developed drivers and use open source libraries [27].
[2]In this paper, safety means not to violate the constraints of the physical components.

system to be safely restarted at runtime. This controller can keep the system inside a subset of safety region, only by updating the actuator input at least once after every system restart. In other words, after a restart, system can restart again after only one command is applied to the actuators. In this paper, this controller is called 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. During a restart, a fresh image of all the software (middleware, RTOS, and applications) is loaded from a read-only storage which 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 [9–12, 14, 15, 29]. Alongside, partial restarting of safety-critical systems using extra hardware was investigated in [4, 5]. To the best of our knowledge, this is the first work that proposes to restart the entire system in a safety-critical environment.

Having only BC and the WD mechanism which enables restarting, allows the system to remain safe, tolerate faults and recover from them. However, it does not make any progress towards its mission goal. To address this issue, BC is complemented with a Mission controller (MC) and a Decision Module (DM). 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. The logical view of this design is depicted in Figure 1.

In the proposed design, the only components that need to be verified for correct functionality are BC, DM and Flushing Task. Any faults 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 time. In summary, this design enables the system to provide formal safety guarantees by verifying only the correctness of BC, DM and Flushing Task instead of entire MC, RTOS, and middleware.

The key contributions of our work are:

- Introducing system-wide restart and software reload from read-only storage as a recovery method for embedded systems with safety-critical constraints.
- 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 a 3DOF helicopter.

The rest of this paper is organized in the following way. We discuss the related works in Section 2 and compare them to our design. In Section 3 we formalize the safety definition of a system. In Section 4 we describe our design and the safety conditions in decision module. In Section 5 we explain how to construct BC. Then we apply our approach to creating a controller for a 3DOF helicopter and verify its robustness with fault-injection testing in Section 6. In Section 7, we discuss the limitations of our proposed approach. Section 8 concludes the paper.

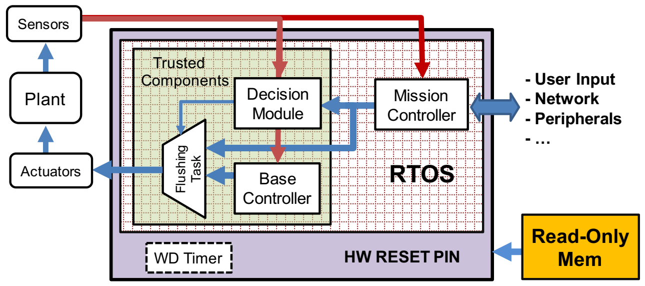

Figure 1: The logical view of the proposed design.

2 RELATED WORK

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 [24–26]. In earlier simplex designs, fault tolerance was achieved in one of two ways. In some of these designs such as [13, 23–26], 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, there is no guarantee of the correct behavior in the presence of system-level faults. Our proposed design protects the system from both application-level and system-level faults.

Some Simplex-based designs such as System-level Simplex [5], S3A[20] and other variants [30] 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 is 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 approach proposed in this paper needs only one processor and tolerates system-level faults.

Note that, the control domain of the proposed BC depends on the system dynamics and the restart time of the platform. Increased restart time, shrinks the domain size. And, for a given system, it may be empty; meaning that the dynamics of the system does not allow a controller with such properties to exist. System-Level Simplex does not have this limitation, because it uses a dedicated hardware that is not impacted by faults (or restarts) in the complex controller unit. Proposed approach is especially suitable for Internet of Things (IoT) applications, requiring increased robustness at low cost and without using extra hardware as System-Level Simplex requires.

Notion of restarting as a means of recovery from faults and improving system availability is previously proposed in literature. 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 [8]. Our approach, as described in this paper, fits in the former category. However, with slight modification, our design can incorporate periodic self-triggered restarts to prevent future unscheduled unavailable times. In the second form, this work can also be categorized in the latter category.

Most of the previous works on restarting are proposed for traditional *non* safety-critical computing systems such as servers and switches. Authors in [9] introduce recursively restartable systems as a design paradigm for highly available systems and uses a combination of revival and rejuvenation techniques. Earlier literature [10–12] 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 [14, 15, 29] 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 [4] was an attempt to utilize restarting as a recovery method for computing systems in safety-critical environments. In [4], we used System-Level Simplex architecture and proposed to restart only the complex subsystem upon occurrence of faults. This was feasible, because the safety subsystem ran on a dedicated hardware unit and was not impacted by the restarts in the complex subsystem. The approach of the current paper is significantly different and uses only only one hardware unit.

3 CONTROL SYSTEM DESCRIPTION

Consider a linear control system given by

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu(t) \\ y(t) = Cx(t), \end{cases} \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the state vector, $u(t) \in \mathbb{R}^i$ is the inputs, and matrices A, B and C define the dynamics of the system and are of the appropriate dimensions.

3.1 Formulating Safety

Physical systems have limits and constraints that need to be respected. For example, a UAV must take the sharp turns only when its velocity is under a safe threshold otherwise the wings can get damaged. In this paper, we define safety region $\mathcal{S}$ as a subset of the state space where all those physical constraints are respected. The system is considered safe as long as it stays inside $\mathcal{S}$. We express the safety region through linear inequalities in the following form:

$$\mathcal{S} = \{x \mid H_x \cdot x \leq h_x\}. \tag{2}$$

Here, $\mathcal{S} \subseteq \mathbb{R}^n$ is called the safety region. The state of the system must always evolve inside $\mathcal{S}$.

Additionally, physical actuators also have limits on their operational ranges which can be similarly expressed by

$$\mathcal{S}_u = \{u \mid H_u \cdot u \leq h_u\}. \tag{3}$$

Here $\mathcal{S}_u \subseteq \mathbb{R}^i$.

A controller guarantees safety if it ensures that the system remains inside $\mathcal{S}$ using only the control commands in $\mathcal{S}_u$.

3.2 Reachable Set

A state x_f is called a reachable state from x_0 within δt time, if for any given t_0, there exist a control input $u(t) = u_0, t \in [t_0, t_0 + \delta t]$ and $u_0 \in \mathcal{S}_u$ such that $x(t_0 + \delta t) = x_f$. We use the following notation for reachable set; $Reach_{\leq\tau}(x, u)$ refers to the set of reachable states from x, under the constant control input u, in up to τ time and $Reach_{=\tau}(x, u)$ are the states reached after exactly τ unit time has elapsed. Also, we naturally extend Reach to initial sets of states, where the resultant set of reachable states is the union of the set of reachable states from each state in the initial set.

4 DESIGN APPROACH

As depicted in Figure 1, our proposed design consists of three main components; Base Controller (BC), Mission Controller (MC) and Decision Module (DM).

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*). MC, on the other hand, is the main controller of the system 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.

All the components of the system run on top of the RTOS. The length of one control cycle of the system is τ_c. kth control cycle where $k \in \mathbb{N}$ refers to the period $[(k-1)\tau_c, k\tau_c]$. 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[3] of the system is an integer multiply[4] 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 time stamp (cycle number) to a fixed memory address.

At the end of 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 DM decision and resets WD. Non-matching time stamps 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

[3]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.

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

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 2.

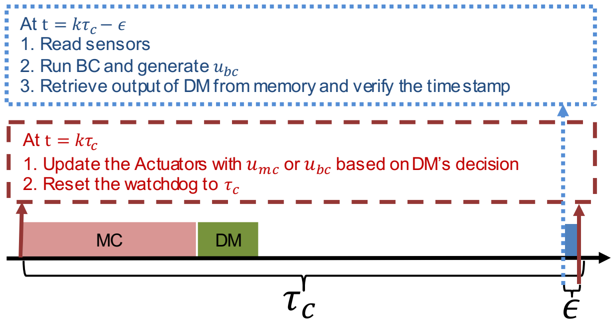
At t = $k\tau_c - \epsilon$
1. Read sensors
2. Run BC and generate u_{bc}
3. Retrieve output of DM from memory and verify the time stamp
At t = $k\tau_c$
1. Update the Actuators with u_{mc} or u_{bc} based on DM's decision
2. Reset the watchdog to τ_c
MC
DM
τ_c
ϵ

Figure 2: Sequence of events within one control cycle.

In the rest of this section, we discuss the assumptions and the fault model of the system. Then, we introduce the properties of the BC and how it is able to safely tolerate the restarts. Finally, we discuss the safe switching logic of the DM.

4.1 Assumptions and Fault Model

In this section, we clarify several assumptions we make about faults and components of the system.

(1) In this work, we are not concerned about hardware faults and we assume that hardware is reliable.
(2) 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.
(3) 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.
(4) 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[5].
(5) We assume that the system-level faults do not happen within the first control cycle τ_c 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 4.2, we demonstrate the necessity of this assumption.

4.2 Properties of the Base Controller

In this section, we introduce what properties the BC must have. Later, in Section 5, we explain how to construct such a controller.

BC has the following properties. There exists a subset $\mathcal{I}$ of the state space, such that for all points $x \in \mathcal{I}$, BC can find a control command $u_{bc} \in \mathcal{S}_u$, such that if u_{bc} is applied to the system at time t_0 and state $x(t_0) \in \mathcal{I}$, then: *(i)* $x(t_0+\tau_c) \in \mathcal{I}$, *(ii)* $x(t_0+\tau_c+\tau_r) \in \mathcal{I}$ and *(iii)* $x(t) \in \mathcal{S}$ for $t \in [t_0, t_0 + \tau_c + \tau_r]$. Note that it is assumed that the actuators hold the command u_{bc} 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, assume that in the above conditions, t_0 is the end of kth control cycle; $t_0 = k\tau_c$. 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 end of the cycle and BC will not be available to update the actuator input. Property *(ii)* implies that in such a case, system will be in $\mathcal{I}$, after the restart completes. This guarantees that the system is stabilizable, 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 in safety, only if it updates the actuator commands at least once after every restart. Therefore, it is necessary for the system to not have any system-level faults within the first cycle after the restart.

4.3 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 that is possible.

In every cycle k, DM runs and evaluates the following conditions. If they 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.

(1) $\text{Reach}_{=\tau_c}(\bar{x}[k], u_{mc}) \subseteq \mathcal{I}$
(2) $\text{Reach}_{=\tau_r+\tau_c}(\bar{x}[k], u_{mc}) \subseteq \mathcal{I}$
(3) $\text{Reach}_{\leq\tau_r+\tau_c}(\bar{x}[k], u_{mc}) \subseteq \mathcal{S}$

Here, τ_r and τ_c are the restart time and the length of the control cycle of the platform. $\bar{x}[k]$ is the state of the system when the actuator command is going to be applied to the system (*i.e.,* 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 safe and restartable. Condition (1) 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 (2) ensures that state will be inside $\mathcal{I}$ when the restart completes (*i.e.,* at $\tau_c + \tau_r$). Furthermore, Condition (3) 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

[5]Commercial chips such as [2] are available that provide programmable PWM controller. Using these intermediate chips prevents the invalid signals that may appear on the GPIO of the board during a restart, from changing the actuation command.

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 previous cycle). $\bar{x}[k]$ can be computed in the following way:

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

where $x[k-1]$ is the sampled state in the previous cycle (state of the system at the beginning of the current control cycle.). 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.

We use the real-time reachability approach proposed in [6] to compute the states that are reachable within a bounded time. The computation time of this algorithm is adjustable. However, the calculated reachable set is tighter with increased computation time. This allows the reachability task to be scheduled in the framework of real-time system computation.

5 BASE CONTROLLER DESIGN

In the previous section, we assumed that a base controller exists with the properties mentioned in Section 4.2. In this section, we explain how to construct such a controller.

To construct the BC, we utilize an algorithm from control theory to find an invariant subset $\mathcal{I} \subseteq \mathcal{S}$ in Section 5.2 and then use $\mathcal{I}$ to find a run-time control strategy for BC in Section 5.3. The particular procedure that we use here is only applicable to linear discrete-time systems. However, there are algorithms in [7] to find invariant subsets for non-linear systems as well.

Continuous dynamics can be converted to the discrete form with sampling time of τ_c in the following way:

$$\begin{cases} \dot{x}(t) = Ax(t) + Bu \\ y(t) = Cx(t) \end{cases} \rightarrow \begin{cases} x[k+1] = A_d x[k] + B_d u[k] \\ y[k] = C_d x[k], \end{cases} \tag{4}$$

where we have the following:

$$A_d = e^{A\tau_c} = \sum_{k=0}^{\infty} \frac{1}{k!}(A\tau_c)^k \simeq \sum_{k=0}^{p} \frac{1}{k!}(A\tau_c)^k, \tag{5}$$

$$B_d = \left(\int_0^{\tau_c} e^{At} dt \right) \cdot B \text{ and } C_d = C. \tag{6}$$

In this section, we show how to construct a BC with the following properties:

$$\begin{gathered} \forall x[k] \in \mathcal{I}, \exists u_0 \text{ where } u[p] = u_0, p \in \{k, k+1, ..., k+m\} \\ \text{such that } (i)\ x[k+1] \in \mathcal{I} \text{ and } (ii)\ x[k+1+m] \in \mathcal{I}, \end{gathered} \tag{7}$$

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

5.1 Readjusting the Safety Region

There is one issue that we need to address before calculating $\mathcal{I}$ and the BC. The property presented in Equation 7 enforces the state to be inside $\mathcal{I}$ after one control cycle and one restart time (if happens) after that. However, it does not imply anything about the trajectory of state within the restart time. To guarantee safety, the state of the system within the restart time must not go outside of $\mathcal{S}$.

To enforce this, we find a subset $\mathcal{S}' \subseteq \mathcal{S}$ such that if $x(t_0) \in \mathcal{S}'$, then $x(t) \in \mathcal{S}$ for any $u(t) \in \mathcal{S}_u$ 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 [6] for computing real-time reachability.

Definitions: Before explaining the procedure, some notations and definitions are necessary. Note that from the definition of $\mathcal{S}$ in Section 3.1, $\mathcal{S}$ 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 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 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 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.

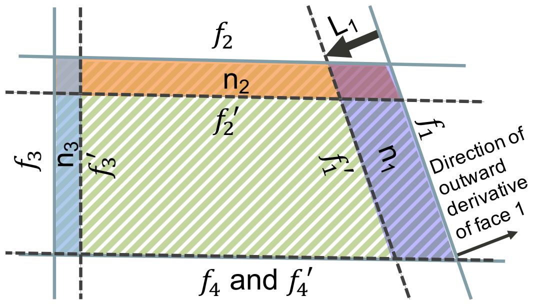

Figure 3: An example to illustrate construction of $\mathcal{S}'$ from $\mathcal{S}$. The area confined with f_1, f_2, f_3 and f_4 is $\mathcal{S}$. The area confined with f_1', f_2', f_3' and f_4' is $\mathcal{S}'$.

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 $H_x^a \cdot x \le h_x^a$, 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, we limit the number of times the algorithm can return to step[6] 2. 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 dynamics. In Section 7, we discuss a practical design approach to reduce the restart time of the multicore platforms.

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 $\mathcal{S}'$.

5.2 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 [7]). We slightly modify this procedure and present it in Algorithm 1 to compute the subset $\mathcal{I} \subseteq \mathcal{S}'$, such that for the discrete-time system in Equation 4, $\mathcal{I}$ satisfies the properties in Section 4.2 or Equation7.

Algorithm 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($H_x^a \cdot x \le h_x^a$) and k = 0
3      while $p < p_{max}$ do
4          [$H_x'$, $h_x'$] := PolytopeToMatrix($I^{(k)}$)
5          pt := Polytope
               $\left(\begin{bmatrix} H_x' \\ H_x' \\ H_u \end{bmatrix} \begin{bmatrix} A_d^{(m+1)} & B_d^{(m+1)} \\ A_d & B_d \\ 0_{m\times n} & I_{m\times m} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \le \begin{bmatrix} h_x' \\ h_u \end{bmatrix}\right)$
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 $B_d^{(m+1)} = (A_d^m + A_d^{m-1} + \ldots + I)B_d$.

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 [7].

If matrix A_d and B_d are controllable, we can use ideas from [22] to ensure convergence. However, in general we cannot guarantee that the procedure in Algorithm 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 be able to apply this approach.

5.3 Base Controller in Runtime

All the previous steps introduced in Subsections 5.1 and 5.2 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:

$$\begin{cases} H_u \cdot u[k] & \le h_u \\ H_x^{\mathcal{I}} \cdot x[k+1] & \le h_x^{\mathcal{I}} \\ H_x^{\mathcal{I}} \cdot x[k+m+1] & \le h_x^{\mathcal{I}} \end{cases} \tag{8}$$

With replacing the $x[k+1]$ and $x[k+m+1]$ from the discrete-time model (4) in the above equations we have the following linear inequalities:

$$\begin{cases} H_u u[k] & \le h_u \\ H_x^{\mathcal{I}} B_d u[k] & \le h_x^{\mathcal{I}} - H_x^{\mathcal{I}} A_d x[k] \\ H_x^{\mathcal{I}} B_d^{(m+1)} u[k] & \le h_x^{\mathcal{I}} - H_x^{\mathcal{I}} A_d^{(m+1)} x[k] \end{cases} \tag{9}$$

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 9. 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.

6 CASE STUDY AND EVALUATION

To demonstrate the practicality of our proposed approach, we implemented a controller system for a 3DOF helicopter[17] (Figure 4) 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 and demonstrate that the system remains safe, despite the faults, and recovers.

[6]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.

6.1 Experimental Setup

For the prototype of the proposed design, a 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 on the considered platform runs FreeRTOS [1], 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 [3]. *(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 coupled memory and releases it from reset. These modifications reduce the real-time core's boot time from seconds to less than 250ms[7].

The control tasks on the real-time core of I.MX7D run with a frequency of 20 Hz ($\tau_c = 50$ ms). Our controller interfaces with the 3DOF helicopter through a PCIe-based *Q8 High-Performance H.I.L. Control and data acquisition unit* [18] 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.

[7]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.

6.2 Physical System Description

3DOF helicopter (displayed in figure 4) 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[17]. 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. We use the linear model of this system obtained from the manufacturer manual[17] for designing the BC and the DM and test the designed controller on the real system.

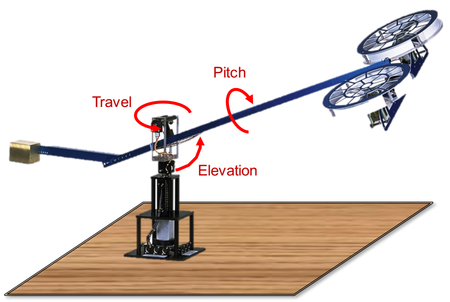

Figure 4: 3 Degree of freedom (3DOF) helicopter.

For 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 4, while respecting the maximum angular velocities. The linear inequalities describing the safety region are given in (10) which is in the form of $H_x \cdot x \leq h_x$. In equation (10), the first two rows of the matrix H_x and vector h_x specify the set of pitch and elevation angles such that the helicopter does not contact with the surface. Rows 3-6 set limits on the maximum angular velocities of the helicopter.

$$\begin{bmatrix} -1 & -0.33 & 0 & 0 & 0 & 0 \\ -1 & 0.33 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 \end{bmatrix} \begin{bmatrix} \epsilon \\ \rho \\ \lambda \\ \dot{\epsilon} \\ \dot{\rho} \\ \dot{\lambda} \end{bmatrix} \leq \begin{bmatrix} 0.3 \\ 0.3 \\ 0.4 \\ 0.4 \\ 1.5 \\ 1.5 \end{bmatrix} \tag{10}$$

Here, variables ϵ, ρ, and λ are the elevation, pitch, and travel angles of the helicopter. Equation (11) describes the limits on the motor voltages of the helicopter in the form of $H_u \cdot u \leq h_u$.

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} v_l \\ v_r \end{bmatrix} \leq \begin{bmatrix} 1.1 \\ 1.1 \\ 1.1 \\ 1.1 \end{bmatrix} \tag{11}$$

Here, v_l and v_r are the voltage for controlling left and right motors.

FreeRTOS on the Cortex-M4 core restarts in 250 ms (upper bound). We apply the algorithm described in Section 5.1 to compute the readjusted safety region for restart time $\tau_r = 0.250$ seconds. In order to improve the computation of $\mathcal{S}'$, it was performed in 5 consequent steps each with a time parameter of $\tau_r/5$. Increasing q beyond 5 did not improve the size of $\mathcal{S}'$. We achieved the following readjusted safety constraints:

$$h_x^a = [0.1418, 0.1418, 0.2828, 0.2828, 0.0825, 0.0825]^T$$

and $H_x^a = H_x$ as given in equation (10).

Using this readjusted safety constraints the invariant region and BC are constructed using the Algorithms described in Sections 5.2 and 5.3. Algorithm 1 computed a region $\mathcal{I}$ confined with 106 inequalities after 14 iterations. The offline computation took 4 hours on Mac Book Pro with 2.5 GHz Intel Core i7 and 16 GB of memory. Finally, the BC is derived using the Equation 9.

6.2.1 Testing the Base Controller. To verify that the constructed base controller has the desired properties, we simulated the system with this controller from all the 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 5 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). And, the control command in this trajectory is $v_r = 0.6863$ and $v_l = 0.7709$. As shown in Figure 5, the trajectory remains inside the safety region defined in Equation 10.

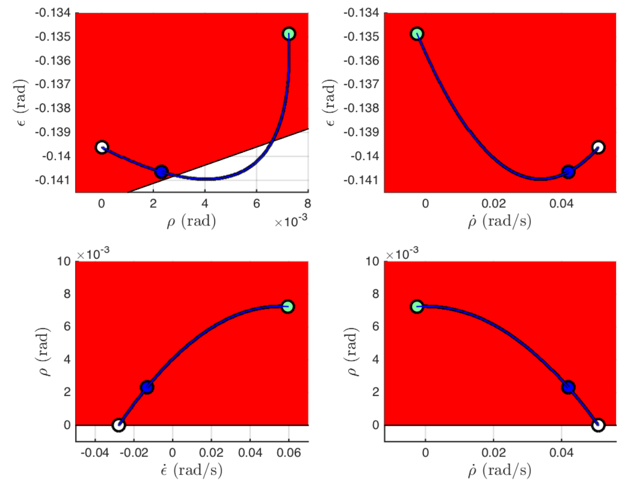

Figure 5: Simulated trajectory of the system under $v_r = 0.6863$ and $v_l = 0.7709$ is inside $\mathcal{I}$ (red region) at times $\tau_c = 50$ ms (blue mark) and $\tau_c + \tau_r = 300$ ms (green mark). White circles mark the beginning of the trajectory. The trajectory is projected into the four planes for clarity.

In addition to the simulations, we implemented the obtained BC on the i.MX7D platform to validate our design approach. Based on the properties of BC in Section 4.2, it should be able to stabilize the system with only one control cycle after every restart. To test the BC, we impose this worst case scenario such that BC could only update the actuators once after every restart (*i.e.,* one actuator update every $\tau_c + \tau_r = 300$ ms). It was observed that if the starting state was in $\mathcal{I}$, BC could stabilize the system.

6.3 Fault Injection

In Table 1, a list of faults that were tested on the implementation is 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 3DOF helicopter as long as it did not jeopardize safety. When the helicopter's state approached the states where the safety conditions were not satisfied, BC took over and prevented the helicopter from hitting the surface underneath. For the system-level faults, we observed that the WD restarted the system and after restart system continued its operation.

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

6.3.1 Maximum Voltage in Wrong Way. The helicopter should not hit the surface even if the MC outputs a voltage that normally would result in a crash. We consider an extreme case of this scenario where the MC generates a voltage that pushes the helicopter towards the surface. 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 and then control was handed back to MC.

6.3.2 Timing faults (CPU and Resource). Our 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 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.

7 DISCUSSION

Software Faults: Current proposed approach does not handle software faults that modify the program logic or output of the BC and the DM at execution time. Utilizing frameworks such as ARM

3DOF Helicopter Fault Injection					
		Safety			
Failure Type	Fault Category	Application-Level Simplex (Single HW Board/SoC)	System-Level Simplex (Additional HW/SoC)	Our Approach (Single HW Board/SoC)	Restarted
No Output	App.	✔	✔	✔	No
Maximum Voltage	App.	✔	✔	✔	No
Time Degraded Control	App.	✔	✔	✔	No
Timing Fault - CPU	RTOS/App.	✗	✔	✔	Yes
Timing Fault - Resource	RTOS/App.	✗	✔	✔	Yes
FreeRTOS Freeze	RTOS	✗	✔	✔	Yes
Computer Reboot	RTOS	✗	✔	✔	Yes

Table 1: Our approach tolerates system-level faults using only one hardware unit. Whereas, System-Level Simplex [5] needs an extra board/SoC to tolerate these faults.

TrustZone [16] and limiting the access to these critical components can mitigate this issue.

Restart Time: As the restart time of the platform increases, domain of the BC shrinks. Therefore, the proposed solution in its current form, even though useful for many platforms, may not suit some platforms with a longer restart time.

We are actively working on an alternative multi-stage booting solution for multicore platforms to mitigate this problem. Our main idea is to boot one core with the bare minimum requirements to execute the BC in the shortest possible time. The BC 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. As a future extension, we are working on implementing this solution on i.MX7D platform. We first boot the real-time core with a FreeRTOS and run the BC on top of it and then boot an embedded Linux on the general purpose core.

Linear Dynamics: In this paper, we have only considered systems that can be modeled by linear differential equations. One can apply our design to non-linear systems using the ideas in [28] and [21]. This may however increase the computational complexity of computing the invariant region $\mathcal{I}$ in the design time.

8 CONCLUSION

Restarting is considered as a reliable way to recover the traditional computing system from software faults. However, restarting safety-critical CPS is challenging. In this work we propose a novel approach that guarantees *safety* and *liveness* in the presence of software faults in the application-layer as well as system-layer faults utilizing only one COTS processor based on complete system-level restarts.

ACKNOWLEDGMENTS

This work is supported by the German Research Foundation (DFG) through the grant ZA 873/1-1, the TUM International Graduate School of Science and Engineering (IGSSE), and the National Science Foundation (NSF) under grant numbers CNS-1302563 and CNS-1646383. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the NSF and other sponsors.

REFERENCES

[1] 2016. FreeRTOS . http://www.freertos.org. (2016). Accessed: Sep. 2016.
[2] 2016. PCA9685: 16-channel, 12-bit PWM Fm+ I2C-bus LED controller. https://goo.gl/FMnOQT. (2016). Accessed: Oct. 2016.
[3] 2017. https://github.com/abditag2/reset-based-recovery. (2017).
[4] Fardin Abdi, Renato Mancuso, Stanley Bak, Or Dantsker, and Marco Caccamo. 2016. Reset-Based Recovery for Real-Time Cyber-Physical Systems with Temporal Safety Constraints. In *IEEE 21st Conference on Emerging Technologies Factory Automation (ETFA 2016).*
[5] Stanley Bak, Deepti K Chivukula, Olugbemiga Adekunle, Mu Sun, Marco Caccamo, and Lui Sha. 2009. The system-level simplex architecture for improved real-time embedded system safety. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE.* IEEE, 99–107.
[6] Slawomir Bak, Taylor T Johnson, Marco Caccamo, and Lui Sha. 2014. Real-time reachability for verified simplex design. In *Real-Time Systems Symposium (RTSS), 2014 IEEE.* IEEE, 138–148.
[7] Franco Blanchini and Stefano Miani. 2008. *Set-theoretic methods in control.* Springer, 156–163.
[8] George Candea, James Cutler, and Armando Fox. 2004. Improving Availability with Recursive Microreboots: A Soft-state System Case Study. *Perform. Eval.* 56, 1-4 (March 2004), 213–248. DOI:http://dx.doi.org/10.1016/j.peva.2003.07.007
[9] George Candea and Armando Fox. 2001. Recursive restartability: Turning the reboot sledgehammer into a scalpel. In *Hot Topics in Operating Systems, 2001. Proceedings of the Eighth Workshop on.* IEEE, 125–130.
[10] George Candea and Armando Fox. 2003. Crash-Only Software. In *HotOS IX: The 9th Workshop on Hot Topics in Operating Systems.* 67–72.
[11] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. 2004. Microreboot- A Technique for Cheap Recovery. In *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04).* 3–3.
[12] George Candea, Emre Kiciman, Steve Zhang, Pedram Keyani, and Armando Fox. 2003. JAGR: An autonomous self-recovering application server. In *Autonomic Computing Workshop. 2003. Proceedings of the.* IEEE, 168–177.
[13] Tanya L Crenshaw, Elsa Gunter, Craig L Robinson, Lui Sha, and PR Kumar. 2007. 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, 400–412.
[14] Sachin Garg, Antonio Puliafito, Miklós Telek, and Kishor S Trivedi. 1995. Analysis of software rejuvenation using Markov regenerative stochastic Petri net. In *Software Reliability Engineering, 1995. Proceedings., Sixth International Symposium on.* IEEE, 180–187.
[15] Yennun Huang, Chandra Kintala, Nick Kolettis, and N Dudley Fulton. 1995. Software rejuvenation: Analysis, module and applications. In *Fault-Tolerant Computing, 1995. FTCS-25. Digest of Papers., Twenty-Fifth International Symposium on.* IEEE, 381–390.
[16] ARM Inc. ARM TrustZone. (????).
[17] Quanser Inc. 3 DOF Helicopter. (????).
[18] Quanser Inc. Q8 Data Acquisition Board. (????).
[19] Edward A Lee. 2008. Cyber physical systems: Design challenges. In *2008 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC).* IEEE, 363–369.

[20] Sibin Mohan, Stanley Bak, Emiliano Betti, Heechul Yun, Lui Sha, and Marco Caccamo. 2013. 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, 65–74.

[21] G. Reißig, A. Weber, and M. Rungger. 2017. Feedback Refinement Relations for the Synthesis of Symbolic Controllers. *IEEE TAC* 62 (2017). DOI: http://dx.doi.org/10.1109/TAC.2016.2593947

[22] Matthias Rungger and Paulo Tabuada. 2016. Computing Robust Controlled Invariant Sets of Linear Systems. *CoRR* abs/ (2016). http://arxiv.org/abs/1601.00416

[23] Danbing Seto and Lui Sha. 1999. An engineering method for safety region development. (1999).

[24] Lui Sha. 1998. Dependable system upgrade. In *Real-Time Systems Symposium, 1998. Proceedings., The 19th IEEE*. IEEE, 440–448.

[25] Lui Sha. 2001. Using simplicity to control complexity. IEEE Software, 20–28.

[26] Lui Sha, Ragunathan Rajkumar, and Michael Gagliardi. 1996. Evolving dependable real-time systems. In *Aerospace Applications Conference, 1996. Proceedings., 1996 IEEE*, Vol. 1. IEEE, 335–346.

[27] Sardar Muhammad Sulaman, Alma Orucevic-Alagic, Markus Borg, Krzysztof Wnuk, Martin Höst, and Jose Luis de la Vara. 2014. 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, 17–24.

[28] P. Tabuada. 2009. *Verification and control of hybrid systems*. Springer, New York.

[29] Kalyanaraman Vaidyanathan and Kishor S Trivedi. 2005. A comprehensive model for software rejuvenation. *Dependable and Secure Computing, IEEE Transactions on* 2, 2 (2005), 124–137.

[30] Prasanth Vivekanandan, Gonzalo Garcia, Heechul Yun, and Shawn Keshmiri. 2016. A Simplex Architecture for Intelligent and Safe Unmanned Aerial Vehicles. In *2012 IEEE International Conference on Embedded and Real-Time Computing Systems and Applications(RTCSA)*.