Technical Report: Reset-Based Recovery for Real-Time Cyber-Physical Systems with Temporal Safety Constraints

Fardin Abdi Taghi Abad[1], Renato Mancuso[1]
Stanley Bak[2], Or Dantsker[3], Marco Caccamo[1]

[1] Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, USA
[2] United States Air Force Research Lab, Rome, USA
[3] Department of Aerospace Engineering, University of Illinois at Urbana-Champaign, Urbana, USA

Abstract

In traditional computing systems, software problems are often resolved by platform restarts. This approach, however, cannot be naïvely used in cyber-physical systems (CPS). In fact, in this class of systems, ensuring safety strictly depends on the ability to respect hard real-time constraints. Several adaptations of the Simplex architecture have been proposed to guarantee safety in spite of misbehaving software components. However, the problem of performing recovery into a fully operational state has not been extensively addressed.

In this work, we discuss how resets can be used in CPS as an effective strategy to recover from a variety of software faults. Our work extends the Simplex architecture in a number of directions. First, we provide sufficient conditions under which safety is guaranteed is spite of fault-induced resets. Second, we introduce a novel technique to express not only state-dependent safety constraints, as typically done in Simplex, but also time-dependent safety properties. Finally, through a proof-of-concept minimal implementation on a small R/C helicopter and simulation-based system modeling, we show the effectiveness of the proposed recovery strategy under the assumed fault model.

I. Introduction

There are an increasing number of CPS applications in almost all the vital infrastructures of our modern society. Such systems often have a set of safety requirements that need to remain satisfied at all times because a violation could have catastrophic consequences. However, software components can exhibit unexpected deviations from the intended behavior due to bugs, potentially violating the safety requirements. Unfortunately, formally assessing the correctness of software components is a hard problem since the existing approaches currently require a large amount of effort (cost) as well as specialized knowledge which is not yet widespread.

The difficulty of producing 100% correct software is a strong incentive to develop techniques to enforce safety requirements in CPS in spite of unexpected misbehavior. However, safety is not the only goal of a CPS, also the capability to remain in a fully operational state is of paramount importance. Techniques designed to maintain safety have received substantial attention in the literature [1–7]. In comparison, the problem of restoring nominal operation for a CPS has received little attention. In this work, we improve the state of the art of safety enforcement for CPS and discuss the use of resets as a strategy to fully recover from transient faults.

The Application-level Simplex architecture, proposed in [1–4], represents a well-known method to provide safety guarantees for CPS. In the Simplex architecture, a verified, *simple* safety controller ensures the stability of the plant. This conservative safety controller is complemented by a high-performance *complex* controller. A decision module continuously evaluates the safety properties and forwards actuation commands from the complex controller whenever the system operates within the safety margins. If a misbehavior is detected from the state of the plant, control is transferred to the safety controller. This prevents the occurrence of faults within the complex controller from compromising the safety of the plant. The main issue with the Application-level Simplex is that safety and complex controller are implemented as two applications on the same platform. Hence, in presence of platform-level faults, there is no guarantee of correct behavior from the safety controller. The issue is addressed in the System-level Simplex architecture [7] by moving the safety controller and the decision module into a dedicated processing unit. The safety controller in both the Application- and System-level Simplex has safety boundaries that are typically pessimistic and statically computed at design time. The work in [5] demonstrated that real-time reachability analysis can be employed to relax such static constraints. In fact, a plant can be allowed to abandon its safety boundaries as long as (i) no constraints are violated, and (ii) the state can be guaranteed to re-enter the safety region.

In this paper, we build upon the work in [5] and improve over System-level Simplex [7] in three main directions. First, we show that when the real-time aspects of the safety controller are considered, it is possible to provide safety guarantees that are as strong as what were provided by System-level Simplex without the need for additional, dedicated hardware. Second, we extend real-time reachability to check safety properties that depend not only on the current state of the system as originally proposed in [5], but also on its history. Finally, to the best of our knowledge, we are the first to extensively discuss how platform-wise resets can be employed in CPS as a way to (i) perform fault recovery and to (ii) restore a full operational status.

In order to evaluate the validity and feasibility of the proposed strategy, we conduct a case study using a radio-controlled helicopter testbed. For our study, we use sensor traces acquired in flight while manually injected faults trigger platform-level recovery through resets. The acquired data is used to tune and validate our helicopter model. Next, we perform simulation-based analysis of the complete system based on the validated model. Our results show that: (i) restarting represents a feasible fault recovery approach; (ii) it is possible to formulate systems constraints so that static and time-dependent safety constraints are respected despite the occurrence of resets; and (iii) if fast reset times can be achieved, the proposed recovery methodology has a negligible impact on system's performance.

This paper is organized as follows. A brief review of the related works is presented in Section II. In Section III, we formalize the two categories of safety guarantees that our design can provide. In Section IV, a background on the Simplex architecture is presented. Section V, provides the overall design methodology. In Section VI, an alternative restartable architecture is proposed. The evaluation on the helicopter system is provided in Section VII. Section VIII concludes the paper.

II. Related Work

Restart based strategies are generally divided into two categories, *revival*, reactively restart a failed component, and *rejuvination*, prophylactically restart functioning components to prevent state degradation. [8] introduces recursively restartable systems as a design paradigm for highly available systems and uses a combination of revival and rejuvenation techniques. Authors in [9–11] propose 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. Some works have focused on failure and fault modelling [12–14] and try to find the optimal rejuvenation strategy. These techniques are proposed for traditional computing systems and are not applicable to CPS. In [15], authors propose improving reliability of real-time control systems by executing simultaneous task replica of varying complexity. When a task fails, one of the shadow tasks can provide the output while the failed task is restarted to a clean state. However, this system has no mechanism to guarantee any system constraints. To our knowledge, this is the first work to extensively consider enabling systematic restart-based recovery with safety guarantees for CPS.

III. System Constraints

The *safety requirements* of a system are conditions that need to remain satisfied at all times during system operation. In this work we consider two categories of constraints: *Hard* Constraints and *Overrun* Constraints.

Hard Constraints are expressed in the form of hard, physical constraints over the system's state space. When considered together, they determine the feasible regions of the state space where the system can operate. Each hard constraint is presented as a linear inequality of the form:

$$a_m^T \cdot x \leq 1,$$

where $x \in \mathbb{R}^n$ is the vector of state variables of the system and $a_m \in \mathbb{R}^n$ is a vector of constants. For instance, for a helicopter, a hard constraint is imposed on the altitude to prevent a crash.

On the other hand, **Overrun Constraints** are defined on the trajectory of the system over time. An overrun constraint has the following form:

$$\forall t; \int_t^{t+T^{win}} \text{Stress}(x(\tau)) \cdot d\tau \leq C \tag{1}$$

Here "Stress" is a *non-negative* function that defines the amount of instantaneous stress on the system for a given state, x. C is the maximum amount of accumulated stress that is allowed over any time window of length T^{win}. Hence, each overrun constraint with index k is specified by the tuple $\langle \text{Stress}_k(x), C_k, T_k^{win} \rangle$.

For example, the propulsion system is limited by its ability to dissipate heat. Consequently, motor datasheets specify the maximum time the motor can be operated at full power. For instance, the maximum duration allowed for "Hacker" brushless motors to operate at full power is 15 seconds [16, 17]. this can be implied as

$$\text{Stress}(p) = \begin{cases} 1 & p > p_h \\ 0 & p \leq p_h \end{cases} \text{ and}$$

$$\forall t; \int_t^{t+16} \text{Stress}(p(\tau)) \cdot d\tau \leq 15$$

where p represents the instantaneous propulsion power and p_h the threshold for full power level.

The combination of all the constraints, is referred to as **System Constraints**. The goal of design verification techniques for CPS is to ensure that all the system constraints are met throughout operation.

IV. Background on Simplex Architecture

The goal of using Simplex is to enable a system designer to use an unverified controller on the system while ensuring the same safety guarantees that a verified safety controller would offer. The 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 an LMI form. 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 linear system is guaranteed using Lyapunov's direct or indirect methods. Furthermore, the matrix P defines an ellipsoid in the state space where all 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.

It follows that the ellipsoid of states, $\mathcal{R} = \{x | x^T Px < 1\}$, is a subset of the recoverable states. As long as the system's state remains inside of the ellipsoid, the system will be driven toward the equilibrium point, i.e. where $x^T Px = 0$, when control is handed over to the safety controller. Since the potential function is strictly decreasing over time, any trajectory starting inside $\mathcal{R}$ will remain there for an unbounded time window. Therefore no unsafe states will ever be reached as there are no such states in $\mathcal{R}$.

V. Methodology

In this section we describe our design methodology. The core of this design is the Simplex Architecture proposed in [1–4]. Our first goal is to extend Simplex to provide runtime guarantees for *hard* constraints and the additional category of *overrun* constraints in spite of faults in complex controller. The second goal is to allow recovery from faults through platform-level restarts.

Our design is comprised of a verified, simple Safety Controller (SC) as well as an unverified Complex Controller (CC). Thanks to the properties discussed in Section IV, SC is able to take control and stabilize the system. To guarantee safety, therefore, the output of SC is set as the default control command for the physical system. A real-time reachability module (RTR) periodically checks whether the safety requirements of the system remain satisfied under all possible control commands of the CC. If these conditions hold, CC is safely left in charge for the next control cycle. This approach prevents logical bugs in the CC from violating any of the system safety constraints.

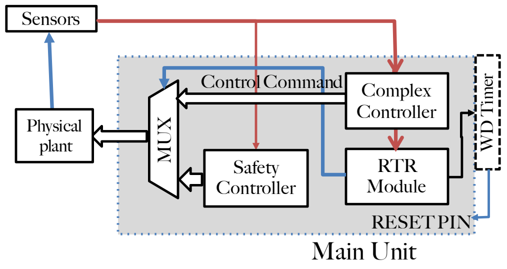


Figure 1: Restartable Simplex architecture.

It follows that respecting the real-time timing constraints of SC tasks is the only necessary condition to guarantee safety. In order to check the schedulability of SC task in presence of restarts, we model the restart as a sporadic non-preemptive task with the highest priority. Next, we check whether the task set consisting of the SC and the *restart task* is schedulable. In Section I we present a methodology that is an extension of classic response-time analysis and provides a sufficient condition for schedulability in presence of restarts.

In this paper, the assumed fault model for the SC and the RTR modules is **fail-stop** and for the CC is **Byzantine**. We rely on hardware watchdog timers to ensure that the platform will recover after any fail-stop failure. Periodic, controlled resets or any strategy based on misbehavior detection performed by the RTR module can be employed to recover from Byzantine failures.

In the rest of this section, we explain in detail how each component should be implemented and how the proposed architecture meets our design goals.

I. Schedulability Analysis with Resets

In this section we present an analysis to reason about the schedulability of critical workload in presence of faults and resets.

A necessary assumption for the analysis has to be made on the frequency of faults and the consequent resets. For a well-tested system, faults that require resets are typically considered to be rare events [18, 19]. In this section we assume that it is possible to determine a minimum inter-arrival time for faults/resets. In order to reason on the *schedulability* in presence of faults, we model a fault and the resulting system-wide reset as a sporadic task. Specifically, we indicate with T_r the minimum inter-arrival time of the faults/resets and with C_r the length of the reset. Conversely, if the minimum inter-arrival time for faults cannot be determined, the alternative architecture described in Section VI can be used.

Let us consider a taskset $\mathcal{T}$ composed of n sporadic tasks $\tau_1 \dots \tau_n$. Each task τ_i is characterized by a minimum inter-arrival time T_i, a worst-case execution time (WCET) C_i and a relative deadline $D_i \leq T_i$. Tasks are scheduled according to fixed-priority scheduling (e.g. RM [20]). We further assume that the considered tasks are controller tasks. As such, each task instance (job) is independent and performs sampling, computation and output of actuation commands. As such, a job remains unaffected by system resets as long as it has completed or it has not been started before a reset occurs. Conversely, if a particular task was executing (or preempted) when the reset was triggered, the affected task instance will need to be re-executed after the reset sequence is completed.

The underlying assumption is that minimal scheduler state can be preserved across resets. We discuss miminal carryover of state across resets in the end of this section. Note that if scheduling algorithms with job-level static priority are used (e.g. fixed-priority, EDF), the state of the scheduler needs to be saved only at the boundaries of tasks' activation and completion.

Furthermore, the following condition is assumed on the minimum inter-arrival time of faults/resets:

$$\max_{\tau_k \in \mathcal{T}} \{C_k\} + C_r < T_r. \tag{2}$$

If the condition in Equation 2 is not satisfied, a sequence of resets could continuously cause the same job to be restarted, ultimately preventing any useful computation to be performed on the platform.

Given our model and assumptions, the goal is to understand under which workload conditions a fault and subsequent reset do not compromise the timely execution of the safety controllers [1] In this work, we consider non-preemptive tasks. The main advantage of having non-preemptive tasks is that at most one task instance is affected by a reset at any instant of time. It is possible to analyze the response time of a task in presence of resets and check its schedulability using Theorem 1.

Theorem 1. *A set of non-preemptive sporadic tasks scheduled according to fixed-priority is schedulable in presence of resets if the response time R_i of each task τ_i, calculated by solving the iterative formula*

$$R_i^{(k+1)} = B_i + \sum_{\tau_j \in hp(\tau_i) \cup \{\tau_i\}} \left\lceil \frac{R_i^{(k)}}{T_j} \right\rceil C_j + I_{r,i} \tag{3}$$

satisfies the condition: $R_i \leq D_i$. In Equation 3 the term B_i is the blocking from low priority tasks and is calculated as:

[1] It is possible to run multiple control applications and consequently multiple SCs and CCs on the same platform.

$$B_i = \max_{\tau_k \in lp(\tau_i)} \{C_k\}. \tag{4}$$

The term $I_{r,i}$ represents the interference on task execution introduced by resets and is calculated as follows:

$$I_{r,i} = \begin{cases} C_r + \max\{B_i, C_i, H_i\} & \textit{if } R_i^{(k)} < T_r \\ \max\{I_{r,i}^{int}, I_{r,i}^{mix}, I_{r,i}^{ext}\} & \textit{if } R_i^{(k)} \geq T_r \end{cases} \tag{5}$$

where the term $H_i = \max_{\tau_k \in hp(\tau_i)} \{C_k\}$ represents the longest instance of task with priority higher than τ_i, while the terms $I_{r,i}^{ext}$, $I_{r,i}^{int}$ and $I_{r,i}^{mix}$ can be computed as in Equations 6, 7 and 8 respectively.

$$I_{r,i}^{ext} = 2C_r + B_i + C_i + (\lceil R_i^{(k)}/T_r \rceil - 2)(H_i + C_r); \tag{6}$$

$$I_{r,i}^{int} = \lceil R_i^{(k)}/T_r \rceil (H_i + C_r); \tag{7}$$

$$I_{r,i}^{mix} = C_r + \max\{B_i, C_i\} + (\lceil R_i^{(k)}/T_r \rceil - 1)(H_i + C_r). \tag{8}$$

Proof. First, note that Equation 3 without the last term corresponds to the classic response time analysis for non-preemptive fixed priority scheduling [21].

The additional term represents $I_{r,i}$ the interference introduced by resets on the task instance under analysis. In case only one reset occurs within the response time of the job under analysis, the reset-related interference is maximized when the reset occurs at the end of the longest job in the interval $[0, R_i^{(k)}]$. This is captured by the first case of Equation 5.

Conversely, if two or more resets are possible within the response time of the considered task, there are three possible outcomes that need to be considered.

Case 1. If $\min\{B_i, C_i\} > H_i$ then the worst-case corresponds to the case when two resets cause the complete re-execution of the blocking task of length B_i and the job under analysis C_i, while the remaining resets cause the re-execution of the longest interfering task of length H_i. In this case, $I_{r,i}^{\text{ext}}$ is an upper-bound on the amount of reset-induced interference.

Case 2. If $H_i > \max\{B_i, C_i\}$ then the worst-case reset-induced interference is observed when the reset always affects the longest interfering task (H_i). This case is captured by $I_{r,i}^{\text{int}}$.

Case 3. Finally, if $\max\{B_i, C_i\} > H_i > \min\{B_i, C_i\}$, the term $I_{r,i}^{\text{mix}}$ captures the worst-case. In this scenario, one reset causes a re-execution of length $\max\{B_i, C_i\}$, while the remaining resets cause the re-execution of instances of the longest interfering task (H_i).

The maximum of the three cases described above is taken in Equation 5 when $R_i^{(k)} \geq T_r$. □

As previously discussed, **only** the schedulaility of **SC tasks** needs to be checked to ensure safety. Hence, the analysis proposed in Equation 3 of Theorem 1 needs to be satisfied only for those tasks that correspond to SC modules. Note that since tasks with lower priority can block higher priority tasks, the task-set $\mathcal{T}$ needs to include all the tasks (SC, CC, RTR, and etc.) running on the platform.

In general, it can be safely assumed that the minimum inter-arrival time of resets is larger than the largest task deadline, i.e. $T_r > \max_{\tau_k \in \mathcal{T}} \{D_k\}$. If this more restrictive condition holds, the analysis can be further simplified to consider only the first case of Equation 5.

In terms of preserving scheduler state across resets, this can be done by saving the schedule status to nonvolatile memory at each context switch. An alternative mechanism is to synchronize the schedule with the real-time clock and rely on timing to detect which task needs to be re-executed.

II. Safety Controller (SC) Design

SC is a simple verified controller that is responsible for ensuring that hard and overrun constraints are satisfied. Our methodology consists in: (i) finding a region of states such that, for any trajectory taken by the system inside this region, all the constraints remain satisfied. Next, (ii) we design the SC such that it can keep the state inside this region for an unbounded amount of time as long as the starting state is inside the region. In order to do this, we first express all the constraints as linear inequalities. Then we use a LMI solver to derive a feedback controller with the discussed properties.

Hard constraints, as described in Section III, are already in the form of linear inequalities. Hence, for a system with q hard constraints, we can easily define a region $\mathcal{S}$ such that all the hard constraints are satisfied:

$$\mathcal{S} = \{x | a_m^T \cdot x \leq 1, m = 1, \ldots, q\}. \tag{9}$$

For an overrun constraint, we first define a subset of the state space as region O in the following way:

$$O = \{x | \text{Stress}(x) \leq (1-\alpha)C/T^{win}\}. \tag{10}$$

The integration of Stress(x) over any trajectory of length T^{win} inside O would remain less than the maximum permitted accumulated stress, C. Here, α is the **Manoeuvrability coefficient** and is $0 \leq \alpha \leq 1$. The choice and impact of α will be further discussed in the context of Lemma 1. It can be easily shown that for any trajectory of length T^{win} inside the region O, the overrun constraint holds:

$$\forall t; \int_t^{t+T^{win}} \text{Stress}(x(\tau)) \cdot d\tau \leq \frac{(1-\alpha)C}{T^{win}} \times T^{win} \leq C \tag{11}$$

However, in order to use the region O in a LMI-solver, we choose p linear inequalities to determine a convex subset, $Conv(O) \subseteq O$ such that:

$$Conv(O) = \{x | c_i^T \cdot x \leq 1, i = 1, ..., p\} \subseteq O$$

Since linear matrix inequality can only handle linear systems, we also need to restrict the system with actuator saturation limits (so that actuation values do not saturate the actuators). Therefore, assuming that $u \in \mathbb{R}^m$ is the control signal to the actuators, saturation limits can be expressed as:

$$b_j^T \cdot u \leq 1, j = 1, \ldots, r$$

For a system whose dynamics are described by $\dot{x} = Ax + Bu$, the SC is a linear state feedback control given by $u = Kx$. The feasible region Γ of the system with q hard constraints, p overrun constraints (each overrun constraint itself has p_i linear inequalities) and r actuator constraints can be described by:

$$\begin{aligned} \Gamma = \{x | a_m^T \cdot x \leq 1, m = 1, \ldots, q, \\ c_{i,k}^T \cdot x \leq 1, k = 1, \ldots, p_i, i = 1, \ldots, p, \\ b_j^T \cdot u \leq 1, j = 1, \ldots, r\} \end{aligned} \tag{12}$$

Note that $\Gamma \subseteq \mathcal{S}$, $\Gamma \subseteq O$, and Γ embeds saturation limits. Now, we can use a LMI solver[2] to find Γ, the gain matrix K of SC (SC is a state feedback controller), and the matrix Q. The latter matrix Q is found such that the Lyapunov potential function $V(x) = x^T Q^{-1} x$ constructed for the system under the feedback control of K (i.e. $\dot{x} = (A + BK)x$) has negative-definite derivatives in

[2]This is a standard minimization problem and can be solved using the approach proposed in the second appendix of the technical report in [22].

the region $\mathcal{R} = \{x | x^T Q^{-1} x < 1\} \subseteq \Gamma$.

We refer to $\mathcal{R}$ as the *Stability region*. Due to the negative-definite derivatives of $V(x)$ inside $\mathcal{R}$, any trajectory starting in $\mathcal{R}$ will remain in $\mathcal{R}$ indefinitely.

III. Real-Time Reachability Module (RTR)

We know that the SC can stabilize the system and guarantee all the constraints as long as the state of the system is inside $\mathcal{R}$. The goal of RTR module is to allow the system to operate beyond the boundaries of region $\mathcal{R}$.

In this section, we derive a set of conditions that if satisfied at the beginning of a cycle, hard and overrun constraints are guaranteed to remain satisfied throughout that cycle under any behavior of the CC. At every cycle T_c, RTR checks if those switching conditions hold (Theorem 2 and 3). If they do, the CC is allowed to control the system. Otherwise, SC will be in charge.

Next, we describe the changes in the implementation of the RTR module, with respect to [5], that are required to check the switching conditions for the additional category of overrun constraints.

In the rest of this section, we use $\text{Reach}_{=T}(x, C)$ to denote the set of states reached by system from an initial set of states x after exactly T seconds have elapsed under the control of controller C. $\text{Reach}_{\leq T}(x, C)$ can be defined as $\bigcup_{t=0}^{T} \text{Reach}_{=t}(x, C)$. In order to compute the reach set, we use a modified version of the the face-lifting technique in [5]. We described our technique in Section III.3.

III.1 Switching Conditions for Hard Constraints

Consider T_c the control interval and T_s an arbitrary settling time for the system after a restart.

Theorem 2. *The hard constraints of the system will always remain satisfied under the control of CC, if at every control interval, T_c, the following conditions hold:*

1. $Reach_{\leq T_c}(x, CC) \subseteq \mathcal{S}$;
2. $Reach_{\leq T_s}(Reach_{\leq T_c}(x, CC), SC) \subseteq \mathcal{S}$;
3. $Reach_{=T_s}(Reach_{\leq T_c}(x, CC), SC) \subseteq \mathcal{R}$.

Proof. First, we provide an intuition of the proof. Condition 1 implies that all the states that can be reached under the CC within the next control cycle satisfy the hard constraints. If a switch to SC is triggered at any moment within the next control cycle, Condition 2 ensures that from the time of switching, for an interval of length T_s, the system will not violate any of the hard constraints. Finally, Condition 3, implies that by the end of T_s, the system is inside the stability region where the hard constraints will remain satisfied indefinitely.

Formally, we prove this by induction. The base case holds since the system assumed to start operation from a safe state. For the inductive step, let's assume that the above conditions were true at the previous cycle, $k-1$, and the hard constraints are maintained from within the past control interval. Now, at cycle k, if any of the above conditions are not satisfied the RTR will trigger a switch to SC. Since the conditions 2 and 3 were true in cycle $k-1$, switching to SC is guaranteed to maintain the hard constraints satisfied indefinitely. Now, if at cycle k all the above conditions hold, first we need to show that if the system is not restarted within the next cycle, hard constraints will hold until the beginning of the cycle. Second, we need to show that controller can be safely switched to SC at any point from now until the beginning of next cycle (and including the beginning of the next cycle).

Condition 1 implies that if no switching occurs within $[kT_c, (k+1)T_c]$, the hard constraints will remain satisfied during this interval and at the beginning of the next cycle. Now, we show that the system is also safely switchable in this interval. If a switching from CC to SC is triggered at a time $t_{switch} \in [k.T_c, (k+1).T_c]$, where the state of the system is $x(t_{switch})$, we have $x(t_{switch}) \in \text{Reach}_{\leq T_c}(x, CC)$, hence based on condition 3, $\text{Reach}_{\leq T_s}(x(T_s), SC) \subseteq \mathcal{S}$ which means that the system will not violate hard constraint for the next T_s seconds. Moreover, according to condition 2, we have $\text{Reach}_{=T_s}(x(t_{switch}), SC) \subseteq \mathcal{R}$ which indicates that after T_s, states will be inside $\mathcal{R}$, hence, hard constraints will remain satisfied indefinitely. □

III.2 Switching Conditions for Overrun Constraints

We assume that switching conditions for overrun constraints are checked only if the switching conditions for hard constraints are already satisfied. Therefore, Condition 3 in Theorem 2, implies that if a switch to SC occurs within the upcoming T_c time units, the SC will be able to safely bring the system back into the stability region within at most T_s time units. Hence, all the trajectories that satisfy the hard constraints have a form such that there is a time point t_s at which the trajectory enters the stability region $\mathcal{R}$ while the SC controller is in charge. For such trajectories, Lemma 1 implies a general condition under which a given overrun constraint remains satisfied throughout the execution.

Lemma 1. *Assume an arbitrary trajectory that at some point in time, t_s, enters the stability region $\mathcal{R}$ while the SC is the active controller from that point forward. An overrun constraint is satisfied throughout such a trajectory if the following condition holds:*

$$\forall t \in [0, t_s - T^{win}] : \int_t^{t+T^{win}} \text{Stress}(x(\tau)) \cdot d\tau \leq \alpha C \tag{13}$$

Proof. Here, α is the maneuverability constant[3]. To show that a overrun constraint is satisfied throughout the trajectory, we need to show that for all the time windows of size T^{win} in $[0, +\infty]$ the accumulated stress is less than the limit of the overrun constraint. First, for any interval of size T^{win} starting at $t \in [0, t_s - T^{win}]$, Condition 13 implies that the overrun constraint is satisfied. Second, for any interval of size T^{win} starting at $t \in [t_s, \infty]$, the entire length of the trajectory during the interval is inside the stability region and the system is controlled by the SC. Hence, it is guaranteed by Equation 11 that the overrun constraint will remain satisfied.

Finally, we need to show that the overrun constraint is satisfied for all the intervals of size T^{win} starting at $t \in [t_s - T^{win}, t_s]$. We have the following:

$$\int_t^{t+T^{win}} \text{Stress}(x(\tau)) \cdot d\tau =$$

$$\int_t^{t_s} \text{Stress}(x(\tau)) \cdot d\tau + \int_{t_s}^{t+T^{win}} \text{Stress}(x(\tau)) \cdot d\tau$$

$$\leq \alpha C + \frac{(1-\alpha)C}{T^{win}}(T^{win} + t - t_s) \leq C$$

Note that the last line follows from $T^{win} + t - t_s \leq T^{win}$. Hence, the overrun constraints are satisfied for the entire time span of the trajectory. □

Lemma 1, implies a general condition for a trajectory to satisfy overrun conditions. However, switching conditions need to be time-discrete in order to be checked by the RTR module in

[3]The *Maneuverability Coefficient* is a design parameters such that $0 \leq \alpha \leq 1$. The choice of a larger α can increase the stability region of the SC. At the same time, it makes the conditions of Theorem 3 harder to satisfy, resulting in more frequent switches from CC to SC. Thus, α needs to be chosen carefully to balance this trade-off.

every cycle. Theorem 3 provides a way to derive discretized safe switching conditions based on Lemma 1.

The key idea in Theorem 3 is to keep track of the accumulated stress during the past $T^{win} - (T_c + T_s)$ time window. Next, we compute the maximum of the sum for the stress that could be accumulated over the future interval of length $T_c + T_s$. This represents the worst-case accumulated stress from the current time until SC can bring back the system inside the stability region. Intuitively, if in total the worst-case future stress and the accumulated (past) stress are below the limit of the overrun constraint, the condition is satisfied. Otherwise, RTR will need to trigger a switch to SC.

Theorem 3. *Assuming that $T^{win} > T_s + T_c$, an overrun constraint will always remain satisfied under the control of CC, if at every control interval j the following condition holds:*

$$\begin{aligned}\sum_{p=j-(R+(M-1)\cdot L)}^{j} & (\text{MaxSumStress}_x([p \cdot T_c, (p+1) \cdot T_c])) \\ & + \text{MaxSumStress}_x([(j+1) \cdot T_c, T_s]) \leq \alpha C\end{aligned} \tag{14}$$

Here, $\text{MaxSumStress}_x([t_1, t_2])$ is a function defined to over-approximate the sum of stress over the trajectory in the interval of $[t_1, t_2]$. It is defined as:

$$\int_{\tau=t_1}^{t_2} \text{Stress}(x(\tau)) \cdot d\tau \leq \text{MaxSumStress}_x([t_1, t_2])$$

Additionally, we define L, Q, and R as follows:

$$L = \left\lceil \frac{T^{win} - (T_s + T_c)}{(M-1)T_c} \right\rceil, Q = \left\lfloor \frac{j}{L} \right\rfloor, R = j - QL$$

Here M is the length of the array we use to keep track of past stress, in which an element stores the cumulative stress over L consecutive control cycles.

Proof. The current time is $t = jT_c$. First, we show that Condition 14 provides an over-approximation of the accumulated stress over the considered time interval ΔT. From the definition MaxSumStress_x we have:

$$\begin{aligned} & \sum_{p=j-(R+(M-1)\cdot L)}^{j} \left(\int_{p \cdot T_c}^{(p+1)\cdot T_c} \text{Stress}(x(\tau)) \cdot d\tau \right) \\ & \qquad + \int_{(j+1)\cdot T_c}^{(j+1)\cdot T_c + T_s} \text{Stress}(x(\tau)) \cdot d\tau \\ \leq & \sum_{p=j-(R+(M-1)\cdot L)}^{j} (\text{MaxSumStress}_x([p \cdot T_c, (p+1) \cdot T_c])) \\ & \qquad + \text{MaxSumStress}_x([(j+1) \cdot T_c, T_s]) \leq \alpha C. \end{aligned}$$

Next, we prove that the considered time interval ΔT is always longer than T^{win}. In fact:

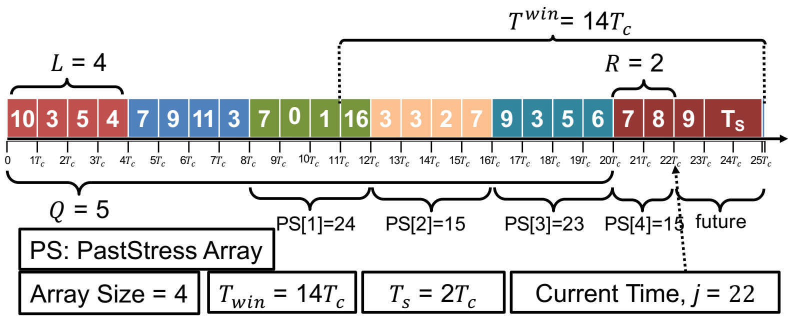
$$\begin{aligned}\Delta T =& ((j) - (j - (R + (M-1)L) + 1))T_c + T_s \\ =& (R + (M-1)L + 1)T_c + T_s \geq T^{win} + RT_c \geq T^{win}\end{aligned}$$

It follows that the total accumulated stress during ΔT is less than αC. Hence, during any interval of size T^{win} the accumulated stress is below the limit of the considered overrun constraint. Thus, Lemma 1 holds and the proof follows. $\square$

ALGORITHM 1: RTR module execution flow

```
Algorithm RTRModule()
    set R = 0 and initialize all elements of array PastStress to MAXDOUBLE
    while true do
        state^{j-1} = readCurrentStateFromSensors /*At the beginning of j-1th control cycle*/
        CCcommand = Control Command applied at j-1th cycle
        state^j, MaxSumStress^{J-1} = Reach_{=T_c}(state^{j-1}, CCcommand)
        PastStress[M] += MaxSumStress^{J-1}
        SumOfPastStress = Sum all elements of PastStress
        reachCC, MaxSumStress^J = Reach_{≤T_c}(state^j, CC, SumOfPastStress)
        reachSCAtTs, - = Reach_{=T_s}(reachCC, SC)
        reachSCBeforeTs, sumStressUntilSettling = Reach_{≤T_s}(reachCC, SC)
        If R == L Then Shift PastStress to left; set R = 0 ; End
        /*At the end of j-1th control cycle*/
        If SumOfPastStress + MaxSumStress^J + sumStressUntilSettling < αC and
          reachCC ⊆ 𝒮 and reachSCAtTs ⊆ ℛ and reachSCBeforeTs ⊆ 𝒮 then Put CC in
          charge; else switch to SC; End
        R++ and Update WD timer.
    end
```

Figure 2 illustrates how Theorem 3 can be applied to the system. In this example, the goal is to guarantee that the accumulated stress over a window of $14T_c$ will not exceed a fixed threshold. The memory array used to keep track of the past stress is 4 blocks in size. From the definition of L in III.2, each memory block stores the total stress over 4 cycles to cover the whole window. The current time is $22T_c$, (22^{nd} cycle). The future time required for the system to settle in the region $\mathcal{R}$ is $T_c + T_s = T_c + 2T_c = 3T_c$. The accumulated stress over the time window of length $14T_c$ must be smaller than the permitted threshold (condition of Theorem 3). Hence, we need to compute the stress over the next $3T_c$ plus the stress over the past $11T_c$ time units. The sum of the values from A[1] to A[4] in memory provides the accumulated stress over the past 14 cycles (i.e. from $8T_c$ to $22T_c$), which is an over-approximation of what required (11 cycles). As long as the total between future stress and past stress remains under the limit, the considered overrun constraint is met.



T^{win}= $14T_c$
L = 4
R = 2
10 3 5 4 7 9 11 3 7 0 1 16 3 3 2 7 9 3 5 6 7 8 9 T_s
0 $1T_c$ $2T_c$ $3T_c$ $4T_c$ $5T_c$ $6T_c$ $7T_c$ $8T_c$ $9T_c$ $10T_c$ $11T_c$ $12T_c$ $13T_c$ $14T_c$ $15T_c$ $16T_c$ $17T_c$ $18T_c$ $19T_c$ $20T_c$ $21T_c$ $22T_c$ $23T_c$ $24T_c$ $25T_c$
Q = 5
PS[1]=24
PS[2]=15
PS[3]=23
PS[4]=15
future
PS: PastStress Array
Array Size = 4
T_{win} = $14T_c$
T_s = $2T_c$
Current Time, j = 22

Figure 2: An example to clarify Theorem 3. Each element shows the maximum cumulative stress within that control cycle.

III.3 RTR Module Implementation

The structure of the main loop of the RTR module is presented in Algorithm 1. For simplicity, only a single overrun constraint is considered. Each iteration of the while loop in Algorithm 1 performs the required computation for a single control cycle.

Here j is the cycle for which the RTR module needs to determine whether the switching conditions hold or not. The decision of the RTR module needs to be ready at the beginning of the cycle j. Hence, those conditions need to be assessed during the previous cycle, $j-1$. Algorithm 1 uses the state at the beginning of $(j-1)^{th}$ cycle (line 4), as well as the exact control command generated at the beginning of $(j-1)^{th}$ cycle (line 5) to find the $\texttt{state}^j$ which is the reachable set of states at the beginning of j (line 6). `PastStress` is an array of size M to keep track of the stress during the past cycles. In each iteration, before evaluating the conditions for cycle j, the `MaxSumStress` is computed over the $(j-1)^{th}$ cycle and added to the last element of `PastStress` (line 7).

Next, the reachable sets required to check Conditions 1-3 in Theorem 2 for cycle j and also the accumulated stress during the j^{th} cycle and until the settling time are computed (lines 9,10, and 11). Finally, in line 14 the algorithm checks weather all the constraints are met. In case they are all met, control commands from CC are forwarded to the actuators, otherwise SC is selected to control the system.

III.4 Reach Function

ALGORITHM 2: Modified `Reach` algorithm that also calculates MaxSumStress

```
1  Algorithm Reach(currentBox, Controller)
2      MaxSumStress= 0;
3      while reachTimeRemaining > 0 do
4          Box [] nebs = constructNeighborhoods(currentBox,reachTimeStep);
5          crossReachTime= minCrossReachTime (nebs);
6          advanceReachTime=min (crossReachTime,reachTimeRemaining);
7          currentBox=advanceBox (nebs,advanceReachTime);
8          MaxSumStress += StressMax(currentBox)×advanceReachTime;
9          reachTimeRemaining -= advanceReachTime;
10     end
11     return currentBox, MaxSumStress
```

The function `Reach`, used in Algorithm 1, implements a modification of the face-lifting real-time reachability algorithm originally proposed in [5] to compute the maximum cumulative stress (`MaxSumStress`) over a given time, in addition to the reachable set of states. Implementation of `Reach` is described in Algorithm 2.

In this technique, a set of states is tracked at specific snapshots in time. Initially, at time 0, the set of states being tracked is set to the initial states of the system. Time is iteratively advanced (by some time step) from the initial states, which changes the set of states being tracked, until the desired final time is reached. The time needed to execute this algorithm is deterministic and adjustable. And, if there is more time left, the accuracy can be improved.

The representation used to track the set of states for real-time reachability is a single n-dimensional hyper-rectangle (box), where n is the number of state space variables. The way the set of states changes over time is only based on the derivatives *near the boundaries of the tracked set of states*. Since all the state trajectories are continuous, the trajectories cannot leave the tracked set of states without first passing through the boundary, and therefore reasoning about the behavior at the boundaries is a sound way to bound all possible trajectories. Figure 3 depicts an example of a region and the neighborhoods around it.

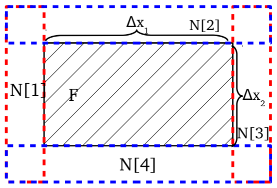


Figure 3: Example: Neighborhoods ($N[1]$ to $N[4]$) are constructed around states in region F.

A set of *neighborhoods*, $N[i]$s, is constructed around each $face_i$ of the tracked states, F with an initial width (line 4). Next, the maximum derivative in the outward direction, d_i^{max}, inside each $N[i]$ is computed and, 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 (line 6). Finally, every face is advanced to $face_i + d_i^{max} \times t^a$, (the face is 'lifted' at the maximum derivative. line 7)). For further details on inward neighborhood versus outward neighborhoods, and the choosing of neighborhood widths and time steps refer to [5].

In order to compute the accumulated stress, we require the system designer to provide a function, `StressMax`, for each overrun constraint that returns the maximum value of Stress over a hyper-rectangular region (box) of states. Notice that this does not require complex computations. For instance, if Stress is a convex function, `StressMax` can evaluate Stress at the corners of the box and find the maximum. In every intermediate step of `Reach` function, the maximum of the Stress over the intermediate reachable box is found using `StressMax`. The upper-bound of accumulated stress is found by multiplying the maximum of stress in the length of the intermediate time interval (line 8).

Remark 1. *After a restart, the RTR module loses its memory. In order to ensure that Overrun Constraints are not violated, elements of the array `PastStress[M]` are initialized to `MaxDouble`. As a result, immediately after the restart is completed, RTR would put SC in charge for a few cycles until a recorded history is available and then the control can be safely passed to the CC.*

VI. An Alternative Architecture

The design method described so far requires that the task of the SC to be schedulable in spite of resets, as discussed in Section I. It follows that the rate at which such a task can operate depends not only on the frequency of faults, but also on the time required for a full restart. Table 1 in Section VII reports the reset time for three different commercial platforms. Let us consider the fastest reboot time in the table (45 ms), and assume that only one SC task is present with a WCET of 1 ms. Under these assumptions, schedulability of the SC task in presence of resets can only be satisfied at a rate lower than or equal to about 20 Hz. Having such a low frequency for SC (i.e. long T_c), makes it harder to satisfy the switching conditions of Theorem 2 and consequently, decreases the size of the region over which the system can safely operate. Depending on the system dynamics and the constraints, this region can become small and prevent the system from making progress.

A possible way to overcome this problem consists of computing the maximum time that a system can be left in open-loop, for instance with fixed actuators' outputs, without abandoning its safety region. In this way, one or more instances of critical tasks can be skipped after a reset without compromising the stability of the system. Such approach is out of the scope of this work and we plan to investigate its feasibility as a part of our future work.

Alternatively, additional hardware can be used to overcome the aforementioned issue. Specifically, it is possible to migrate the SC to a dedicated processing unit. The resulting architecture is depicted in Figure 4. This prevents the SC from being affected by restarts, as long as control is automatically transferred to the SC while the main computation unit undergoes a restart. This architecture is similar to what was proposed in our previous work, System-level Simplex [7]. System-level Simplex, however, exploits this hardware redundancy to protect the system against the faults that may occur in the underlying layers of system (such as the OS) and utilizing it to recover from faults is not fully investigated and evaluated.

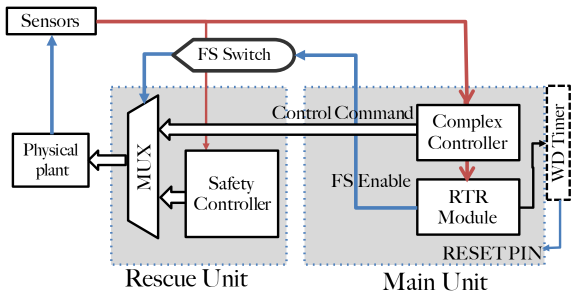


Figure 4: Restartable Simplex architecture with dedicated safety controller hardware unit.

The mechanism to reliably switch between controllers is enabled by a *fail-safe switch* (FS). The input to the FS is the `FSEnable` signal, which can assume three values: *active low*, *active high*, or *invalid*. The FS selects control commands from the CC if it receives an active high `FSEnable` signal. Conversely, if the input is active low or invalid, which is the case during the restart of the main unit, the FS selects the SC as the control unit. This approach constitutes a reliable way to immediately put the safety controller in charge when the main unit is undergoing a restart. Intuitively, all the results presented in Section V are also valid in this architecture. Moreover, the system remains stable no matter what is the frequency or the length of resets, because the SC is able to maintain stability for arbitrarily long time intervals.

VII. Evaluation

In this section, we present an evaluation of the proposed architecture. We demonstrate robustness against faults and timely recovery of a RC helicopter system through a combined testbed evaluation and simulation-based system modelling. Since the control rate for the helicopter needs to run at 50 Hz, we deployed the architecture described in Section VI.

I. Model Development and SC Design

For the helicopter system, the goal is to design a SC that can maintain the following constraints. The safety constraint is to maintain a minimum altitude of 10 meters from the ground (i.e. not crash). The overrun constraint for this system is on the amount of time that the vertical velocity (velocity over z-axis) is higher than 3 m/s seconds. This velocity shall not be maintained for more than 15 seconds within any time window of length 60 seconds:

$$\text{Stress}(\dot{z}) = \begin{cases} 1 & \dot{z} > 3 \\ 0 & \dot{z} \leq 3 \end{cases}, \forall t; \int_{t}^{t+60} \text{Stress}(\dot{z}(\tau))d\tau \leq 15$$

I.1 Model Dynamics

Let's first describe the helicopter dynamics. We use a model obtained from the aerospace literature that has been proposed and utilized for helicopter controller design in [23–26]. In this model, control authority for a helicopter is obtained via lift generated by the main and tail rotors. The vector lift generated by a rotor disk lies along its axis of rotation and the tail rotor orientation is fixed. The main rotor lift is decomposed into three components $(-\omega_2, \omega_1, u) \in \mathcal{A}$ and the tail rotor force into one component $(0, -\omega_3, 0) \in \mathcal{A}$, all in the body-fixed frame. In this notation, u is the principal lift force source and $\omega = (\omega_1, \omega_2, \omega_3)$ are the three torque controls (the first two coming from the lateral components of the main rotor lift due to inclination of the main rotor disk). The classical 'yaw', 'pitch', and 'roll' Euler angles $\eta = (\phi, \theta, \psi)$ are being used in this model.

Matrix R represents the orientation of the helicopter:

$$R = \begin{pmatrix} c_\theta c_\phi & s_\psi s_\theta c_\phi - c_\psi s_\phi & c_\psi s_\theta c_\phi + s_\phi s_\phi \\ c_\theta s_\phi & s_\psi s_\theta s_\phi + c_p s i c_\phi & c_\psi s_\theta s_\phi s_\psi c_\phi \\ -s_\theta & s_\psi c_\theta & c_\psi c_\theta \end{pmatrix}$$

The full helicopter dynamics in the generalized coordinates (ξ, η) are derived from the Euler-Lagrange equations.

$$m\ddot{\xi} = R \begin{pmatrix} -\omega_2 \\ \omega_1 - \omega_3 \\ -u \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix} \tag{15}$$

$$\mathbb{I}\ddot{\eta} = -C(\eta, \dot{\eta})\dot{\eta} + \tau \tag{16}$$

here, $\xi = (x, y, z)$ and $C(\eta, \dot{\eta})$ is the coriolis matrix. Torque applied to the helicopter transforms into generalised forces on the Euler coordinates (ϕ, θ, ψ) via

$$\begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \end{pmatrix} = \frac{1}{r} \begin{pmatrix} -s_\theta & 0 & 1 \\ c_\theta s_\psi & c_\phi & 0 \\ c_\theta c_\psi & -s_\psi & 0 \end{pmatrix} \tau \tag{17}$$

where r denotes the offset of the main rotor hub from the center of the mass of the helicopter. Equation 15 can be re-written as the following:

$$m\ddot{\xi} = u \begin{pmatrix} -c_\psi s_\theta c_\phi - s_\psi s_\phi \\ -c_\psi s_\theta s_\phi + s_\psi c_\phi \\ -c_\psi c_\theta \end{pmatrix} + \begin{pmatrix} 0 \\ 0 \\ mg \end{pmatrix} + \begin{pmatrix} c_\theta c_\phi & (s_\psi s_\theta c_\phi - c_\psi s_\phi) \\ c_\theta c_\phi & (s_\psi s_\theta s_\phi + c_\psi c_\phi) \\ -s_\theta & s_\psi c_\theta \end{pmatrix} \begin{pmatrix} -\omega_2 \\ \omega_1 - \omega_3 \end{pmatrix} \tag{18}$$

I.2 Model Linearization

In order to design the SC using the steps explained in section II, the model needs to be expressed in a linear form. From equation 18, for the z-dynamics of the system we have

$$m\ddot{z} = -uc_\psi c_\theta + mg + s_\theta \omega_2 + s_\psi c_\theta(\omega_1 - \omega_3) \tag{19}$$

In order to satisfy the safety constraints, we set the goal of the SC to stabilize the altitude ($\dot{z} = 0$) and to level the helicopter ($\phi = 0$, $\theta = 0$, $\psi = 0$). We set the linear domain such that the components of η are norm smaller than $\pi/3$. Now, in the neighbourhood of the equilibrium point, the last two terms of equation 19, that represent the small body force perturbations, are relatively small and can be neglected for the controller design. Thus, we have the following for the dynamics of the altitude.

$$m\ddot{z} = -uc_{\psi}c_{\theta} + mg \tag{20}$$

Now, we can use a control transformation to get to a linearized system. Consider the following control transformation:

$$u = m.\frac{\mu + g}{c\psi c_{\theta}}, \quad \tau = C(\eta, \dot{\eta})\dot{\eta} + \mathbb{I}.\hat{\tau} \tag{21}$$

By replacing 21 into the 15 and 20, the explicit forms of linearized system in the absence of the small body forces in equation 19 is

$$\begin{cases} \dot{z} = v_z, & \dot{v_z} = -\mu & \dot{\phi} = \gamma, & \dot{\gamma} = \hat{\tau}_{\phi} \\ \dot{\theta} = \omega, & \dot{\omega} = \hat{\tau}_{\theta} & \dot{\psi} = \lambda, & \dot{\lambda} = \hat{\tau}_{\psi} \end{cases} \tag{22}$$

Above equations can be represented as $\dot{x} = Ax + Bu$ where $x = [z, v_z, \phi, \gamma, \theta, \omega, \psi, \lambda]^T$ and $u = [\mu, \hat{\tau}_{\phi}, \hat{\tau}_{\theta}, \hat{\tau}_{\psi}]^T$. Now, we design the state feedback controller of $u = Kx$.

I.3 Safety and Overrun Constraints

The hard constraint on the minimum height can be expressed as a linear inequality as: $a_1 = [1/10, 0, 0, 0, 0, 0, 0, 0]^T$. For the overrun constraint, we consider a maneuverability coefficient $\alpha = 1$. Hence, Equation 10 can be written as

$$O = \{x | \text{Stress}(x) \leq 0\} \Rightarrow \forall x \in O, v_z = \dot{z} \leq 3$$

. It follows that linear inequality resulting from the above constraint is: $c_1 = [0, 1/3, 0, 0, 0, 0, 0, 0]^T$.

I.4 SC Design

The goal is to design a SC which is able to satisfy these constraints. First, the linear model used for the system is only valid in the region $|\phi|, |\theta|, |\psi| \leq \pi/3$. In order to ensure that the system remains in this region, we add the following constraints:

$$a_2 = -a_3 = [0, 0, 3/\pi, 0, 0, 0, 0, 0]^T$$
$$a_4 = -a_5 = [0, 0, 0, 0, 3/\pi, 0, 0, 0]^T$$
$$a_6 = -a_7 = [0, 0, 0, 0, 0, 0, 3/\pi, 0]^T$$

Denoting with u_{max} the saturation limit for the lift power of the main rotor, we have $\mu < c_{\psi}c_{\theta}u_{max} - 1$. For the region $|\psi|, |\theta| < \pi/3$ we can write $\mu \leq \cos(\pi/3)^2 u_{max} - 1 = u_{max}/4 - 1$. We define $\mu_{max} = u_{max}/4 - 1$. Thus we have

$$b_1 = -b_2 = [1/\mu_{max}, 0, 0, 0]^T$$

Finally, the stability region of the SC can be defined as:

$$\Gamma = \{x \mid a_i^T x < 1, i = 1, \ldots, 7, c_1^T x < 1, b_1^T u < 1, b_2^T u < 1\}$$

Given the linearized system of equations (see Equation 22) with the set of constraints Γ, we need to solve the minimization problem in Section II to obtain K and Q. Finally, the SC has dynamics $u = Kx$ and the stability region can be derived as $\mathcal{R} = \{S \mid x^T Q^{-1} x \leq 1\}$.

I.5 Model Validation and Identification

One of the advantages of the RTR algorithm is that the reachability of the system needs to be calculated only for a short interval of time in the future. As long as the model can predict the real behavior with a reasonable error for a short interval, RTR can be utilized.

First, we identified the parameters of the non-linear model from recorded flight data using the MATLAB grey-box model identification toolbox. In order to validate our model, we compared those flight traces against the simulated traces and measured the error. The error trend for the altitude is depicted in Figure 5. For each interval length, 100 simulations starting at random points of time were performed. Hence, the developed model is realistic enough for the purpose of evaluating the proposed methodology.

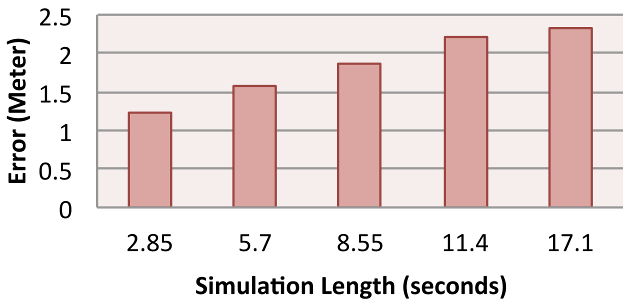


Figure 5: Simulation error for the altitude parameter.

II. Restarting in Action

The goal of this experiment is to demonstrate that the proposed switching and recovery method is applicable to a *real* safety critical CPS. We used a minimal implementation of the design with SC on a dedicated processor, a remote operator as CC, and a manual signal as the source of restart. Our testbed consists of an Align T-Rex 450 radio-controlled helicopter equipped with a ArduPilot APM 2.6 board [27] as the main unit and a Bavarian Demon board [28] as the rescue unit[4]. A Futaba FSU-2 [29] is utilized as the failsafe switch (FS).

In this test, during normal operation, the restart module generates a valid PWM signal, which instructs the FS to forward CC-generated commands. When the restart command from the ground control is received, an interrupt is triggered on the main unit that initiates a reset. During the reset, the PWM value in input to the FS becomes invalid, triggering a switch to the rescue unit. After the restart is completed, the main unit outputs a valid PWM pulse again, and control is handed back to the CC. The trace recorded from the helicopter during the flight is depicted in Figure 6. The first and second graphs, depict the altitude and the level angles of the helicopter, respectively. The third graph shows the time frame during which the SC was active. Even though the time required to restart the APM was only 85 ms, we manually forced the system to stay longer in the booting mode for evaluation purposes. The video recorded for this experiment can be found at [30].

III. Evaluation with the Simulated Model

For further analysis and testing of situations that are difficult to implement on the real system, we conducted our analysis using the validated model.

[4]It is important to mention that the Bavarian Demon board is not a fully programmable board but it is essentially a tunable PID controller. This unit can easily be replaced with a PID controller implemented on a general purpose micro-controller.

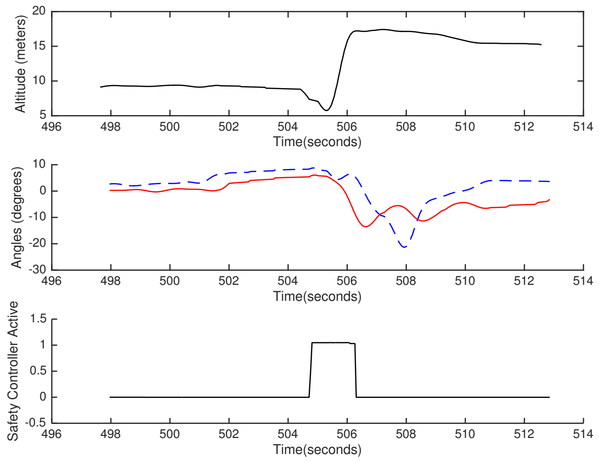


Figure 6: Altitude and the level angles of the helicopter during an in-flight restart

III.1 Progress Analysis

Restarting the platform, impacts the progress of the system towards the CC goal. The two parameters that determine the amount of impact are (i) the frequency of restarts and (ii) the time required to complete the restart. Figure 7, depicts the normalized comparison for various restart intervals and reboot lengths for a helicopter system where the CC is designed to keep the helicopter in a fixed altitude and with a fixed forward velocity. As seen in Figure 7, cases with a small ratio of reboot length to restart interval have an almost negligible progress slowdown.

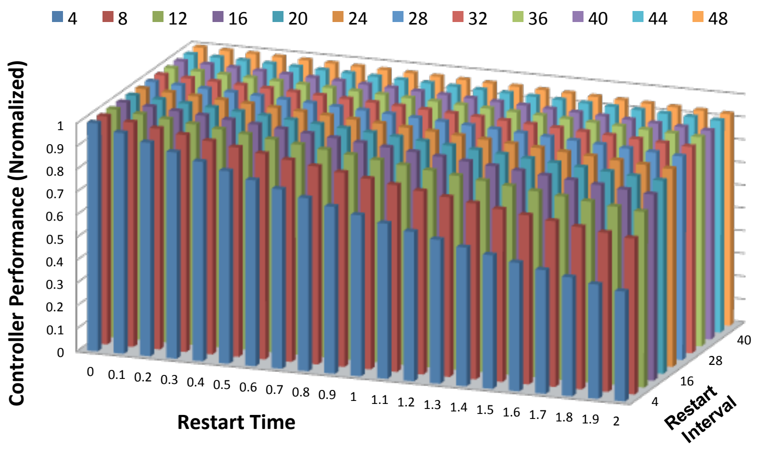


Figure 7: Impact of the restart interval and restart length on the control performance of system.

Next, we measured the time requirements for restarting three embedded platforms used in various applications: Freescale MPC564xL board (designed for automotive applications), Ardupilot APM 2.6 (commonly used in low-end UAVs) and the Intel Edison board (designed to target Internet-of-Things applications). The restart times measured for these platforms are presented in Table 1.

The conclusion that arises from the results in Figure 7 and Table 1, is that the impact of restarts on the progress of an embedded system with a typical fault rate can remain negligible. It follows that, if specific properties about the state of the system can be inferred after a reset, controlled periodic resets could also be introduced as a low-overhead strategy to "refresh" a live CPS and

Platform Name	OS Type	Restart Time	
Freescale MPC564xL	ERIKA RTOS	45 ms	
Ardupilot APM 2.6	ArdOS	80 ms	
Intel Edison	Yocto Linux	2031 ms	

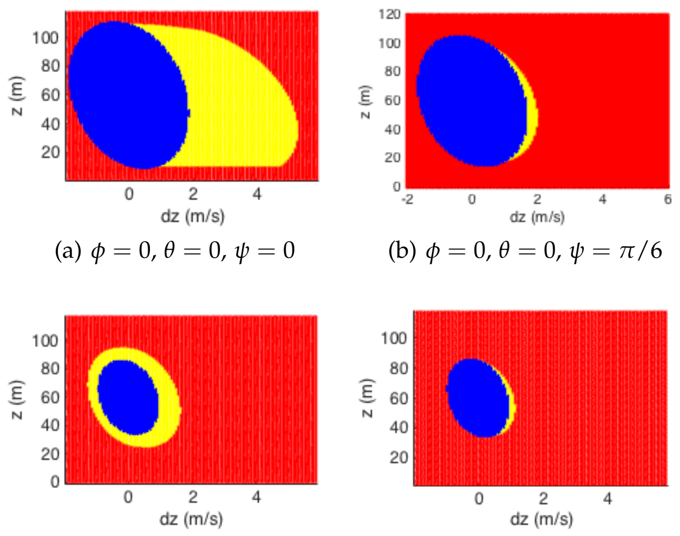
Table 1: Time required for full system restart

preventing the occurrence of unexpected faults.

III.2 Stabilizable Region Comparison

In this experiment, we compare the size of the operational region of the helicopter system under the original LMI-based Simplex [1–4], face-lifting real-time reachability [5] and our proposed modified RTR.

Since LMI-Simplex and RTR techniques cannot provide guarantees on the overrun constraints, in this experiment, only the hard constraints are considered. The projection of the stabilizable region, for z and $\dot{z}$ is shown in Figure 8. In order to demonstrate an adverse case behavior, we have changed the projection plane such that we can observe the behavior near the boundaries of the stability region. In Figure 8(a), 8(b), 8(c) and 8(d) the level angles of helicopter were increased, which resulted in a reduction in the size of the stability region. As seen in the figures, when only the hard constraints of the system are considered, the obtained stability region via face-lifting real-time reachability and our modified RTR are identical. These figures, highlight the benefit of using real-time reachability and modified real-time reachability by the larger provably safe recoverable region (yellow).



(a) $\phi = 0, \theta = 0, \psi = 0$ (b) $\phi = 0, \theta = 0, \psi = \pi/6$

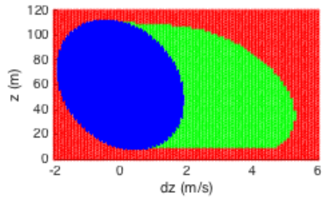
(c) $\phi = 0, \theta = \pi/6, \psi = \pi/6$ (d) $\phi = \pi/6, \theta = \pi/6, \psi = \pi/6$

Figure 8: Projection of stabilization region. Blue: LMI-Simplex; yellow: RTR and modified RTR; and red: unrecoverable. In all the figures we have $\dot{\phi} = 0, \dot{\theta} = 0, \dot{\theta} = 0$.

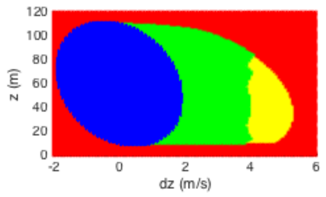
III.3 Modified RTR with Overrun Constraints

Next, we demonstrate that providing further guarantees on the overrun constraints can limit the operational region of system. The overrun constraint considered here was formulated in Section I.

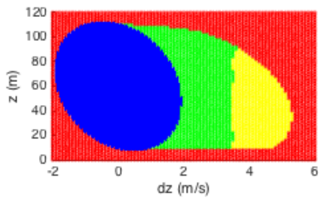
Whether the overrun constraints is satisfied depends not only on the current state, but also on the trajectory followed by the system. Therefore, we project the stabilization region under increasing levels of accumulated stress over the past time window. Figures 9(a) to 9(d) depict the stability regions of the system from a given state for different values of accumulated stress. From left to right, top to bottom, the considered amount of accumulated stress is 12, 14, 14.5 and 15. It can be noted that in Figure 9(a) classic face-lifting RTR and our modified RTR produce an identical region because 3 seconds are sufficient for the SC to reduce $v_x = \dot{z}$ below the 3 m/s threshold. As the accumulated stress increases, the size of the green region decreases. Finally, in Figure 9(d), where the accumulated stress is already 15, the $\dot{z} = 3$ boundary cannot be crossed at any time.



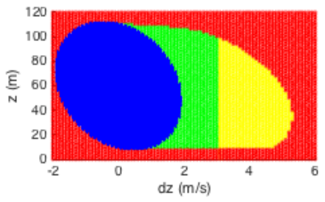
(a) Cumulative stress = 12



(b) Cumulative stress = 14



(c) Cumulative stress = 14.5



(d) Cumulative stress= 15

Figure 9: Blue: LMI-Simplex; yellow: RTR; green: modified RTR; and red: unrecoverable. Yellow: only hard constraints satisfied. Green: both overrun and hard constraints satisfied.

VIII. Conclusions

In this paper, we enable continuously-actuated CPS using Simplex design to (i) to recover from the faults in a timely manner by restarting at runtime. Moreover, (ii) we propose a novel technique to guarantee a more complex category of safety constraints with a temporal aspect. And, (iii) through a proof-of-concept minimal implementation on a small unmanned helicopter and simulation-based system modeling, we show the effectiveness of proposed recovery architecture under the assumed fault model.

References

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

[2] ——, "Using simplicity to control complexity." IEEE Software, 2001, pp. 20–28.

[3] 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.

[4] 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.

[5] 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.

[6] S. Z. Bak, "Industrial application of the system-level simplex architecture for real-time embedded system safety," 2009.

[7] 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.

[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] ——, "Crash-only software," 2003.

[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. 3–3.

[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] C. Zimmer and F. Mueller, "Fault resilient real-time design for noc architectures," in *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on*. IEEE, 2012, pp. 75–84.

[16] E-Flite Inc., "Power 10 brushless outrunner motor datasheet," http://www.e-fliterc.com/ProdInfo/Files/EFLPower10OutrunnerInstructions.pdf, accessed: 2015-10-10.

[17] B. Venkataraman, B. Godsey, W. Premerlani, E. Shulman, M. Thaku, and R. Midence, "Fundamentals of a motor thermal model and its applications in motor protection," in *Protective Relay Engineers, 2005 58th Annual Conference for*. IEEE, 2005, pp. 127–144.

[18] H. Kopetz and G. Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, 2003.

[19] H. Kopetz, *On the fault hypothesis for a safety-critical real-time system*. Springer, 2004.

[20] 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.

[21] L. George, N. Rivierre, M. Spuri, and I. national de recherche en informatique et en automatique (France), *Preemptive and Non-preemptive Real-time Uniprocessor Scheduling*, ser. Rapports de recherche. INRIA Centre, 1996.

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

[23] F. Mazenc, R. Mahony, and R. Lozano, "Forwarding control of scale model autonomous helicopter: a lyapunov control design," in *Decision and Control, 2003. Proceedings. 42nd IEEE Conference on*, vol. 4, Dec 2003, pp. 3960–3965 vol.4.

[24] O. Shakernia, Y. Ma, T. J. Koo, and S. Sastry, "Landing an unmanned air vehicle: Vision based motion estimation and nonlinear control," *Asian journal of control*, vol. 1, no. 3, pp. 128–145, 1999.

[25] E. Licéaga-Castrol, "A liouvillian systems approach for the trajectory planning-based control of helicopter models," *Int. J. Robust Nonlinear Contrul*, vol. 10, pp. 301–320, 2000.

[26] I. A. Raptis and K. P. Valavanis, *Linear and nonlinear control of small-scale unmanned helicopters*. Springer Science & Business Media, 2010, vol. 45.

[27] 3D Robotics, "Ardupilot apm2.6," http://3drobotics.com/kb/apm-2-6/, accessed: 2015-9-24.

[28] CAPTRON Electronic GmbH, "Bavarian demon datasheet, 3x/3xs series," http://www.bavariandemon.com/fileadmin/user_upload/downloads/bavarianDEMON-Instructions-3SX-3X_V6.1.pdf, accessed: 2015-9-24.

[29] Futaba Inc., "Fsu2 instruction manual," http://manuals.hobbico.com/fut/fsu2-manual.pdf, accessed: 2015-10-11.

[30] Fardin Abdi, "Flight demo video," https://youtu.be/tHcJUvBKd8Q.