# Soter: Programming Safe Robotics System using Runtime Assurance

*Ankush Desai*
*Shromona Ghosh*
*Sanjit A. Seshia*
*Natarajan Shankar*
*Ashish Tiwari*

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2018-127
http://www2.eecs.berkeley.edu/Pubs/TechRpts/2018/EECS-2018-127.html

August 21, 2018

Acknowledgement

The work of the first and third authors was supported in part by the TerraSwarm Research Center, one of six centers supported by the STARnet phase of the Focus Center Research Program (FCRP) a Semiconductor Research Corporation program sponsored by MARCO and DARPA, by the DARPA BRASS and Assured Autonomy programs, and by Toyota under the iCyPhy center.

# SOTER: Programming Safe Robotics System using Runtime Assurance

Ankush Desai°, Shromona Ghosh°, Sanjit A. Seshia°, Natarajan Shankar*, Ashish Tiwari*
° University of California, Berkeley, CA, USA
* SRI International, Menlo Park, CA, USA

## ABSTRACT

Autonomous robots increasingly depend on third-party off-the-shelf components and complex machine-learning techniques. This trend makes it challenging to provide strong design-time certification of correct operation. To address this challenge, we present SOTER[1], a programming framework that integrates the core principles of runtime assurance to enable the use of uncertified controllers, while still providing safety guarantees.

Runtime Assurance (RTA) is an approach used for safety-critical systems where design-time analysis is coupled with run-time techniques to switch between unverified advanced controllers and verified simple controllers. In this paper, we present a runtime assurance programming framework for modular design of provably-safe robotics software. SOTER provides language primitives to declaratively construct a RTA module consisting of an advanced controller (untrusted), a safe controller (trusted), and the desired safety specification ($\phi$). If the RTA module is well formed then the framework provides a *formal guarantee* that it satisfies property $\phi$. The compiler generates code for monitoring system state and switching control between the advanced and safe controller in order to guarantee $\phi$. SOTER allows complex systems to be constructed through the composition of RTA modules.

To demonstrate the efficacy of our framework, we consider a real-world case-study of building a safe drone surveillance system. Our experiments both in simulation and on actual drones show that SOTER-enabled RTA ensures safety of the system, including when untrusted third-party components have bugs or deviate from the desired behavior.

.

## 1 INTRODUCTION

Robotic systems are increasingly playing diverse and safety-critical roles in society, including delivery systems, surveillance, and personal transportation. This drive towards autonomy is also leading to ever-increasing levels of complexity, including integration of advanced data-driven, machine-learning components. However, advances in formal verification and systematic testing have yet to catch up with this increased complexity. Moreover, the dependence of robotic systems on third-party off-the-shelf components and machine-learning techniques is predicted to increase. This has resulted in a widening gap between the complexity of systems being deployed and those that can be certified for safety and correctness via formal verification.

One approach to bridging this gap is to leverage techniques for *run-time assurance*, where the results of design-time verification can be leveraged in building a system that monitors itself and its environment at run time, and switches to a provably safe operating mode, potentially at lower performace and sacrificing certain non-critical objectives. A prominent example of a Run-Time Assurance (RTA) framework is the *Simplex Architecture* [1], which has been used for building provably-correct safety-critical avionics [2, 3], robotics [4] and cyber-physical systems [5–7]. The Simplex architecture combines an uncertified advanced controller (AC) with a certified for correctness safe controller (SC) and a decision module (DM), where the role of DM is to switch between AC and SC such that the overall system remains safe. However, most of these prior applications of RTA do not provide high-level *language support* for constructing provably-safe RTA systems in a *modular* fashion while designing for the *timing and communication behavior* of such systems. Prior techniques either apply RTA to a single untrusted component in the system or wrap the large monolithic system into a single instance of Simplex which makes the design and verification of the corresponding SC and DM difficult or infeasible. Schierman *et al.* [2] investigated how the RTA framework can be used at different levels of the software stack of an unmanned aircraft system. In a more recent work [8], Schierman *et. al.* proposed a component-based simplex architecture (CBSA) that combines assume-guarantee contracts with RTA for assuring the runtime safety of component-based cyber-physical systems. However, in order to ease the construction of RTA systems, there is a need for a general programming framework for building provably-safe robotic software systems with run-time assurance that also considers implementation aspects such as timing and communication.

In this paper, we seek to address this need using SOTER, a programming framework for building safe robotics systems using runtime assurance. A SOTER program is a collection of periodic processes, termed *nodes*, that interact with each other using a publish-subscribe model of communication (which is popular in robotics, e.g., [9]). An RTA module in SOTER consists of an advanced controller node, a safe controller node and a safety specification; if the module is well-formed then the framework provides a guarantee that the system satisfies the safety specification. SOTER allows programmers to declaratively construct an RTA module with specified timing behavior, combining provably-safe operation with the feature of using the AC whenever safe so as to achieve good performance. Our evaluation demonstrates that SOTER is effective at achieving this blend of safety and performance.

SOTER incorporates constructs for decomposing the design and verification of the overall RTA system into that for individual RTA modules while retaining guarantees of safety for the overall composite system. SOTER includes a compiler that generates the DM node that implements the switching logic, and which generates C

[1] SOTER: Greek god personifying the spirit of safety and deliverance from harm.

*Draft, August 2018,*
2018.

code to be executed on common robotics software platforms such as Robot Operating System (ROS) [9] and MavLink [10].

We evaluate the efficacy of the SOTER framework by building a safe autonomous drone surveillance system. We show that SOTER can be used to build a complex robotics software stack consisting of both third-party untrusted components and complex machine learning modules, and still provide system-wide correctness guarantees. The generated code for the robotics software has been tested both on an actual drone platform (the 3DR [11] drone) and in simulation (using the ROS/Gazebo [12] and OpenAI Gym [13]). Our results demonstrate that the RTA-protected software stack built using SOTER can ensure the safety of the drone both when using unsafe third-party controllers and in the presence of bugs introduced using fault-injection in the advanced controller.

To summarize, our paper makes the following novel contributions:

1. A programming framework for a Simplex-based run-time assurance system that provides language primitives for the modular design of safe robotics systems (Sec. 3);
2. A theoretical formalism based on computing reachable sets that keeps the system provably safe while also maintaining smooth switching behavior between the safe and advanced controllers (Sec. 4), and
3. Experimental results in simulation and on real drone platforms demonstrating how SOTER can be used for guaranteeing correctness of a system even in the presence of untrusted or unverified components (Sec. 5).

## 2 OVERVIEW

In this section, we first provide a brief overview of the classic run-time assurance architecture. Next, we present our robotics case study, an autonomous drone surveillance system. Finally, we provide an overview of how our RTA framework can be used to guarantee the desired safety invariants of the surveillance system.

### 2.1 Runtime Assurance Architecture

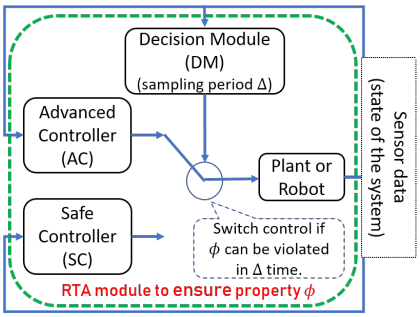



**Figure 1: RTA Architecture**

Figure 1 presents the RTA architecture (similar to Simplex [1]) consisting of three sub-components: **(1)** The *advanced controller* (AC) that controls the robot under nominal operating conditions, and is designed to achieve *high-performance* with respect to specialized metrics (e.g., fuel economy, time). The AC is optimized for performance, and does not come with a certificate of safety. **(2)** The *safe controller* (SC) that can be pre-certified to keep the robot within a region of safe operation for the plant/robot. **(3)** The *decision module* (DM) which is pre-certified (or automatically synthesized to be correct) to periodically monitor the state of the plant and determine when to switch from AC to SC so that the system is guaranteed to stay within the safe region.

When the AC is in control of the system, the DM monitors (samples) the system state every $\Delta$ period to check whether the system can violate the desired safety specification ($\phi$) in time $\Delta$. If so, the DM switches control to the SC. We refer to the conditions under which DM switches from AC to SC as the *switching condition*.

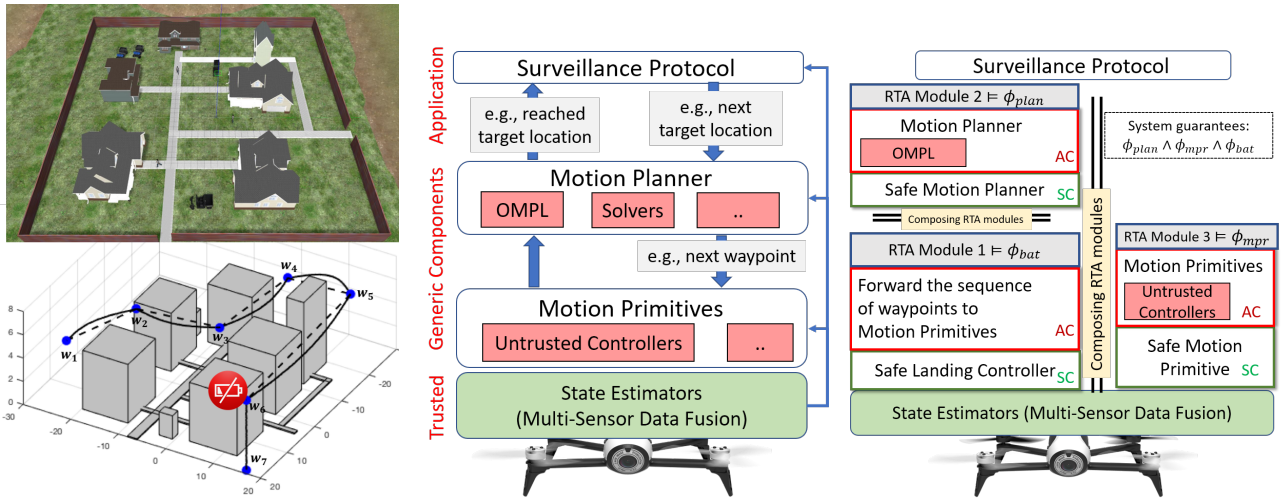
The set of all states of the system that satisfy the safety specification $\phi$ is denoted as the *safe set* ($\phi_{safe}$). Informally, a system protected by runtime assurance is guaranteed to satisfy the safety specification $\phi$, if the following properties hold: **(1)** There exists a subset of the safe set, denoted as the *recoverable* set, such that if the DM switches control to the SC in a recoverable state, then the system is guaranteed to remain in the safe set as long as the SC is in control. **(2)** If AC is in control and the switching condition evaluates to false, then the system is guaranteed to remain in the recoverable states until the next DM period, irrespective of the operations performed by the AC during that period.

### 2.2 Case Study: Drone Surveillance System

In this paper, we consider the problem of building a surveillance system where an autonomous drone must safely patrol a city. Figure 2a (top) presents a snapshot of the workspace from the Gazebo simulator [8]. Figure 2a (bottom) presents the obstacle map for the workspace, all obstacles (houses, cars, etc.) being static.

Figure 2b presents the software stack for the drone surveillance system. The application layer implements the surveillance protocol that ensures the application specific property ($\phi_{app}$), e.g., all surveillance points must be visited infinitely often. The generic components of the software stack consists of the motion planner and the motion primitives. The application generates the next target location for the drone. The motion planner computes a motion plan which is a sequence of waypoints from the current location to the target location. The waypoints $w_1 \ldots w_6$ in Figure 2a (bottom) represent one such motion plan generated by the planner and the dotted lines represent the reference trajectory for the drone. The motion primitives module on receiving the next waypoint generates the required low-level controls necessary to closely follow the reference trajectory. The trajectory in Figure 2a (bottom) represents the actual trajectory of the drone, which deviates from the reference trajectory because of the underlying dynamics, disturbances, etc.

In our drone surveillance case study, we would like the system to satisfy two critical safety invariants: **(1)** *Obstacle Avoidance* ($\phi_{obs}$): The drone must never collide with any obstacle. **(2)** *Battery Safety* ($\phi_{bat}$): The drone must never crash because of low battery, instead, when the battery is low it must land safely (e.g., in Figure 2a (bottom), low battery is detected at $w_6$ and the mission is aborted to land safely). $\phi_{obs}$ can be further decomposed into two parts $\phi_{obs} := \phi_{plan} \wedge \phi_{mpr}$; **(A)** *Safe Motion Planner* ($\phi_{plan}$): The motion planner must always generate a motion-plan such that the reference trajectory does not collide with any obstacle, **(B)** *Safe Motion Primitives* ($\phi_{mpr}$): When tracking the reference trajectory between any two waypoints generated by the motion planner, the controls generated by the motion primitives must ensure that the drone closely follows the trajectory and avoids collisions.




(a) Case Study: Drone Surveillance (b) Robotics Software Stack for Drone Surveillance (c) RTA-Protected Robotics Software Stack

**Figure 2: Runtime Assurance Protected Robotics Software Stack for Drone Surveillance using SOTER**

In practice, when implementing these modules, the programmer may use several uncertified components (red blocks in Figure 2b). The green blocks Figure 2 are trusted components. For example, implementing an on-the-fly motion planner may involve solving an optimization problem or using an efficient search technique that relies on a solver or a third-party library (e.g., OMPL [14]). Implementing planners that optimize for efficiency while satisfying safety properties (e.g., the generated motion plan must be feasible given the current battery charge) is challenging. Similarly, motion primitives are either designed using machine-learning techniques like Reinforcement Learning [15] or optimized for specific tasks without considering safety, or are off-the-shelf controllers provided by third parties. Ultimately, in the presence of such uncertified or hard to verify components (Figure 2b), no formal guarantees of safety can be provided.

***Motivation:*** The underlying problem that motivates the RTA framework is that it can be extremely difficult to design a single controller that is both safe and high-performance. Furthermore, the trend in robotics is towards *advanced*, data-driven controllers, such as neural networks, that usually do not come with safety guarantees. Our work is motivated by this need to enable the use of such advanced controllers while retaining strong guarantees of safety.

In this paper, we present SOTER that enables building *a reliable version* (Figure 2c) of the software stack *with runtime assurance* of safety invariant: $\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$. We decompose the stack into three components: **(1)** An RTA-protected motion planner that guarantees $\phi_{plan}$ (Section 5.3), **(2)** A battery-safety RTA module that guarantees $\phi_{bat}$ (Section 5.2), and **(3)** An RTA-protected motion primitive module that guarantees $\phi_{mpr}$ (Section 5.1). Our theory of well-formed RTA modules (Section 4) ensures that if the constructed modules are well-formed then they satisfy the desired safety invariant and their composition (Section 4.1) helps prove that the system-level specification is satisfied.

## 3 RUNTIME ASSURANCE MODULE

In this section, we present the generic RTA architecture implemented in the SOTER programming framework and formalize the semantics of an RTA-protected system.

### 3.1 Topics and Nodes

SOTER supports a publish-subscribe model of communication. A program in SOTER is a collection of periodic *nodes* communicating with each other by publishing on and subscribing to message topics. A node periodically listens to data published on certain topics, performs computation, and publishes computed results on certain other topics. A topic is an abstraction of a communication channel. Formally, a topic is a tuple $(e, v)$ consisting of a unique name $e \in \mathcal{T}$, where $\mathcal{T}$ is the universe of all topic names, and a value $v \in \mathcal{V}$, where $\mathcal{V}$ is the universe of all possible values that can be communicated using topic $e$. For simplicity of notation, assume all topics share the same set $\mathcal{V}$ of possible values.

Let $\mathcal{N}$ represent the set of names of all the nodes. Let $\mathcal{L}$ represent the set of all possible values the local state of any node could have during its execution. A *valuation* of a set $X \subseteq \mathcal{T}$ of topic names is a map from each topic name $x \in X$ to the value $v$ stored at the topic $(x, v)$. We write $\texttt{Vals}(X)$ for the valuations of set $X$.

A node in SOTER is a tuple $(N, I, O, T, C)$ where:

1. $N \in \mathcal{N}$ is the unique name of the node.
2. $I \subseteq \mathcal{T}$ is the set of names of all topics subscribed to by the node (inputs).
3. $O \subseteq \mathcal{T}$ is the set of names of all topics on which the node publishes (output). The output topics are disjoint from the set of input topics ($I \cap O = \emptyset$).
4. $T \subseteq \mathcal{L} \times (I \to \mathcal{V}) \times \mathcal{L} \times (O \to \mathcal{V})$ is the transition relation of the node. If $(l, \texttt{Vals}(I), l', \texttt{Vals}(O)) \in T$, then on the *input* (subscribed) topics valuation of $\texttt{Vals}(I)$, the local state of the node moves from $l$ to $l'$ and publishes on the *output* topics to update its valuation to $\texttt{Vals}(O)$.
5. $C = \{(N, t_0), (N, t_1), \dots\}$ is the time-table or calendar representing the times $t_0, t_1, \dots$ at which the node $N$ takes a step.

Intuitively, a node is essentially a periodic input-output state-transition system: at every time instant in its calendar, the node reads the values in its input topics, updates its local state, and publishes values on its output topics. Note that we are using the timeout-based discrete event simulation [16] to model the periodic real-time process as a standard transition system (more details in Section 3.3). Each node specifies, using a time-table, the fixed times at which it should be scheduled. For a periodic node with period $\delta$, the calendar will have entries $(N, t_0), (N, t_1), \ldots$ such that $t_{i+1} - t_i = \delta$ for all $i$.

We refer to the components of a node with name $N \in \mathcal{N}$ as $I(N), O(N), T(N)$ and $C(N)$ ($\delta(N)$ as its period) respectively.

**Assumptions.** The formalism presented in the rest of paper makes two simplifying assumptions: **(1)** each transition (code in the body of the node) can be executed instantaneously, and **(2)** the underlying system ensures that the transition system steps according to the schedule specified in the node calendar $C$ and for all nodes $t_0 = 0$.

```
1 type coord = (x: float, y: float, z: float);
2 topic NextWaypoint : coord;
3 ...
4 node MotionPrimitive
5 period 10;
6 subscribes LocalPosition, NextWaypoint;
7 publishes  Control;
8 { /* body */ }
```

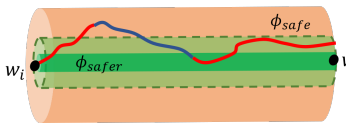
**Figure 3: Declaration of topics and nodes**

*Example 3.1 (Example of a Node in Soter).* Figure 3 presents the declaration of a node `MotionPrimitive` that subscribes to topics `LocalPosition` and `NextWaypoint`. Figure 3 (line 2) declares the topic `NextWaypoint` that can be used to communicate messages of type `coord` (coordinates in 3D space). `MotionPrimitive` node runs periodically every $10ms$ and publishes the next control action on the `Control` topic. The body of the node is a sequential program that performs local computations and publishes messages on the `Control` topic. For the exposition of this paper, we ignore the details of the body, it can be any sequential program that performs the required read-subscribed-topics → compute → publish step. □

## 3.2 RTA Module

Let $\mathcal{S}$ represent the state space of the system i.e., the set of all possible configurations of the system. We assume that the desired safety property is given in the form of a subset $\phi_{safe} \subseteq \mathcal{S}$ (*safe states*). The goal is to ensure using an RTA module that the system always stays inside the safe set $\phi_{safe}$.

The Figure below illustrates the behavior of a Soter based RTA-protected system. We want the drone to move from its current location $w_i$ to the target location $w_f$, and the desired safety property is that the drone must always remain inside the region $\phi_{safe}$ (outermost tube). Initially, the AC node is in control of the drone, and since it is not certified for correctness it may generate controls action that tries to push the drone outside the $\phi_{safe}$ region.




If the AC is wrapped inside an RTA module then the DM must detect this imminent danger and switch to the SC with enough time for SC to gain control over the drone. The SC must be certified to keep the drone inside $\phi_{safe}$ and also move it to a state in $\phi_{safer}$ where the DM evaluates that it is safe enough to return control back to the AC. In the rest of this section, we present the RTA module that enables such behavior and formalize the properties components of the RTA module (e.g., SC, $\phi_{safer}$, etc.) must satisfy to ensure property $\phi_{safe}$ holds and also allow control to return back to the AC to maximize performance.

Definition 3.1 (RTA Module). *An RTA module is a tuple* $(N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$ *where:* ***(1)*** $N_{ac} \in \mathcal{N}$ *is the advanced controller* (AC)*,* ***(2)*** $N_{sc} \in \mathcal{N}$ *is the safe controller* (SC)*,* ***(3)*** $N_{dm} \in \mathcal{N}$ *is the decision module* (DM)*,* ***(3)*** $\Delta \in \mathbb{R}^+$ *represents the period of the* DM *(*$\delta(N_{sc}) = \Delta$*),* ***(4)*** $\phi_{safe} \subseteq \mathcal{S}$ *is the desired safety property.* ***(5)*** $\phi_{safer} \subseteq \phi_{safe}$ *is a stronger safety property.*

```
1 if (mode=SC ∧ s_t ∈ φ_safer) mode = AC /*switch to AC*/
2 elseif (mode=AC ∧ Reach_M(s_t, *, 2Δ) ⊈ φ_safe) mode = SC /*
    switch to SC*/
3 else mode = mode /* No mode switch */
```

**Figure 4: Decision Module Switching Logic for Module M**

Given an RTA module $M$ as described above, Figure 4 presents the switching logic that sets the *mode* of the RTA module given the current state $s_t$ of the system. The DM node evaluates this switching logic once every $\Delta$ time units. When it runs, it first reads the current state $s_t$ and sets *mode* based on it. Note that the set $\phi_{safer}$ determines when it is safe to switch from $N_{sc}$ to $N_{ac}$. $Reach_M(s, *, t) \subseteq \mathcal{S}$ represents the set of all states reachable in time $[0, t]$ starting from the state $s$, using any non-deterministic controller. We formally define *Reach* in Section 3.3, informally, $Reach_M(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ checks that the system will remain inside $\phi_{safe}$ in the next $2\Delta$ time. This $2\Delta$ look ahead is used to determine when it is necessary to switch to using $N_{sc}$, in order to ensure that the $N_{sc}$ ($\delta(N_{sc}) \leq \Delta$) will be executed at least once before the system leaves $\phi_{safe}$. Soter automatically generates a unique DM node ($N_{dm}$) for each primitive RTA module declaration.

For an RTA module $(N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$, the decision module DM is the node $(N_{dm}, I_{dm}, \emptyset, T_{dm}, C_{dm})$ where:

1. The local state is a binary variable $mode : \{AC, SC\}$.
2. The topics subscribed by *DM* include the topics subscribed by either of the nodes; that is, $I(N_{ac}) \subseteq I_{dm}$ and $I(N_{sc}) \subseteq I_{dm}$.
3. The node does not publish on any topic. But it updates a global data structure that controls the outputs of AC and SC nodes (more details in Section 3.3).
4. If $(mode, \texttt{Vals}(I_{dm}), mode', \emptyset) \in T_{dm}$, then the local state moves from *mode* to *mode'* based on the logic in Figure 4.
5. $C_{dm} = \{(N_{dm}, t_0), (N_{dm}, t_1), \ldots\}$ where $\forall_i |t_i - t_{i+1}| = \Delta$ represents the time-table of the node.

We are implicitly assuming that the topics $I_{dm}$ read by the DM contain enough information to evaluate $\phi_{safe}$, $\phi_{safer}$, and the reachability computation.

## 3.3 Semantics of an RTA System

In Soter, a complex system is designed as a composition of RTA modules. An RTA system $S$ is a set of *composable* RTA modules.

***Composable modules.*** A set of modules $S = \{M_0, M_1, \ldots, M_n\}$ are composable if:

1. The nodes in all modules are disjoint, if $N^i_{ac}$, $N^i_{sc}$, and $N^i_{dm}$ represent the AC, SC and DM nodes of a module $M_i$ then, for all $i, j$ s.t. $i \neq j$, $\{N^i_{ac}, N^i_{sc}, N^i_{dm}\} \cap \{N^j_{ac}, N^j_{sc}, N^j_{dm}\} = \emptyset$.
2. The outputs of all modules are disjoint, for all $i, j$ s.t. $i \neq j$, $O(M_i) \cap O(M_j) = \emptyset$.

We use $\mathsf{dom}(X)$ to refer to the domain of map $X$ and $\mathsf{codom}(X)$ to refer to the codomain of $X$.

***RTA system attributes.*** Given an RTA system $S = \{M_0, \ldots, M_n\}$, its attributes (used for defining the operational semantics) can be inferred as follows:

1. $ACNodes \in \mathcal{N} \to \mathcal{N}$ is a map that binds a DM node $n$ to the particular AC node $ACNodes[n]$ it controls, i.e., if $M_i \in S$ then $(N^i_{dm}, N^i_{ac}) \in ACNodes$
2. $SCNodes \in \mathcal{N} \to \mathcal{N}$ is a map that binds a DM node $n$ to the particular SC node $SCNodes[n]$ it controls, i.e., if $M_i \in S$ then $(N^i_{dm}, N^i_{sc}) \in SCNodes$
3. $Nodes \subseteq \mathcal{N}$ represents the set of all nodes in the RTA system, $Nodes = \mathsf{dom}(ACNodes) \cup \mathsf{codom}(ACNodes) \cup \mathsf{codom}(SCNodes)$.
4. $OS \subseteq \mathcal{T}$ represents the set of outputs of the RTA system, $OS = \bigcup_{n \in Nodes} O(n)$.
5. $IS \subseteq \mathcal{T}$ represents the set of inputs of the RTA system (inputs provided by the environment), $IS = \bigcup_{n \in Nodes} I(n) \setminus OS$.
6. $CS$ represents the calendar or time-table of the RTA system, $CS = \bigcup_{n \in Nodes} C(n)$.

We refer to the attributes of a RTA system $S$ as $ACNodes(S)$, $SCNodes(S)$, $Nodes(S)$, $OS(S)$, $IS(S)$, and $CS(S)$ respectively.

We next present the semantics of an RTA system. Note that the semantics of an RTA module is the semantics of an RTA system where the system is a singleton set.

We use the timeout-based discrete event simulation model [16] for modeling the semantics of an RTA system. The calendar $CS$ stores the future times at which nodes in the RTA system must step. Using a variable $ct$ to store the current time and $FN$ to store the enabled nodes, we can model the real-time system as a discrete transition system.

***Configuration.*** A configuration of an RTA system is a tuple $(L, OE, ct, FN, Topics)$ where:

1. $L \in Nodes \to \mathcal{L}$ represents a map from a node to the local state of that node.
2. $OE \in \mathcal{N} \to \mathbb{B}$ represents a map from a node to a boolean value indicating whether the output of the node is enabled or disabled. This is used for deciding whether AC or SC should be in control. The domain of $OE$ is $\mathsf{codom}(ACNodes) \cup \mathsf{codom}(SCNodes)$.
3. $ct \in \mathbb{R}$ represents the current time.
4. $FN \subseteq \mathcal{N}$ represents the set of nodes that are remaining to be fired at time $ct$.
5. $Topics \in \mathcal{T} \to \mathcal{V}$ is a map from a topic name to the value stored at that topic, it represents the globally visible topics. If $X \subseteq \mathcal{T}$ then $Topics[X]$ represents a map from each $x \in X$ to $Topics[x]$.

The initial configuration of any RTA system is represented as $(L_0, OE_0, ct_0, FN_0, Topics_0)$ where: $L_0$ maps each node in its domain to default local state value $l_0$ if the node is AC or SC, otherwise, $mode = SC$ for the DM node, $OE_0$ maps each SC node to `true` and AC node to `false` (this is to ensure that each RTA module starts in SC mode), $ct_0 = 0$, $FN_0 = \emptyset$, and $Topics_0$ maps each topic name to its default value $v \in \mathcal{V}$.

We represent the operational semantics of a RTA system as a transition relation over its configurations (Figure 5).

ITE(x, y, z) represents if x then y else z

(ENVIRONMENT-INPUT)
$$\frac{e \in IS \qquad v \in \mathcal{V}}{(L, OE, ct, FN, Topics) \to (L, OE, ct, FN, Topics[e \mapsto v])}$$

(DISCRETE-TIME-PROGRESS-STEP)
$$\frac{FN = \emptyset^{(dt1)} \; ct' = \min(\{t \mid (x, t) \in CS, t > ct\})^{(dt2)} \; FN' = \{n \mid (n, ct') \in CS\}^{(dt3)}}{(L, OE, ct, FN, Topics) \to (L, OE, ct', FN', Topics)}$$

(DM-STEP)
$$\frac{\begin{array}{c} dm \in FN \qquad FN' = FN \setminus \{dm\} \\ dm \in \mathsf{dom}(ACNodes) \qquad (l, \{(\mathsf{STATE}, s_t)\}, l', \emptyset) \in T(dm) \qquad ac = ACNodes[dm] \\ sc = SCNodes[dm] \qquad \mathbf{ITE}(l' = AC, en = \mathsf{true}, en = \mathsf{false})^{(dm1)} \end{array}}{(L, OE, ct, FN, Topics) \to (L[dm \mapsto l'], OE[ac \mapsto en, sc \mapsto \neg en]^{(dm2)}, ct, FN', Topics)}$$

(AC-OR-SC-STEP)
$$\frac{\begin{array}{c} n \in FN \qquad FN' = FN \setminus \{n\} \\ n \notin \mathsf{dom}(ACNodes) \qquad in = Topics[I(n)] \qquad (l, in, l', out) \in T(n) \\ \mathbf{ITE}(OE[n], Topics' = out \cup Topics[\mathcal{T} \setminus \mathsf{dom}(out)], Topics' = Topics)^{(n1)} \end{array}}{(L, OE, ct, FN, Topics) \to (L[n \mapsto l'], OE, ct, FN', Topics')}$$

**Figure 5: Semantics of an RTA System**

There are two types of transitions: **(1)** discrete transitions that are instantaneous and hence does not change the current time, and **(2)** time-progress transitions that advance time when no discrete transition is enabled. DM-STEP and AC-OR-SC-STEP are the discrete transitions of the system. ENVIRONMENT-INPUT transitions are triggered by the environment and can happen at any time. It updates any of the input topics $e \in IS$ of the module to $(e, v)$. DISCRETE-TIME-PROGRESS-STEP represents the time-progress transitions that can be executed when no discrete transitions are enabled (dt1). It updates $ct$ to the next time at which a discrete transition must be executed (dt2). $FN$ is updated to the set of nodes that are enabled and must be executed(dt3) at the current time. DM-STEP represents the transition of any of the DM nodes in the module. The important operation performed by this transition is to enable or disable the outputs of the AC and SC node (dm2) based on its current *mode* (dm1). Finally, AC-OR-SC-STEP represents the step of any AC or SC node in the module. Note that the node updates the output topics only if its output is enabled (based on $OE(n)$ (n1)).

***Reachability.*** Note that the state space $\mathcal{S}$ of an RTA system is the set of all possible configurations. The set of all possible reachable states of an RTA system is a set of configurations that are reachable from the initial configuration using the transition system described in Figure 5. Note that since the environment transitions are nondeterministic, potentially many states are reachable even if the RTA modules are all deterministic.

Let $Reach_M(s, N_{sc}, t) \subseteq \mathcal{S}$ represent the set of all states of the RTA system $S$ reachable in time $[0, t]$ starting from the state $s$, using only the controller SC node $N_{sc}$ of the RTA module $M \in S$. In other words, instead of switching control between SC and AC of the RTA module $M$, the DM always keeps SC node in control. $Reach_M(s, *, t) \subseteq \mathcal{S}$ represents the set of all states of the RTA system $S$ reachable in time $[0, t]$ starting from the state $s$, using only a completely nondeterministic module instead of $M \in S$. In other

words, instead of module $M$, a module that generates nondeterministic values on the output topics of $M$ is used.

The notation *Reach* is naturally extended to a set of states: $Reach_M(\psi, x, t) = \bigcup_{s \in \psi} Reach_M(s, x, t)$ is the set of all states reachable in time $[0, t]$ when starting from a state $s \in \psi$ using $x$. Note that, $Reach_M(\psi, N_{sc}, t) \subseteq Reach_M(\psi, *, t)$.

## 4 CORRECTNESS OF AN RTA MODULE

Given a safe set $\phi_{safe}$, our goal is to prove that the RTA-protected system always stays inside this safe set. We need the RTA module to satisfy some additional conditions in order to prove its safety.

An RTA module $M = (N_{ac}, N_{sc}, N_{dm}, \Delta, \phi_{safe}, \phi_{safer})$ is said to be *well-formed* if its components satisfy the following properties:

1. (P1a) The maximum period of $N_{ac}$ and $N_{sc}$ is $\Delta$, i.e., $\delta(N_{dm}) = \Delta$, $\delta(N_{ac}) \leq \Delta$, and $\delta(N_{sc}) \leq \Delta$.
2. (P1b) The outputs of the $N_{ac}$ and $N_{sc}$ nodes must be same, i.e., $O(N_{ac}) = O(N_{sc})$.
3. The safe controller, $N_{sc}$, must satisfy the following properties:
   - (P2a) (*Safety*) $Reach_M(\phi_{safe}, N_{sc}, \infty) \subseteq \phi_{safe}$. This property ensures that if the system is in $\phi_{safe}$, then it will remain in that region as long as we use $N_{sc}$.
   - (P2b) (*Liveness*) For every state $s \in \phi_{safe}$, there exists a time $T$ such that for all $s' \in Reach_M(s, N_{sc}, T)$ , we have $Reach_M(s', N_{sc}, \Delta) \subseteq \phi_{safer}$. In words, from every state in $\phi_{safe}$, after some finite time the system is guaranteed to stay in $\phi_{safer}$ for at least $\Delta$ time.
4. (P3) $Reach_M(\phi_{safer}, *, 2\Delta) \subseteq \phi_{safe}$. This condition says that irrespective of the controller, if we start from a state in $\phi_{safer}$, we will continue to remain in $\phi_{safe}$ for $2\Delta$ time units. Note that this condition is stronger than the condition $\phi_{safer} \subseteq \phi_{safe}$.

Theorem 4.1 (**Runtime Assurance**). *For a well-formed RTA module $M$, let $\phi_{\texttt{Inv}}(\texttt{mode}, s)$ denote the predicate $(\texttt{mode=SC} \wedge s \in \phi_{safe}) \vee (\texttt{mode=AC} \wedge Reach_M(s, *, \Delta) \subseteq \phi_{safe})$. If the initial state satisfies the invariant $\phi_{\texttt{Inv}}$, then every state $s_t$ reachable from $s$ will also satisfy the invariant $\phi_{\texttt{Inv}}$.* ***(Proof in the Appendix)***

The invariant established in Theorem 4.1 ensures that if the assumptions of the theorem are satisfied, then all reachable states are always contained in $\phi_{safe}$.

Remark 4.1 (Guarantee switching and avoid oscillation). *The liveness property (P2b) guarantees that the system will definitely switch from $N_{sc}$ to $N_{ac}$ (to maximize performance). Property (P3) ensures that the system will stay in AC for some time and not switch back immediately to SC. Note that the liveness property (P2b) is not needed for Theorem 4.1.*

Remark 4.2 (AC is a black-box). *Our well-formedness check does not involve proving anything about $N_{ac}$. (P1a) and (P1b) require that $N_{ac}$ samples at most as fast as $N_{dm}$ and generates the same outputs as $N_{sc}$, this is for smooth transitioning between $N_{ac}$ and $N_{sc}$. We only need to reason about $N_{sc}$, and we need to reason about all possible controller actions (when reasoning with $Reach_M(s, *, \Delta)$). The latter is worst-case analysis, and includes $N_{ac}$'s behavior. One could restrict behaviors to $N_{ac} \cup N_{sc}$ if we wanted to be more precise, but then $N_{ac}$ would not be a black-box anymore.*

Definition 4.1 (Regions). *Let $R(\phi, t) = \{s \mid s \in \phi \wedge Reach_M(s, *, t) \subseteq \phi\}$. For example, $R(\phi_{safe}, \Delta)$ represents the region or set of states in $\phi_{safe}$ from which all reachable states in time $\Delta$ are still in $\phi_{safe}$.*

***Regions of operation of a well-formed RTA module.*** We informally describe the behavior of an RTA protected module by organizing the state space of the system into different regions of operation (Figure 6). R1 represents the unsafe region of operation for the system. Regions R2-R5 represent the safe region and R3-R5 are the recoverable regions of the state space. The region R3\R4 represents the *switching control* region (from AC to SC) as the time to escape $\phi_{safe}$ for the states in this region is less than $2\Delta$.

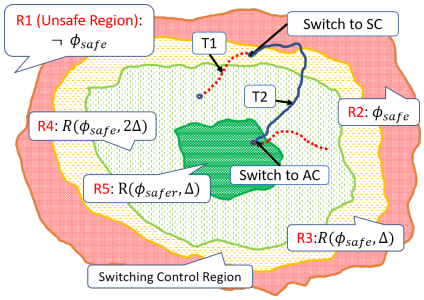



**Figure 6: Regions of State Space.**

As the DM is guaranteed to sample the state of the system at least once in $\Delta$ time (property (P1a)), the DM is guaranteed to switch control from AC to SC if the system remains in the switching control region for at least $\Delta$ time, which is the case before system can leave region R3. Consider the case where T1 represents a trajectory of the system under the influence of AC, when the system is in the switching control region the DM detects the imminent danger and switches control to SC. (P1a) ensures that $N_{sc}$ takes control before the system escapes $\phi_{safe}$ in the next $\Delta$ time. Property (P2a) ensures that the resultant trajectory T2 of the system remains inside the safe region and Property (P2b) ensures that the system eventually enters region R5 where the control can be returned to AC for maximizing the performance of the system. Property (P3) ensures that the switch to AC is safe and the system will remain in AC mode for at least $\Delta$ time.

Remark 4.3 (Choosing $\phi_{safer}$ and $\Delta$). *The value of $\Delta$ is critical for ensuring safe switching from AC to SC. It also determines how conservatively the system behaves: for example, large value of $\Delta$ implies a large distance between boundaries of region R4 and R5 during which SC (conservative) is in control. Small values of $\Delta$ and a larger R5 region ($\phi_{safer}$) can help maximize the use of AC but might increase the chances of switching between AC and SC as the region between the boundaries of R4 and R5 is too small. Currently, we let the programmer choose these values and leave the problem of automatically finding the optimal values as future work.*

***From theory to practice:*** We are assuming here that the checks in Property (P2) and Property (P3) can be performed. The exact process for doing so is outside the scope of this paper. However, we leveraged existing work that can be used for performing these checks. First, consider the problem of synthesizing the safe controller $N_{sc}$ for a given safe set $\phi_{safe}$. $N_{sc}$ can be synthesized using pre-existing safe control synthesis techniques. For example, for the motion primitives, we can use a framework like FaSTrack [17] for synthesis of low-level $N_{sc}$. We elaborate on this further in Sec 5.1.

Next, we note that the DM needs to reason about the reachable set of states for a system when either the controller is fixed to $N_{sc}$ or is nondeterministic. Again, there are plenty of tools and techniques for performing reachability computations [18]. One particular concept that SOTER requires here is the notion of *time to failure less than 2*$\Delta$ ($\texttt{ttf}_{2\Delta}$). The function $\texttt{ttf}_{2\Delta} : \mathcal{S} \times 2^{\mathcal{S}} \rightarrow \mathbb{B}$, given a state $s \in \mathcal{S}$ and a predicate $\phi \subseteq \mathcal{S}$ returns *true* if starting from $s$, the minimum time after which $\phi$ may not hold is less than or equal to $2\Delta$. The check $Reach(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ in Figure 4 can be equivalently described using the $\texttt{ttf}_{2\Delta}$ function as $\texttt{ttf}_{2\Delta}(s_t, \phi_{safe})$. We show how $\texttt{ttf}_{2\Delta}$ for motion primitives is computed in Sec 5.1.

```
1  type State = ..;
2  ...
3  fun PhiSafer_MPr (s : State) : bool { ... }
4  fun TTF2D_MPr (s : State) : bool { ... }
5  ...
6  node MotionPrimitiveSC period 60;
7  subscribes LocalPosition, LocalVelocity, NextWaypoint;
8  publishes  Control;
9  { /* body */ }
10
11 rta SafeMotionPrimitive = { MotionPrimitive,
       MotionPrimitiveSC, 150, PhiSafer_MPr, TTF2D_MPr};
```

**Figure 7: Declaration of a RTA module**

*Example 4.1 (Example of an RTA module).* Figure 7 presents the declaration of an RTA module consisting of the `MotionPrimitive` (Figure 3) and `MotionPrimitiveSC` as the AC and SC nodes. The boolean function `PhiSafer_Mpr` is used to check if the system state is in $\phi_{safer}$ and `TTF2D_Mpr` corresponds to the $\texttt{ttf}_{2\Delta}$ function for the safe motion primitives RTA module. These functions are used for evaluating the switching logic described in Figure 4. □

## 4.1 Correctness of an RTA System

A large system is generally built by composing multiple components together. When the system-level specification is decomposed into a collection of simpler component-level specifications, one can scale provable guarantees to large, real-world systems.

***Composition.*** If RTA modules $P$ and $Q$ are composable then their composition $P \parallel Q$ is a system consisting of the two modules $\{P, Q\}$. The operational semantics of the RTA system $P \parallel Q$ is as described in Section 3.3. Note that composition of two RTA systems $S1$ and $S2$ is an RTA system $S1 \cup S2$ if all modules in $S1 \cup S2$ are composable.

THEOREM 4.2 (COMPOSITIONAL RTA SYSTEM). *Let* $S = \{M_0, \ldots M_n\}$ *be an RTA system. If for all* $i$, $M_i$ *is a well-formed RTA module satisfying the safety invariant* $\phi^i_{Inv}$ *(Theorem 4.1) then, the RTA system* $S$ *satisfies the invariant* $\bigwedge_i \phi^i_{Inv}$. ***(Appendix)***

Theorem 4.2 plays an important role in building the reliable software stack in Figure 2c. Each RTA module individually satisfies the respective safety invariant and their composition helps establish the system-level specification.

We note that the definition of $DM$ for an RTA module $M$ is sensitive to the choice of the environment for $M$. Consequently, every attribute of $M$ (such as well-formedness) depends on the context in which $M$ resides. We implicitly assume that all definitions for a single RTA $M$ are based on a completely nondeterministic context. All results hold for this interpretation, but they also hold for some more constrained environments.

***Input-Output Contracts.*** An RTA module is an open system interacting with its environment using the input-output topics. When defining the semantics of an RTA module, we assumed that the environment can generate any input $v \in \mathcal{V}$. Hence, SC must satisfy the correctness properties (P2) under any environment. Designing such an SC is extremely hard and in most cases infeasible. In practice, a safe controller is designed under certain *input assumptions*. Also, when an RTA module is composed with another module, the input of one module may be an output of other. In which case, the RTA module must provide *output guarantees* that satisfy the input assumptions of the composed module. We would like to ensure that the output guarantee holds even when the AC is in control.

To specify simple input-output contracts, SOTER allows attaching contracts to topics. The declaration below attaches an assumption `ControlRange` with the values stored at topic `VelocityControl`.

```
1 fun ControlRange(c : controlType) : bool { .. }
2 topic VelocityControl:(controlType) assumes ControlRange;
```

To capture input assumptions, the ENVIRONMENT-INPUT rule (Figure 5) is updated to allow only those values that satisfy the corresponding topic assumption. To provide the output guarantees, the publish semantics is updated such that a publish operation is disabled if it does not satisfy the topic assumption on which it is publishing the value.

# 5 CASE STUDY: SURVEILLANCE SYSTEM

We empirically evaluate SOTER framework by building an RTA-protected software stack (Figure 2c) that satisfies the safety invariant: $\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$. The goal of our evaluation is threefold: (Goal1) Demonstrate how the SOTER-RTA can ensure that the drone safely navigates ($\phi_{mpr}$) in the workspace using untrusted low-level controllers, and also how the programmable switching feature of an RTA module can help maximize the performance. (Goal2) Demonstrate how the SOTER-RTA can safely switch from mission mode to recovery mode under low-battery conditions to guarantee $\phi_{bat}$. (Goal3) Demonstrate how the SOTER-RTA can ensure $\phi_{plan}$ in the presence of software bugs in the third-party motion planner.

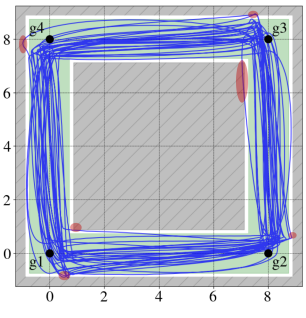
***SOTER Implementation.*** The SOTER tool-chain consists of two components: the compiler and a C runtime. The compiler converts the source-level syntax of a SOTER program into C code. This code contains statically-defined C array-of-structs and functions for the topics, nodes, and functions declarations. The *OE* that controls the output of each node is implemented as a shared-global data-structure updated by all the *DM*s in the program. The SOTER C runtime executes the program according to the program's operational semantics by using the C representation of the nodes. The periodic behavior of each node was implemented using OS timers for our experiments, deploying the generated code on real-time operating system is future work.

***Experimental Setup*** For our experiments on real drone hardware, we use 3DR Iris [11] drone that comes with the open-source Pixhawk PX4 [10] autopilot. The simulation results were done in Gazebo [12] simulator environment that has high fidelity models of Iris drone. In our simulations, we execute the PX4 firmware in the loop. Details about demonstration videos and other experiments is available in the Supplementary Material.

### 5.1 RTA for Safe Motion Primitives

A drone navigates in the 3D space by tracking trajectories between waypoints computed by the motion planner (Figure 2a). Given the next waypoint, an appropriate motion primitive is used to track the reference trajectory. Informally, a motion primitive consists of a pre-computed control law (sequence of control actions) that regulates the state of drone as a function of time.

In our experiments, we use the motion primitives provided by the PX4 autopilot [10] as our advanced controller. A motion primitives take as input the next waypoint and generates the low level control to traverse the reference trajectory from current position to the target waypoint. Since, the control is optimized for performance rather than safety, it can be potentially unsafe.




To demonstrate this, we experimented with the motion primitives provided by the drone manufacturers. The drone was tasked to repeatedly visit locations $g_1$ to $g_4$ in that order,i.e., the sequence of waypoints $g_1, \ldots g_4$. The blue lines represent the trajectories of the drone. Given the complex dynamics of a drone and noisy sensors, ensuring that it precisely follows a fixed trajectory (ideally a straight line joining the waypoints) is extremely hard. The advanced controller (untrusted) optimizes for time and, hence, during high speed maneuvers the reduced control on the drone leads to overshoot and trajectories that collide with obstacles (represented by the red regions). Note that the advanced controller can be used during majority of this mission except for a few instances of unsafe maneuvers. This motivates our case-study: to build a RTA-protected motion primitive that ensures safety when using the performant third-party controller.

To achieve (Goal1), there are three important steps: (1) Design of the safe controller $N_{sc}$; (2) Designing the $\texttt{ttf}_{2\Delta}$ function that controls switching from the AC to SC for the motion primitive; (3) Programming the switching from SC to AC and choosing an appropriate $\Delta$ and $\phi_{safer}$ so that the system is not too conservative.

***Designing*** $N_{sc}$**.** For the motion primitive, we need to guarantee that the trajectory taken by the drone does not collide with any obstacles. The $N_{sc}$ must satisfy the Property (P2), where $\phi_{safe}$ is the region not occupied by any obstacle. Techniques from control theory, like *reachability* [19] can be used for designing $N_{sc}$. However, reachability based techniques are slow and scale exponentially with state dimension; and hence cannot be used to compute $N_{sc}$ in an online fashion. Recently, FaSTrack [17] has been proposed which can compute a generic $N_{sc}$ offline. FaSTrack helps quantifies the maximum deviation from the reference trajectory that might occur while regulating the drone to a desired waypoint using $N_{ac}$.

To use FaSTrack, we need the non-linear 12D state dynamics of the quadrotor with position $(x, y, z)$; velocity $(v_x, v_y, v_z)$; pitch, roll and yaw $(\theta_x, \theta_y, \theta_z)$; and pitch, roll and yaw rates, $(\omega_x, \omega_y, \omega_z)$. For our analysis, we consider a 10D model by setting $\theta_z = \omega_z = 0$ which can be solved using decomposition techniques [20] to make the computations scalable. The advantage of using FaSTrack is two fold; (1) it provides us $N_{sc}$ and a *tracking error bound* (TEB) which can be used to compute $\phi_{safer}$; and (2) the computations are done offline but its results can be used online in an efficient manner. Intuitively, TEB quantifies the maximum deviation between the trajectories when using $N_{ac}$ and $N_{sc}$ on the drone. We refer the readers to [17] for the further details.

***Designing*** $\texttt{ttf}_{2\Delta}$ ***that controls switching from AC to SC.*** To design the switching condition from AC to SC, we need to compute the ttf function that checks $Reach(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$ (Figure 4) where $s_t$ is the current state.

Consider the 2D representation of the workspace ((Figure 2a)) in Figure 8b. The obstacles (shown in grey) represent the $\phi_{unsafe}$ region, any region outside is $\phi_{safe}$. We first pad all obstacles with TEB (shown in red) for motion planning. A region of TEB thickness around the obstacle is contained in $\phi_{safe}$, but safety is guaranteed *only* for $N_{sc}$ in this region and not $N_{ac}$. Moreover, $N_{sc}$ can guarantee safety for all locations in $\phi_{safe}$ (P2). We can use the level set toolbox [19], to compute the backward reachable set from $\phi_{safe}$ in $2\Delta$ (shown in yellow), i.e., the set between the boundary of the yellow region and the obstacle represents the states from where the drone can leave $\phi_{safe}$ (collide with obstacle) in $2\Delta$. This set can be represented as a value function $V : \mathcal{S} \rightarrow \mathbb{R}$ which associates a real value to each state such that,

$$V(s) = \begin{cases} > 0 & \text{if } s \in R(\phi_{safe}, 2\Delta) \\ 0 & \text{if } s \in \text{ boundary of } R(\phi_{safe}, 2\Delta) \\ < 0 & \text{if } s \notin R(\phi_{safe}, 2\Delta) \end{cases} \quad (1)$$

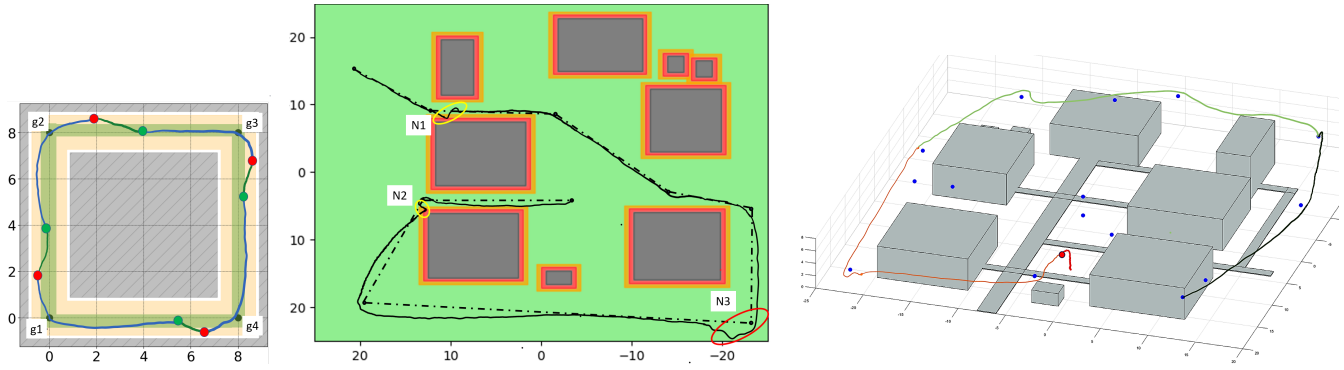
Intuitively, $V(s) \leq 0$ if it there exists a control law which can take the drone less than $2\Delta$ to leave $\phi_{safe}$. Hence, $\texttt{ttf}_{2\Delta} := V \leq 0$.

***Programming switching from*** SC ***to*** AC**.** In order to maximize the performance of the system, the RTA module must switch from SC to AC after the system has recovered. In our experiments, we choose $\phi_{safer} = R(\phi_{safe}, 2\Delta)$ (shown in green). $N_{sc}$ is designed such that given $\phi_{safer}$, Property (P2b) holds. The DM transfers control to AC when it detects that the drone is in $\phi_{safer}$, which is the backward reachable set from $\phi_{safe}$ in $2\Delta$ time. The value function returned by this computation can be used to define the check $s_t \in \phi_{safer}$ (Figure 4).

Choosing the period $\Delta$ is an important design decision. Choosing a large $\Delta$ can lead to overly-conservative $\texttt{ttf}_{2\Delta}(s_t, \phi_{safe})$ and $\phi_{safer}$. In other words, a large $\Delta$ pushes the switching boundaries further away from the obstacle. In which case, a large part of the workspace is covered by red or yellow region where the SC (conservative controller) is in control. For the motion primitives, ttf is the boundary of $\phi_{safer}$.This is the largest $\phi_{safer}$ we can compute without breaking the well-formed constraints.

Using FaSTrack with $\delta(N_{ac}) = \delta(N_{sc}) = 0.01$, we get a TEB of 0.313m. We chose $\Delta \in (0.05, 0.1)$ to get good performance.

***Experimental results.*** We implemented the safe motion primitive as a RTA module using the components described above. Figure 8a presents one of the interesting trajectories where the SC takes control multiple times and ensures the over all correctness of the mission. The green tube inside the yellow tube represents the $\phi_{safer}$ region. The red dots represent the points where the DM switches control to SC and the green dots represents the points where the DM returns control back to the AC for optimizing performance. The approximate time taken by the drone to go from $g_1$ to $g_4$ is 10 secs when only the unsafe $N_{ac}$ is in control (can lead to collisions),




(a) RTA for Safe Motion Primitive (b) Safe Motion Primitives during Surveillance Mission (c) Battery Safety during Surveillance Mission

**Figure 8: Evaluation of RTA-Protected Drone Surveillance System built using SOTER**

14 secs using the RTA protected safe motion primitive, and 24 secs when only using the safe controller. Hence, using RTA provides a "safe" middle ground without sacrificing performance too much.

Figure 8b presents the 2D representation of our workspace in Gazebo (Figure 2a). The dotted lines represents one of the reference trajectories of the drone using the surveillance mission. The trajectory in black shows the trajectory of the drone when using the RTA-protected software stack consisting of the safe motion primitive. The parts of trajectory marked N1 and N2 is where the $N_{sc}$ takes control, pushes the drone back in to $\phi_{safer}$ (green) and returns control back to the $N_{ac}$. We observe that the $N_{ac}$ is in control for most part of the surveillance mission even in cases when the drone deviates from the reference trajectory (N3) but is safe.

## 5.2 RTA for Battery Safety

We would like our software stack to provide battery-safety guarantee (Goal2), that prioritizes landing the drone safely when the battery charge reaches below a threshold level.

***Defining RTA components.*** We first augment the state of the drone with the current battery charge, $b_t$. $N_{ac}$ is a node that receives the current motion plan from the planner and forwards it to the motion primitives module. $N_{sc}$ is a certified planner that safely lands the drone from its current position. The set of all safe states for the battery safety, $\phi_{safe} := b_t > 0$, i.e., the drone is safe as long as the battery does not run out of charge. We define $\phi_{safer} := b_t >$ 85%, i.e., the battery charge is greater than 85%. Since, the battery discharges at a slower rate compared to changes in the position of the drone, we can define a larger $\Delta$ for the battery RTA compared to the motion primitive RTA. For our experiments, we chose $\Delta = 0.1s$.

***Designing*** $\texttt{ttf}_{2\Delta}(b_t, \phi_{safe})$**.** To design the $\texttt{ttf}_{2\Delta}$, we first define two terms: (1) Maximum battery charge to land $T_{max}$; and (2) Maximum battery discharge in $2\Delta$, $cost^*$. In general $T_{max}$ depends on the current position of the drone. However, we approximate $T_{max}$ as the battery required to safely land from the maximum height attained by the drone. Although, conservative it is easy to compute and can be done offline. To find $cost^*$, we first define a function $cost$, which given the low level control to the drone and a time period; returns the amount of battery the drone discharges by applying that control for the given time period. Then, $cost^* = \max_u cost(u, 2\Delta)$ is the maximum discharge that occurs in time $2\Delta$ across all possible controls, $u$. We can now define $\texttt{ttf}_{2\Delta}(b_t, \phi_{safe}) = b_t - cost^* < T_{max}$. It guarantees that the DM switches control to SC if the current battery level may not be sufficient to land safely if AC were to apply the worst possible control.

***Programming switching from*** SC ***to*** AC**.** The DM returns control to $N_{ac}$ once the drone is sufficiently charged. This is defined by $\phi_{safer}$, which is chosen to assert that the battery has at least 85% charge before the DM can hand control back to AC. The resultant RTA module is well formed and satisfies battery safety properties.

***Experimental results.*** We implemented the battery safety RTA module with the components defined above. Figure 8(c) shows a trajectory, where the battery goes below the safety threshold causing DM to transfer control to $N_{sc}$ which safely lands the drone. For the purposes of our experiment, we mock the battery sensor and decrease the charge monotonically. The trajectory is broken down into 4 regions, (1) dark green where the battery level is greater than 70%, (2) the light green where the battery level is in (50%, 70%), (3) the orange where the battery level is in (30%, 50%), and (4) the red where the `ttf` is true and the SC takes over to land the drone safely. *The video corresponding to a similar experiment on a real drone is available in the supplementary material.*

## 5.3 RTA for Safe Motion Planner

We use OMPL [14], a third-party motion-planning library that implements many state-of-the-art sampling-based motion planning algorithms. We implemented the motion-planner for our surveillance application using the RRT* [21] algorithm from OMPL. To evaluate (Goal3), we injected bugs into the implementation of RRT* such that in some cases the generated motion plan can collide with obstacles (violating $\phi_{plan}$). To achieve (Goal3), we leverage input-output contracts feature provided by SOTER. We annotated the topic `MotionPlan` on which the planner publishes the motion plan with an assumption that the motion plan should not collide with an obstacle. As a result the system was safe and the publish operations that violate the assumption of the motion primitive were ignored.

## 5.4 Evaluation Summary

To summarize, we used the theory of well-formed RTA module to construct three RTA modules: motion primitives, battery safety, and motion planner. We leverage Theorem 4.1 to ensure that the modules individually satisfy the safety invariants $\phi_{mpr}$, $\phi_{bat}$, and $\phi_{plan}$ respectively. The RTA-protected software stack (Figure 2c)

is a composition of the three modules and using Theorem 4.2 we can guarantee that the system satisfies the desired safety invariant of $\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$. Most of the graphs shown in this section are from the software-in-loop simulations of the RTA-protected software stack. We also deployed the generated code on real drone to conduct similar experiments. The details and videos are provided in supplementary material.

## 6 RELATED WORK

We have discussed related work throughout the paper. The simplex architecture has popularly been used in applications other than avionics and robotics as well. In [8], the authors present a component-based simplex architecture and A-G contracts are used to automatically determine the switching logic and perform coordinated switching if required. The paper uses these principles to design prototype software stack for QuickBot in Matlab. In this paper, we take inspiration and address the two problems mentioned as future work in the paper **(1)** we present a programming framework for building systems compositionally so that the overall system safety problem can be decomposed into RTA invariants guaranteed by individual modules **(2)** we use the framework to build safe UAV missions. Another important distinction is that our formalism uses $\phi_{safer}$ to ensure performant behavior of the system. In [5], the authors apply simplex approach for sandboxing cyber-physical systems and present automatic reachability based approaches for inferring switching conditions. We formalize a generic runtime assurance architecture and implement it in programming framework for mobile robotic systems.

Safe control of safety-critical systems is a very active area of research. Reachability [19] analysis is frequently used to study the safety of control systems. The idea of using an advanced controller (AC) under nominal conditions; while at the boundaries, using optimal safe control (SC) to maintain safety has been used in [22] for operating quadrotors in the real world. In [23] the authors uses a switching architecture ([24]) to switch between a nominal safety model and learned performance model to synthesize policies for a quadrotor to follow a trajectory. We can use these approaches to design SC and also in some cases use our framework to build these systems that already using switching logic.

Runtime verification has been applied to robotics [25–29] where monitors are used to check the status of path planner and tasks executions. In this paper, we formalize and implement a runtime assurance based programming framework that supports recovery to ensure safe execution of robotic system in the real-world. Recently, ModelPlex [30] combines offline verification of CPS models with runtime validation of system executions for compliance with the model to build correct by construction runtime monitors which provides correctness guarantees for CPS executions at runtime. While ModelPlex is similar in motivation to our RTA framework, it differs in two aspects: (1) it relies on full knowledge of the model (or in our case $N_{ac}$) to be available, (2) while it synthesizes the switching condition to the $N_{sc}$, it does not provide the conditions under which $N_{ac}$ can take over control, i.e., they do not build $\phi_{safer}$.

## REFERENCES


[1] L. Sha, "Using simplicity to control complexity," *IEEE Software*, 2001.

[2] J. D. Schierman, M. D. DeVore, N. D. Richards, N. Gandhi, J. K. Cooper, K. R. Horneman, S. Stoller, and S. Smolka, "Runtime assurance framework development for highly adaptive flight control systems," tech. rep., Barron Associates, Inc. Charlottesville, 2015.

[3] D. Seto, E. Ferriera, and T. Marz, "Case study: Development of a baseline controller for automatic landing of an f-16 aircraft using linear matrix inequalities (lmis)," Tech. Rep. CMU/SEI-99-TR-020, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2000.

[4] D. Phan, J. Yang, R. Grosu, S. A. Smolka, and S. D. Stoller, "Collision avoidance for mobile robots with limited sensing and limited information about moving obstacles," *Formal Methods in System Design*, 2017.

[5] S. Bak, K. Manamcheri, S. Mitra, and M. Caccamo, "Sandboxing controllers for cyber-physical systems," in *IEEE/ACM Second International Conference on Cyber-Physical Systems*, 2011.

[6] M. Clark, X. Koutsoukos, J. Porter, R. Kumar, G. Pappas, O. Sokolsky, I. Lee, and L. Pike, "A study on run time assurance for complex cyber physical systems," tech. rep., AIR FORCE RESEARCH LAB WRIGHT-PATTERSON AFB OH AEROSPACE SYSTEMS DIR, 2013.

[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 *IEEE Symposium on Real-Time and Embedded Technology and Applications*, 2009.

[8] D. Phan, J. Yang, M. Clark, R. Grosu, J. D. Schierman, S. A. Smolka, and S. D. Stoller, "A component-based simplex architecture for high-assurance cyber-physical systems," *arXiv preprint arXiv:1704.04759*, 2017.

[9] M. Quigley, K. Conley, B. P. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, and A. Y. Ng, "Ros: an open-source robot operating system," in *ICRA Workshop on Open Source Software*, 2009.

[10] "PX4 Autopilot." https://pixhawk.org/, 2017.

[11] "3D Robotics." https://3dr.com/, 2017.

[12] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," IEEE, 2004.

[13] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016. arXiv:1606.01540.

[14] I. A. Şucan, M. Moll, and L. E. Kavraki, "The Open Motion Planning Library," *IEEE Robotics & Automation Magazine*, 2012.

[15] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of artificial intelligence research*, 1996.

[16] B. Dutertre and M. Sorea, "Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata," in *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems*, Springer, 2004.

[17] S. L. Herbert*, M. Chen*, S. Han, S. Bansal, J. F. Fisac, and C. J. Tomlin, "Fastrack: a modular framework for fast and guaranteed safe motion planning," *IEEE Conference on Decision and Control*, 2017.

[18] G. Frehse, C. Le Guernic, A. Donzé, S. Cotton, R. Ray, O. Lebeltel, R. Ripado, A. Girard, T. Dang, and O. Maler, "Spaceex: Scalable verification of hybrid systems," in *Computer Aided Verification (CAV)*, 2011.

[19] I. M. Mitchell, A. M. Bayen, and C. J. Tomlin, "A time-dependent hamilton-jacobi formulation of reachable sets for continuous dynamic games," *IEEE Transactions on Automatic Control*, 2005.

[20] M. Chen, S. L. Herbert, M. S. Vashishtha, S. Bansal, and C. J. Tomlin, "Decomposition of reachable sets and tubes for a class of nonlinear systems," 2017. arXiv:1611.00122v3.

[21] S. Karaman and E. Frazzoli, "Sampling-based algorithms for optimal motion planning," *The international journal of robotics research*, 2011.

[22] A. K. Akametalu, S. Kaynama, J. F. Fisac, M. N. Zeilinger, J. H. Gillula, and C. J. Tomlin, "Reachability-based safe learning with gaussian processes," in *IEEE Conference on Decision and Control,*, 2014.

[23] A. Aswani, P. Bouffard, and C. J. Tomlin, "Extensions of learning-based model predictive control for real-time application to a quadrotor helicopter," in *American Control Conference, ACC 2012, Montreal, QC, Canada, June 27-29, 2012*, 2012.

[24] A. Aswani, H. González, S. S. Sastry, and C. Tomlin, "Provably safe and robust learning-based model predictive control," *Automatica*, 2013.

[25] O. Pettersson, "Execution monitoring in robotics: A survey," *Robotics and Autonomous Systems*, 2005.

[26] I. Lee, H. Ben-Abdallah, S. Kannan, M. Kim, O. Sokolsky, and M. Viswanathan, "A monitoring and checking framework for run-time correctness assurance," 1998.

[27] A. Desai, T. Dreossi, and S. A. Seshia, *Combining Model Checking and Runtime Verification for Safe Robotics*. 2017.

[28] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, 2017.

[29] J. Huang, C. Erdogan, Y. Zhang, B. Moore, Q. Luo, A. Sundaresan, and G. Rosu, "Rosrv: Runtime verification for robots," in *International Conference on Runtime Verification*, 2014.

[30] S. Mitsch and A. Platzer, "Modelplex: verified runtime validation of verified cyber-physical system models," *Formal Methods in System Design*, 2016.

[31] R. Alur and T. A. Henzinger, "Reactive modules," *Formal methods in system design*, vol. 15, no. 1, pp. 7–48, 1999.
[32] N. A. Lynch and M. R. Tuttle, "An introduction to input/output automata," 1988.
[33] T. M. Moldovan and P. Abbeel, "Safe exploration in markov decision processes," *CoRR*, 2012.

# A THEOREMS AND PROOFS

THEOREM A.1 (**RUNTIME ASSURANCE**). *For a well-formed RTA module M, let* $\phi_{\texttt{Inv}}(\texttt{mode}, s)$ *denote the predicate* $(\texttt{mode=SC} \wedge s \in \phi_{safe}) \vee (\texttt{mode=AC} \wedge Reach_M(s, *, \Delta) \subseteq \phi_{safe})$. *If the initial state satisfies the invariant* $\phi_{\texttt{Inv}}$, *then every state* $s_t$ *reachable from* $s$ *will also satisfy the invariant* $\phi_{\texttt{Inv}}$.

PROOF. Let $(\texttt{mode}, s)$ be the initial mode and initial state of the system. We are given that the invariant holds at this state. Since the initial mode is SC, then, by assumption, $s \in \phi_{safe}$. We need to prove that all states $s_t$ reachable from $s$ also satisfy the invariant. If there is no mode change, then invariance is satisfied by Property (P2a). Hence, assume there are mode switches. We prove that in every time interval between two consecutive executions of the DM, the invariant holds. So, consider time $T$ when the DM executes.
(Case1) The mode at time $T$ is SC and there is no mode switch at this time. Property (P2a) implies that all future states will satisfy the invariant.
(Case2) The mode at time $T$ is SC and there is a mode switch to AC at this time. Then, the current state $s_T$ at time $T$ satisfies the condition $s_T \in \phi_{safer}$. By Property (P3), we know that $Reach_M(s_T, *, 2\Delta) \subseteq \phi_{safe}$, and hence, it follows that $Reach_M(s_T, *, \Delta) \subseteq \phi_{safe}$, and hence the invariant $\phi_{\texttt{Inv}}$ holds at time $T$. In fact, irrespective of what actions AC applies to the plant, Property (P3) guarantees that the invariant will hold for the interval $[T, T + \Delta]$. Now, it follows from Property (P1) that the DM will execute again at or before the time instant $T + \Delta$, and hence the invariant holds until the next execution of DM.
(Case3) The current mode at time $T$ is AC and there is a mode switch to SC at this time. Then, the current state $s_T$ at time $T$ satisfies the condition $Reach_M(s_T, *, 2\Delta) \not\subseteq \phi_{safe}$. Since the mode at time $T - \epsilon$ was still AC, and by inductive hypothesis we know that the invariant held at that time; therefore, we know that $Reach_M(s_{T-\epsilon}, *, \Delta) \subseteq \phi_{safe}$. Therefore, for the period $[T - \epsilon, T - \epsilon + \Delta]$, we know that the reached state will be in $\phi_{safe}$ and the invariant holds. Moreover, SC will get a chance to execute in this interval at least once, and hence, from that time point onwards, Property (P2a) will guarantee that the invariant holds.
(Case4) The current mode at time $T$ is AC and there is a no mode switch. Since there is no mode switch at $T$, it implies that $Reach_M(s_T, *, 2\Delta) \subseteq \phi_{safe}$ and hence for the next $\Delta$ time units, we are guaranteed that $Reach_M(s_T, *, \Delta) \subseteq \phi_{safe}$ holds. □

THEOREM A.2 (COMPOSITIONAL RTA). *Let* $S = \{M_0, \ldots M_n\}$ *be an RTA system. If for all i,* $M_i$ *is a well-formed RTA module satisfying the safety invariant* $\phi^i_{Inv}$ *(Theorem A.1) then, the RTA system S satisfies the invariant* $\bigwedge_i \phi^i_{Inv}$.

PROOF. Note that this theorem simply follows from the fact that composition just restricts the environment. Since we are guaranteed output disjointness during composition, composition of two modules is guaranteed to be language intersection. The proof for such composition theorem is described in details in [31, 32]. □

# B EVALUATION

## B.1 Runtime Assurance for Machine Learning Components

Use of machine learning techniques for designing controllers and policies in robotics is increasing as these systems grow in complexity and operate in uncertain environments. Specifically, Reinforcement learning [15] (RL) is being used intensively for synthesizing controllers for robotic systems. In RL, an agent learns a controller or policy by interacting with its environment and receiving a rewards for its action. However, the policy learned depends heavily on the scenarios the agent has seen, and generalization to new environments can be erroneous. Ideally the system uses the machine learned modules under nominal conditions but switches to a safe component when the output of the learned module cannot be trusted. This can be captured by RTA module.

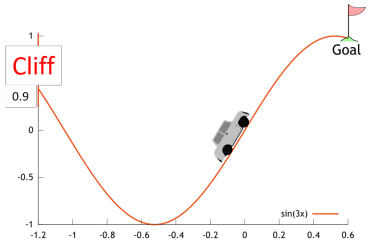



**Figure 9: Mountain-car environment with cliff**

For our case study, we use a popular OpenAI Gym [13] example of a Mountain-car. In the mountain-car example (Figure 9), a car is on a one-dimensional track, positioned between two mountains. The goal is to drive up the mountain on the right to the flag; however, the car's engine is not strong enough to scale the mountain in a single pass. Therefore, the only way to succeed is to drive back and forth to build up momentum. Moreover, we modify the original scenario to include a cliff on the left, which the mountain-car must avoid at all costs. The configuration of the system is a pair $\mathbf{x}_t = (x_t, v_t)$ where $p_t$ is the current position and $v_t$ is the current velocity of the car; and the input control is the acceleration, $u = a_t$.

To design the RTA module, we need to (1) Design the components of the RTA module $N_{ac}$, $N_{sc}$, $\Delta$, $\phi_{safe}$ and $\phi_{safer}$; and (2) Design the switching logic from AC to SC and from SC to AC.

***Designing the RTA components.*** $N_{ac}$ in this case is a state based controller trained using reinforcement learning. $N_{sc}$ and $\phi_{safe}$ can be computed using the level set tool box [**?** ]. We compute the backward reachable set from the goal by treating the cliff as an obstacle, which returns the set of all states which can ultimately reach the goal without falling of the cliff. This set is our $\phi_{safe}$. Our $N_{sc}$ is the safe optimal controller returned by the level set toolbox. In our experiments we choose $\phi_{safer} = R(\phi_{safe}, 2\Delta)$ and $\Delta = \delta(()N_{sc}) = \delta(()N_{ac})$.

***Designing switching logic.*** In this experiment, we define the switching condition from AC to SC with the boolean function $\texttt{ttf}_{2\Delta}(\mathbf{x}_t, \phi_{safe}) = \mathbf{x}_t \notin R(\phi_{safe}, 2\Delta)$ which is true for all states which can leave $\phi_{safe}$ in $2\Delta$ time. In this example this ends up being states in $\phi_{safe}$ which are outside $\phi_{safer}$. For switching from SC to AC, we choose the boundary of $R(\phi_{safer}, \Delta) \subset \phi_{safer}$.

***Experimental results.*** Figure 10 shows an unsafe behavior (blue trajectory) of the learned controller where the trajectory starting at $s_0$ enters the unsafe region (shown in gray), which is the set of position and velocities which lead to the car falling off the cliff.

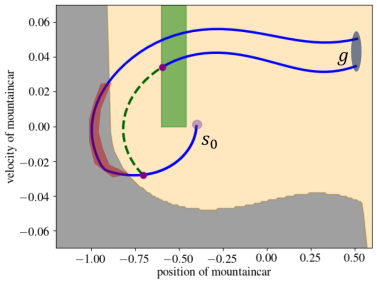



**Figure 10: RTA protected safe mountain-car**

$\phi_{safe}$ is the region shown in light orange, and $R(\phi_{safer}, \Delta)$ is shown in green. The system is designed such that $\Delta = 1$ and it satisfies all the well-formedness constrains for the RTA module. The resulting RTA module produces the trajectory shown in blue ($N_{ac}$ in control), green ($N_{sc}$ in control) and blue ($N_{ac}$ back in control) with the switching points shown as purple dots. The resultant system is always able to drive the trajectory to the goal $g$, the advanced controller is used most part of the trajectory (blue), except for safe controller being used when the car is too close to the cliff.

## B.2 Runtime Assurance for Safe Exploration

When operating in environments which are unknown *a-priori*, a robot faces the challenge of exploring the environment safely and still accomplishing the desired goal. A large body of research, classified as safe exploration [33], focuses on developing techniques to explore the environment in a safe manner.

As a case study, we use the RTA approach for decomposing the problem of optimized exploration from the problem of providing safety guarantee for a robot working in a previously unknown environment. In the obstacle avoidance property considered in the Section 5.1 in the paper, the motion planner was aware of the static obstacles in the system.

To design the RTA module to safely explore unknown environments, we need to (1) Design RTA components $N_{ac}$, $N_{sc}$ and $\phi_{safe}$, (2) Design the switching condition $\texttt{ttf}_{2\Delta}$ for switching from AC to SC, and (3) Programming the switching from SC to AC and choosing an appropriate $\Delta$ and $\phi_{safer}$.

***Design RTA components.*** In this experiment, $N_{ac}$ is a motion planner designed to explore the environment in an optimal way with minimum number of steps and the $N_{sc}$ is responsible to bring the system to an *a-priori* known part of the environment. $\phi_{safe}$ is the entire state space outside the obstacles.

***Design*** $\texttt{ttf}_{2\Delta}$ ***for switching from AC to SC.*** If the environment was known *a-priori* we could have used the reachability based technique proposed in Section 5.1 in the paper. However, in the absence of full knowledge of the environment, we approximate $\texttt{ttf}_{2\Delta} := \{s : s \in \mathcal{S} \text{ s.t. } s + v_{max} \cdot 2 \cdot \Delta \notin \phi_{safe}\}$ where $v_{max}$ is the maximum velocity attainable by the quadrotor in $x, y$, or $z$ direction. Intuitively, it checks if a state would leave $\phi_{safe}$ in $2\Delta$ if it was moving with its highest velocity. This function is more conservative compared to $\texttt{ttf}_{2\Delta}$ proposed in Section 5.1 computed

using reachability. However, this is fast to compute and can be computed on the fly, making it particularly attractive to be used in partially observable environment.

***Programming switching from SC to AC.*** Since the environment is unknown, we have to be conservative about our set $\phi_{safer}$. In our experiments, $\phi_{safer}$ is a predefined known area of the state space. The switching from SC to AC occurs at the boundary of the set $R(\phi_{safer}, \Delta)$. Similar to Section 5.1, $\Delta$ should be chosen to avoid overly-conservative $\texttt{ttf}_{2\Delta}$ and $R(\phi_{safer}, \Delta)$.

***Experimental results.*** We used our RTA module to safely explore an environment(Figure 11) by avoiding collision with the surrounding wall in gray whose location is unknown *a-priori*. $\phi_{safe}$ is the entire workspace contained within the gray wall, $R(\phi_{safer}, \Delta)$ is the green square at the center of the workspace. Additionally, $\Delta$ is chosen such that $R(\phi_{safe}, \Delta)$ is the square with the black boundary and $R(\phi_{safe}, 2\Delta)$ is the square with the dashed black boundary.

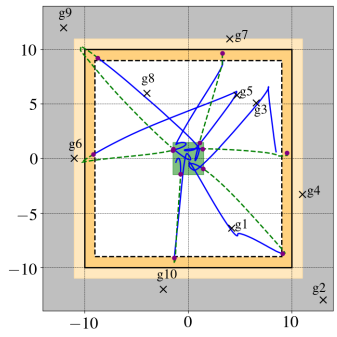



**Figure 11: Safe exploration using RTA module**

In our experiment, the exploring motion planner generates goal points $g_1 - g_{10}$ (black crosses in 11) for the drone to traverse, sequentially. For each goal point, $g_i$, $N_{ac}$ plans a path from the current position of the quadcopter, $\mathbf{x}_t$ to the $g_i$. However, during exploration when $g_i$ satisfies $\texttt{ttf}_{2\Delta}$, our RTA module detects the wall at runtime, switches to SC (shown by green dot in Figure 11) when the trajectory leaves $R(\phi_{safe}, 2\Delta)$ while still inside $R(\phi_{safe}, \Delta)$. $N_{sc}$ brings the trajectory back to $\phi_{safer}$ (shown by the orange trajectory). Once inside $R(\phi_{safer}, \Delta)$, the DM hands back control to the $N_{ac}$ (shown by red dot) and the exploration process begins again.