A Tutorial on Runtime Verification and Assurance

Ankush Desai

EECS 219C

Background

Formal Verification (e.g., Model checking):

- Formal, sound, provides guarantees.
- Doesn’t scale well - state explosion problem.
- Checks a model, not an implementation.
- Most people avoid it – too much effort.

Testing (ad-hoc checking):

- Most widely used technique in the industry.
- Scales well, usually inexpensive.
- Test an implementation directly.
- Informal, doesn’t provide guarantees.

Runtime Verification

Attempt to bridge the gap between formal methods and ad-hoc testing.

- A program is monitored while it is running and checked against properties of interest.
- Properties are specified in a formal notation (LTL, RegEx, etc.).
- Dealing only with finite traces.

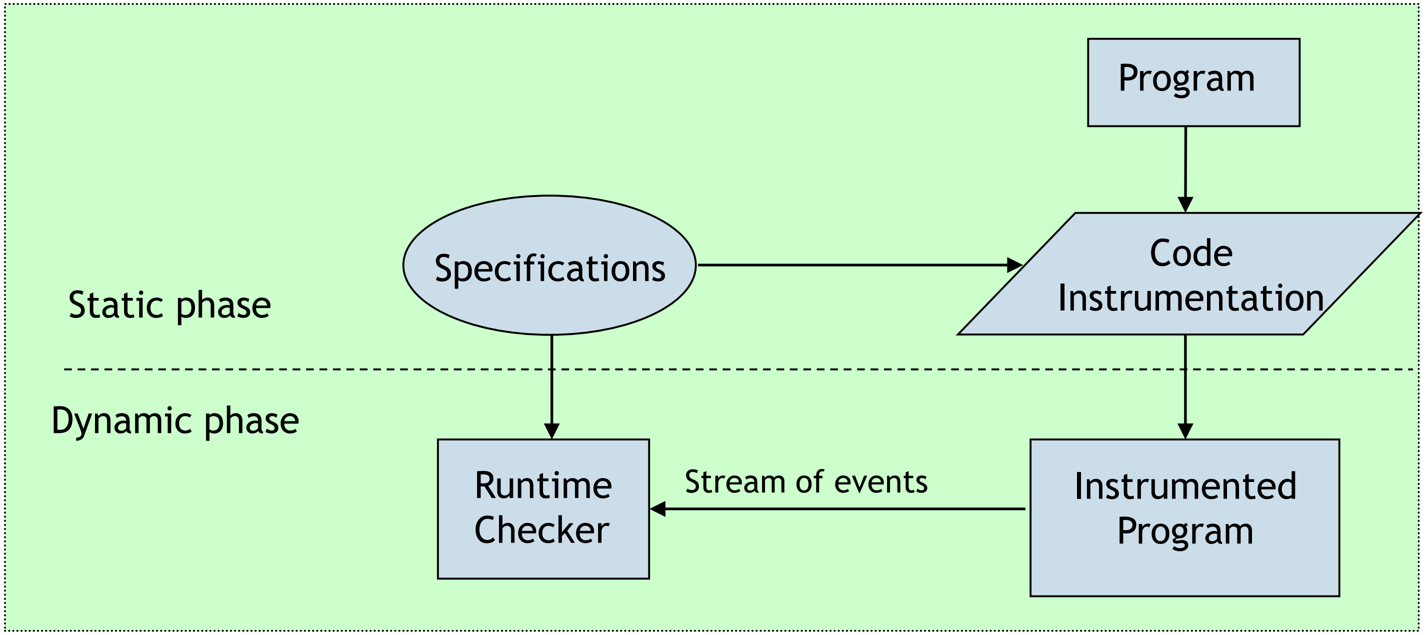
Considered as a light-weight formal method technique.

- Testing with formal "flavour".
- Still doesn't provide full guarantees.

Runtime Verification, cont'd

How to monitor a program?

- Need to extract events from the program while it is running.
- code instrumentation.



Main Challenge: Efficient monitoring

1. Low instrumentation + communication overhead.
2. An efficient monitor should have the following properties:

- No backtracking.
  - Memory-less: doesn’t store the trace.
- Space efficiency.
- Runtime efficiency.
  - A monitor that runs in time exponential in the size of the trace is unacceptable.
  - A monitor that runs in time exponential in the size of the formula is usable, but should be avoided.

Still, What is Runtime Verification?

There are three interpretations of what runtime verification is, in contrast with formal verification discussed in this course.

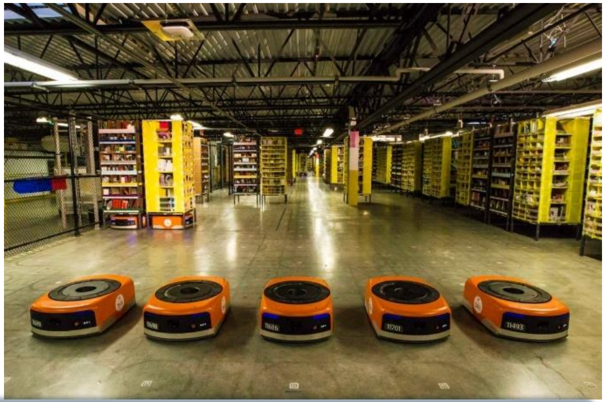
1. RV as lightweight verification, non-exhaustive simulation (testing) plus formal specifications
2. RV as getting closer to implementation, away from abstract models.
3. RV as checking systems after deployment while they are up and running.

DRONA: A Framework for Programming Safe Robotics Systems

Ankush Desai

University Of California, Berkeley

Autonomous Mobile Robotics



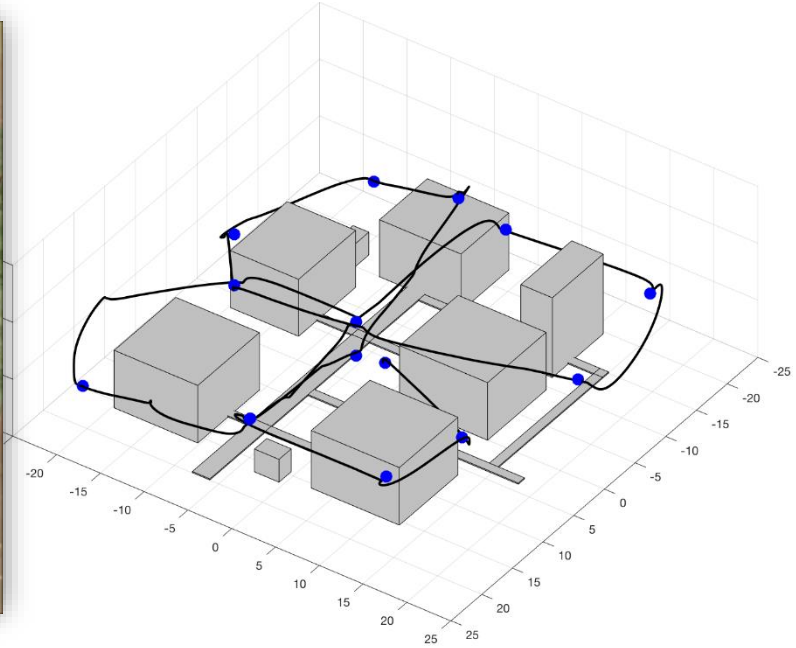
A major challenge in autonomous mobile robotics is programming robots with *formal guarantees* and *high assurance* of correct operation.

Delivery Systems

Agriculture

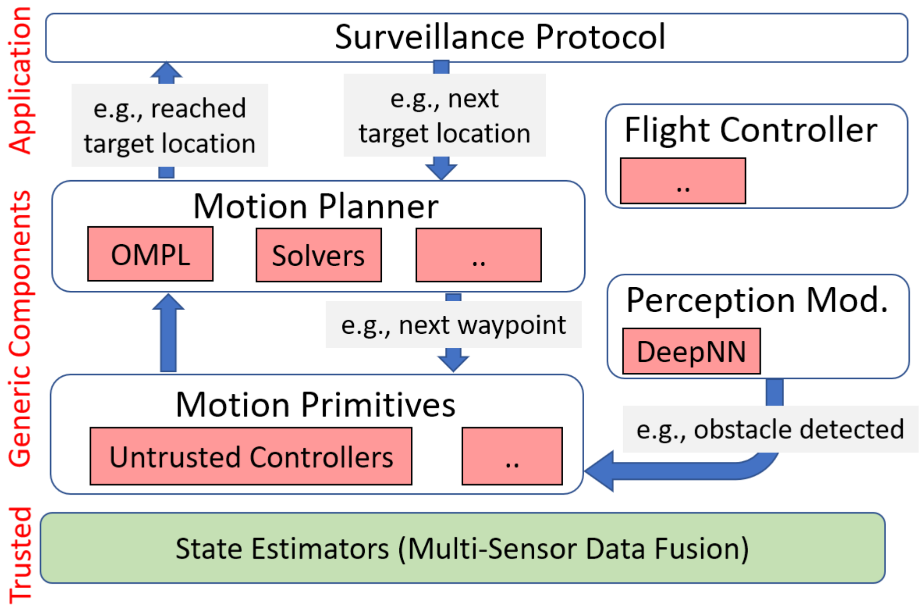
Surveillance Application

Workspace in Gazebo Simulator



Obstacle Map and Drone Trajectory

Robotics Software Stack



Challenges

- Safe programming of concurrent, distributed, reactive, event-driven system.

Design-time: Programming

- Verification and testing of robotics software in the presence of untrusted blocks.

Design-time: Verification

- Guaranteeing safety in the presence of untrusted blocks.

Run-time: Assurance

Related Work

Design-time: Programming

- Programming abstractions for implementing robotics applications: ROS, StarL, ..
- Reactive Synthesis.
    1. Synthesis of high-level controllers (mission/motion planners) from LTL specifications [LTLMoP, TuLiP].
    2. Synthesis high and low-level controllers from LTL specification [SMC].

Pros	Cons
Correct-by-construction controller or strategy.	Generates strategies but not executable code. (gap)
	Uses under-approximate models during synthesis.
	No end-to-end correctness guarantees.

Related Work

Design-time: Verification and Validation

- Reachability.
  - Reachability analysis of the robotics system model (e.g., hybrid system) [Level Set Toolbox, SpaceEx, Flow*, Sapo]
  - Synthesize safe controller as part of the reachability analysis [FastTrack,..]

Pros	Cons
Verification and proof of correctness for the model of the system.	Verifies models but not the executable code. (gap)
	Scalability issues as the dimensions and discrete states increases.
	No end-to-end correctness guarantees.

Related Work

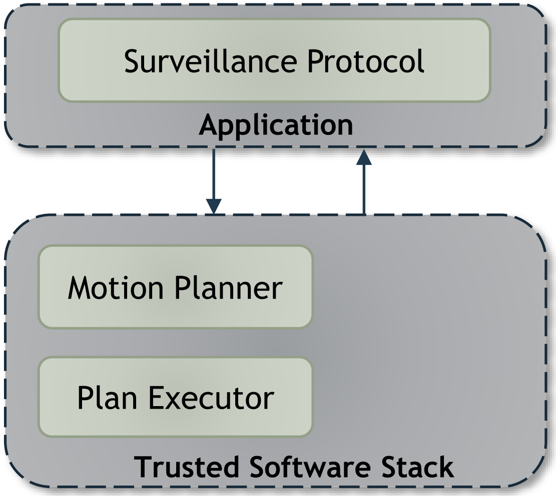
Design-time: Verification and Validation

- Simulation-based Falsification.
  - Testing the software implementation by simulating it in a loop with high-fidelity models of the system [Breach, S-Taliro].

Pros	Cons
Easy to use and more scalable than reachability analysis in terms of complexity of the system and bug finding.	No proofs, no guarantees.
Falsification on actual software implementation!	Scalability issues in terms of coverage for real-world systems.

Our Contributions

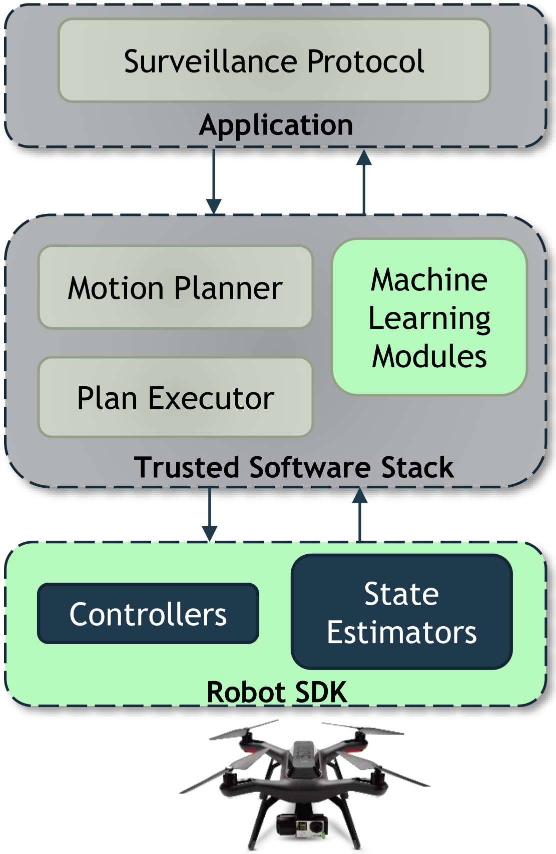
1. A high-level programming language for implementing reactive software:
   - P Programming Language.



Our Contributions

1. A high-level programming language for implementing reactive software:
   - P Programming Language.

2. Verification of the robotics software.
   - Using discrete abstractions of the untrusted components.



Surveillance Protocol

Application

Motion Planner

Plan Executor

Machine Learning Modules

Trusted Software Stack

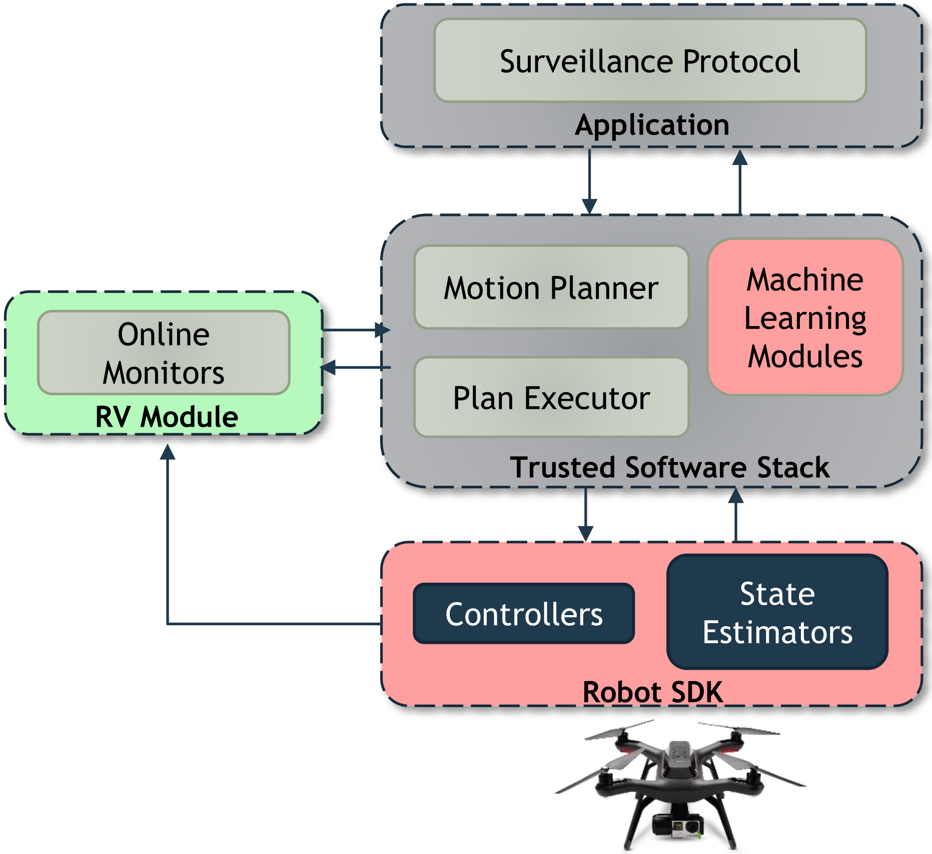
Controllers

State Estimators

Robot SDK

Our Contributions

1. A high-level programming language for implementing reactive software:
    - P Programming Language.
2. Verification of the robotics software.
    - Using discrete abstractions of the robot behavior.
3. Use Runtime Assurance to ensure that the assumptions hold.



Surveillance Protocol

Application

Motion Planner

Plan Executor

Machine Learning Modules

Trusted Software Stack

Online Monitors

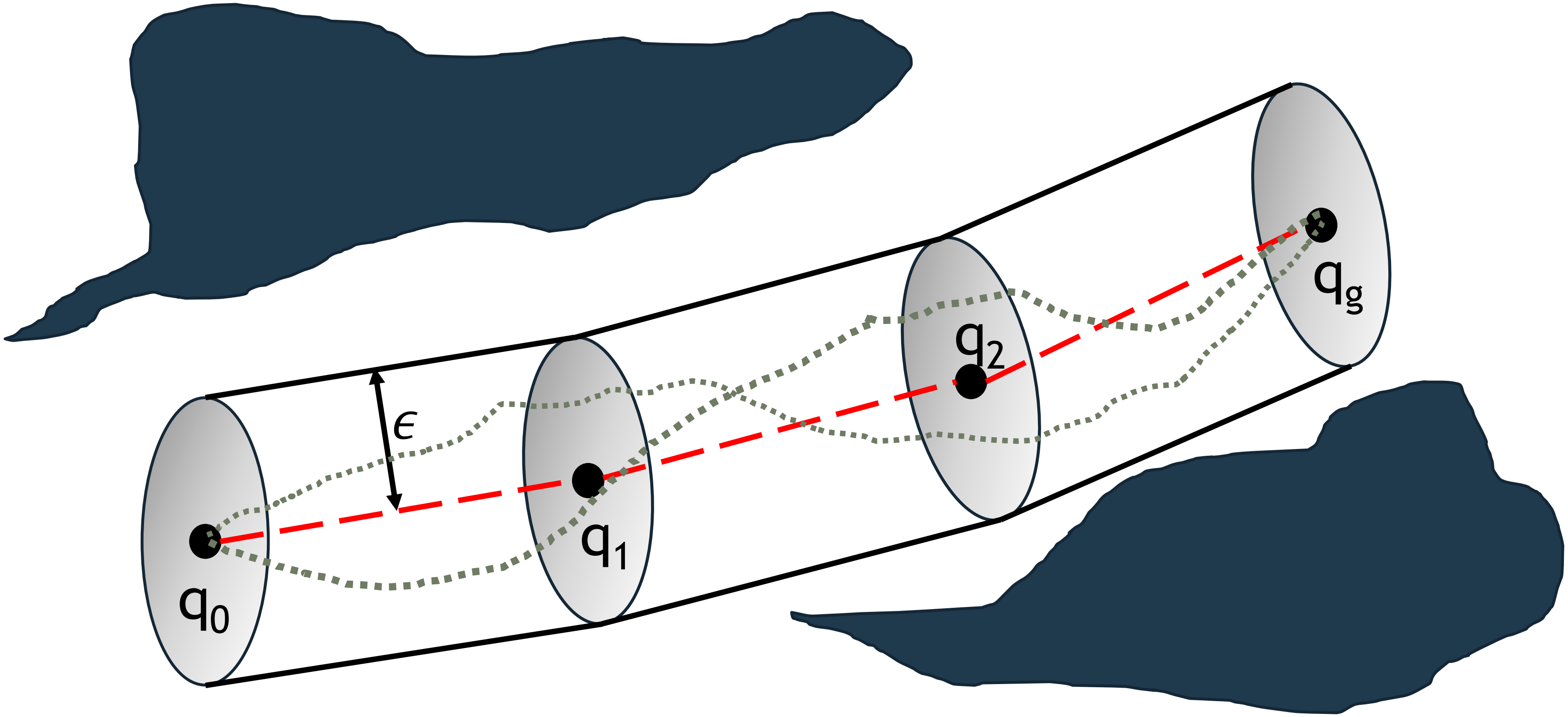
RV Module

Controllers

State Estimators

Robot SDK

What are these abstractions?

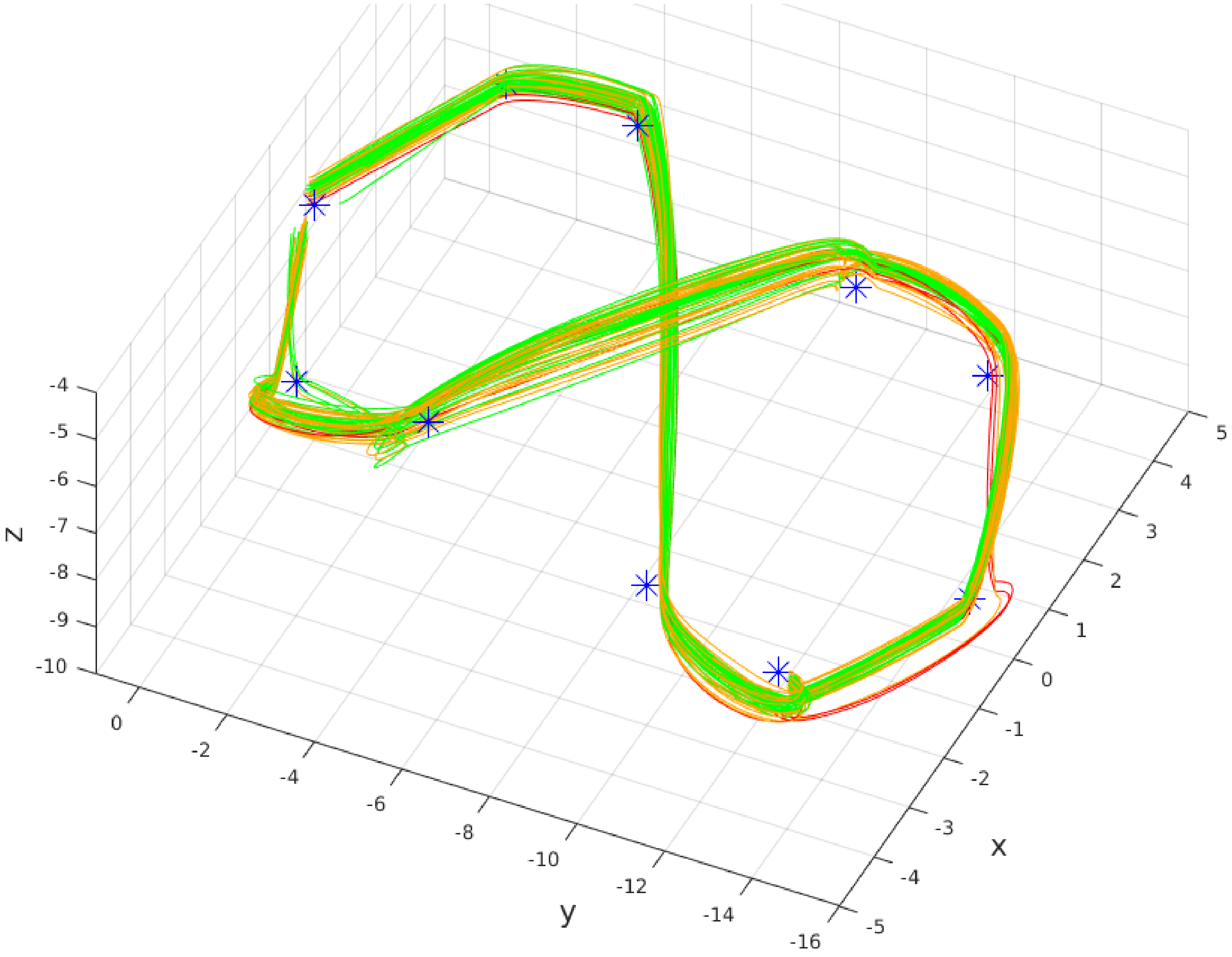


Verified Motion Planner

- Verify that the plans generated by the motion planner are always ϵ distance away from all obstacles.

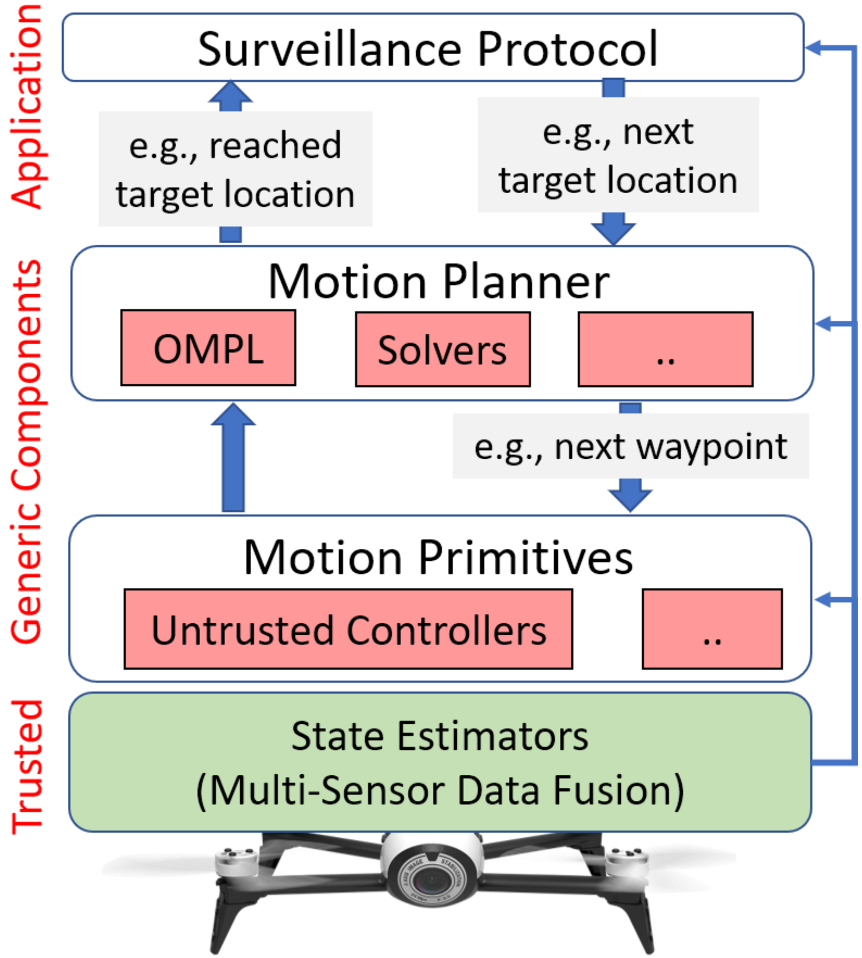
- The planner is safe and provides the guarantee of obstacle-avoidance under the assumption.

Validating Low-Level Controllers



Challenge 3: Guaranteeing Safety at Run time when Design-time assumptions are violated

Robotics Software Stack



Application

Generic Components

Trusted

Surveillance Protocol

e.g., reached target location

e.g., next target location

Motion Planner

OMPL

Solvers

..

e.g., next waypoint

Motion Primitives

Untrusted Controllers

..

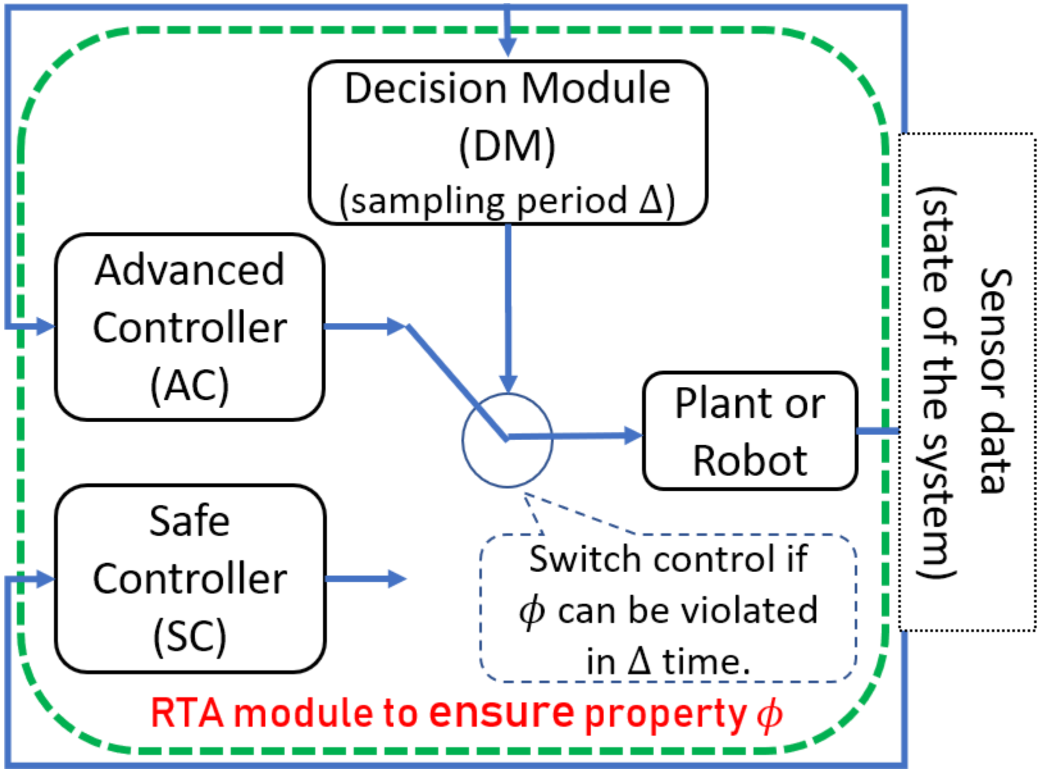
State Estimators (Multi-Sensor Data Fusion)

(1) Obstacle Avoidance (ϕ_{obs}): Stay a minimum distance from obstacles.

(2) Battery Safety (ϕ_{bat}): Land safely when the battery is low.

How to provide safety guarantees?

Simplex Architecture for Run-time Assurance



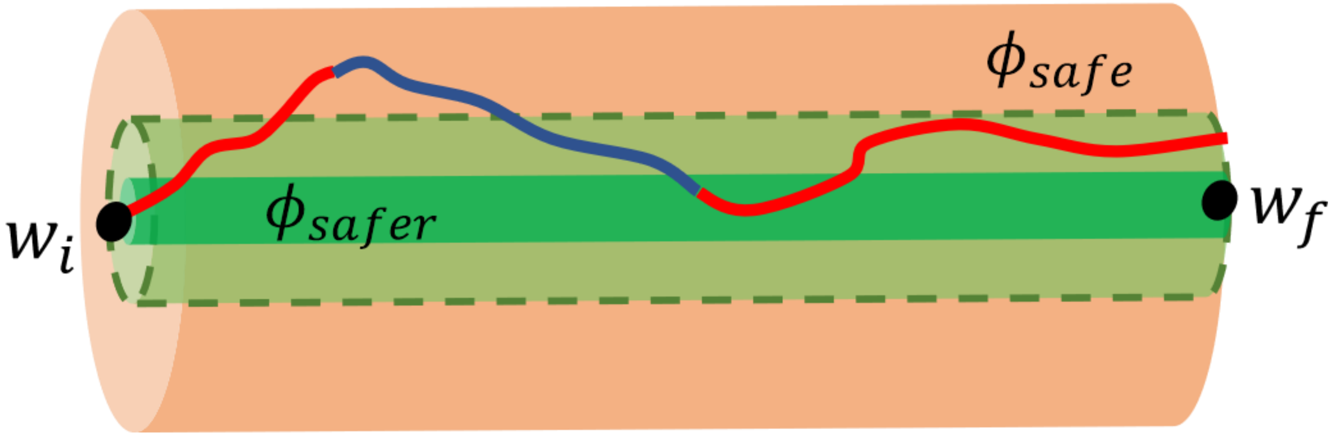
“The simplex architecture for safe on-line control system upgrades” [Lui Sha, RTSS’98]

Runtime Assurance (RTA) Module

An RTA Module is a tuple $(M_{ac}, M_{sc}, \phi_{safe}, \phi_{safer}, \Delta)$

- M_{ac} is the Advanced Controller machine.
- M_{sc} is the Safe (certified) Controller machine.
- ϕ_{safe} is the desired safety specification.
- ϕ_{safer} is a stronger safety specification ($\phi_{safer} \subseteq \phi_{safe}$).
- Δ is the sampling rate of the DM.

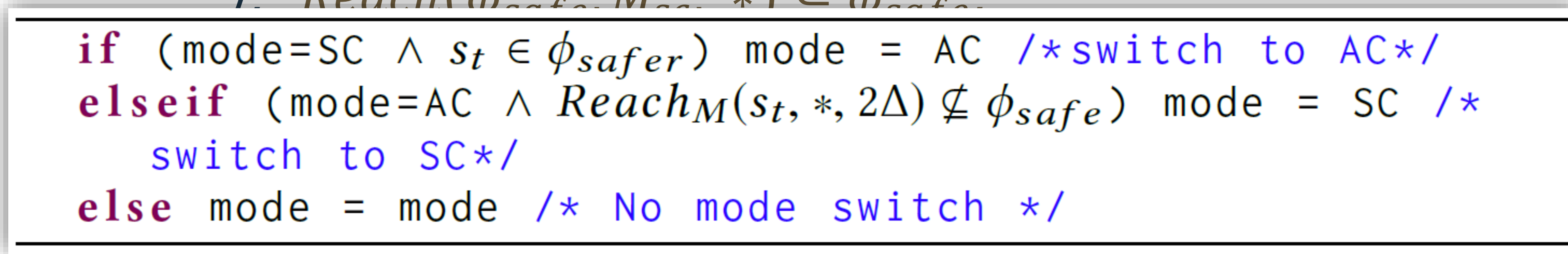
RTA-Protected Motion Primitive



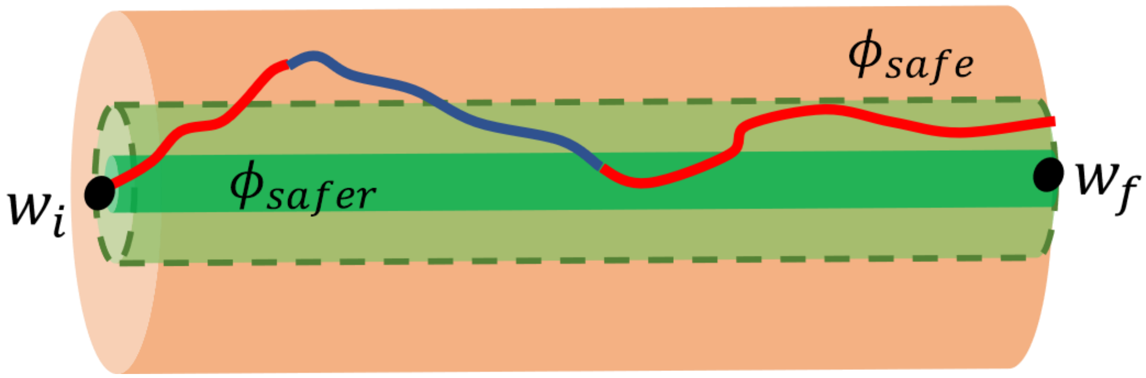
A RTA machine is well-formed

An RTA Module $(M_{ac}, M_{sc}, \phi_{safe}, \phi_{safer}, \Delta)$ is well-formed:

- Outputs of M_{ac} and M_{sc} are the same.
- M_{ac} and M_{sc} have same period (<= Δ).
- The M_{sc} satisfies the following properties:
    1. $Reach(\phi_{safe}, M_{sc}, *) \subseteq \phi_{safe}$;



```
if (mode=SC ∧ $s_t \in \phi_{safer}$) mode = AC /*switch to AC*/
elseif (mode=AC ∧ $Reach_M(s_t, *, 2\Delta) \not\subseteq \phi_{safe}$) mode = SC /*
   switch to SC*/
else mode = mode /* No mode switch */
```



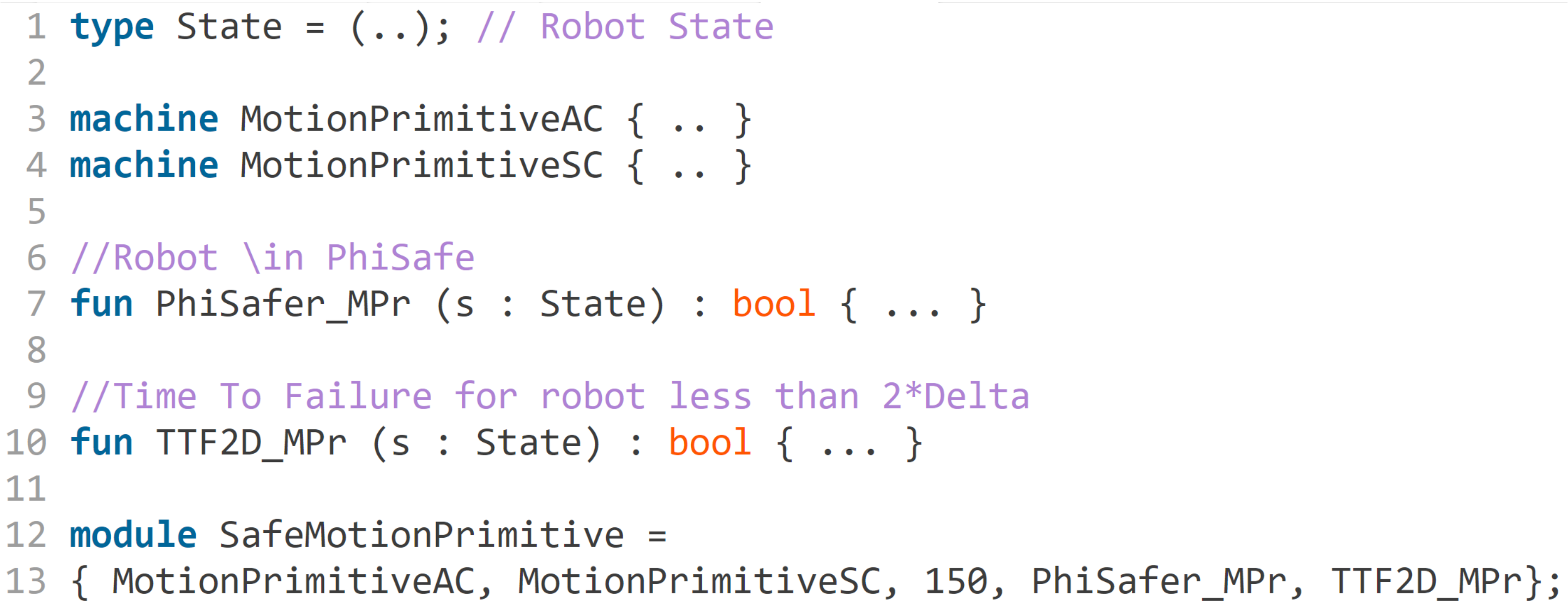
Theorem: The following is an inductive invariant:

Mode = SC $\wedge$ s $\in$ φ_{safe}

$\vee$

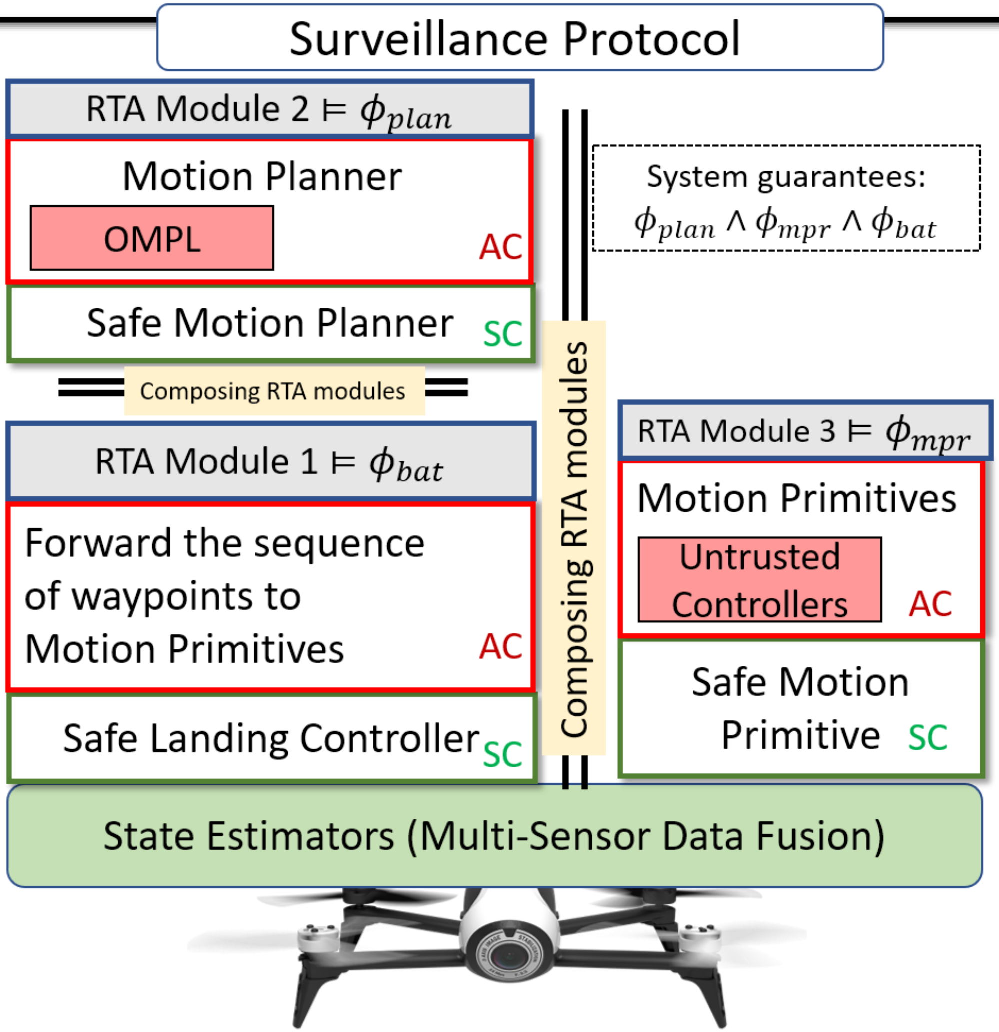
Mode = AC $\wedge$ Reach(s,*,Δ) $\in$ φ_{safe}

Declaring RTA Module



```
type State = (..); // Robot State

machine MotionPrimitiveAC { .. }
machine MotionPrimitiveSC { .. }

//Robot \in PhiSafe
fun PhiSafer_MPr (s : State) : bool { ... }

//Time To Failure for robot less than 2*Delta
fun TTF2D_MPr (s : State) : bool { ... }

module SafeMotionPrimitive =
{ MotionPrimitiveAC, MotionPrimitiveSC, 150, PhiSafer_MPr, TTF2D_MPr};
```

Compositional Runtime Assurance



Surveillance Protocol

RTA Module 2 $\models \phi_{plan}$

Motion Planner

OMPL

AC

Safe Motion Planner SC

Composing RTA modules

RTA Module 1 $\models \phi_{bat}$

Forward the sequence of waypoints to Motion Primitives

AC

Safe Landing Controller SC

Composing RTA modules

System guarantees:

$\phi_{plan} \wedge \phi_{mpr} \wedge \phi_{bat}$

RTA Module 3 $\models \phi_{mpr}$

Motion Primitives

Untrusted Controllers

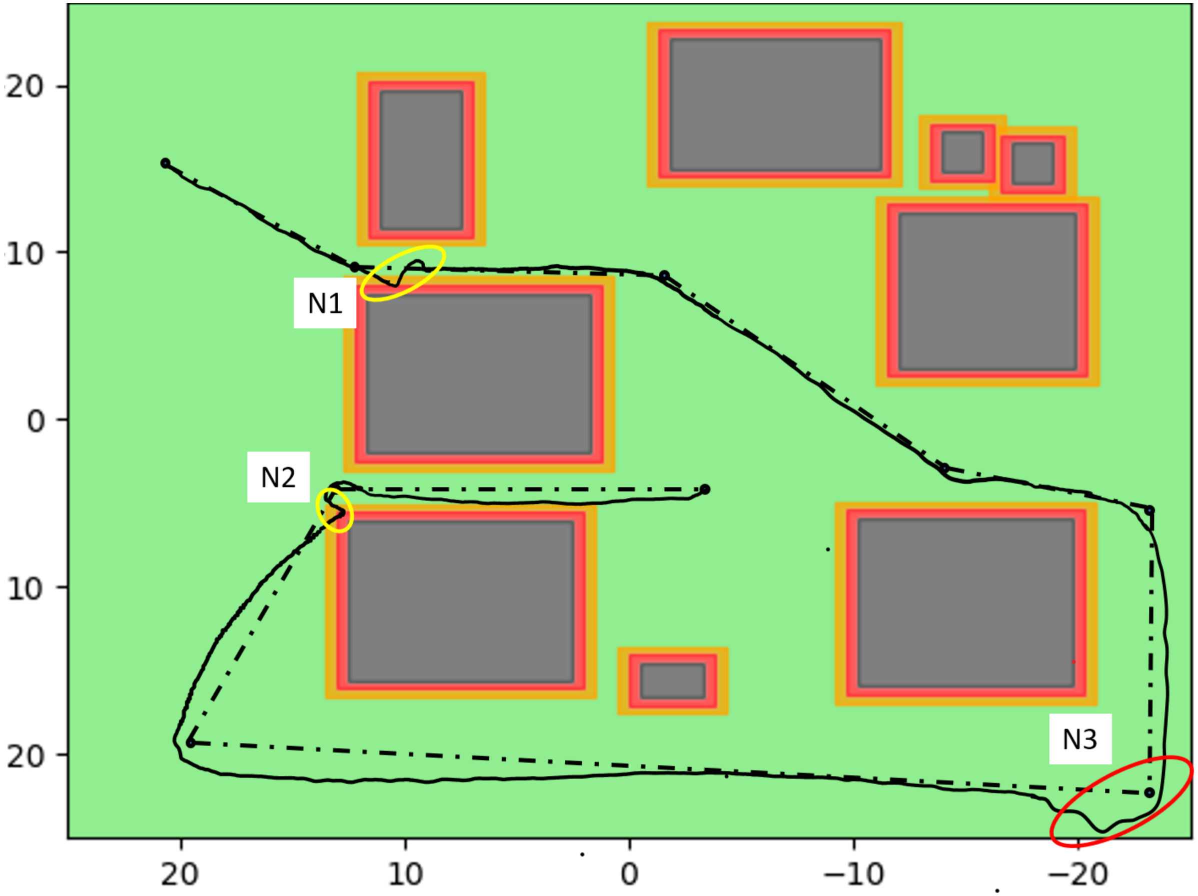
AC

Safe Motion Primitive SC

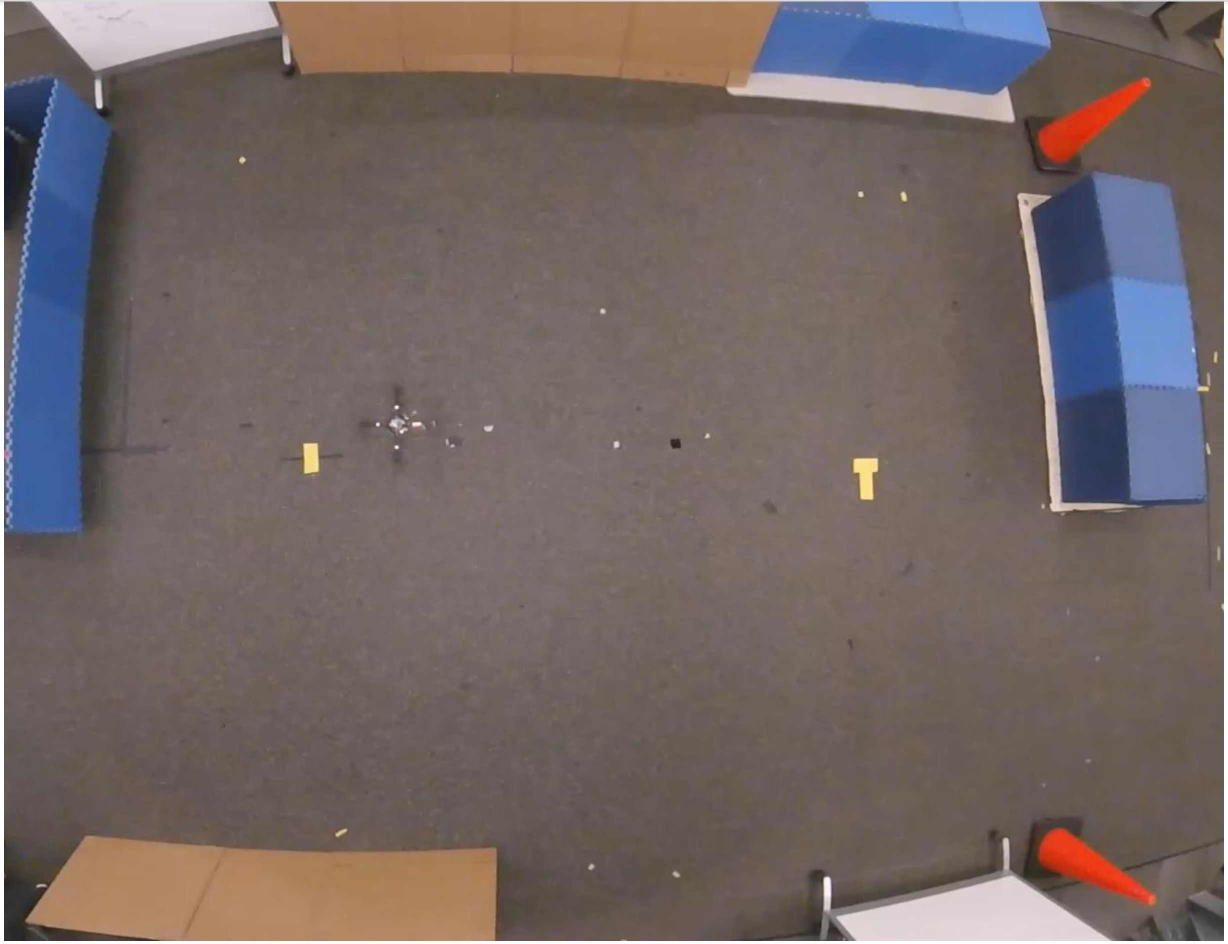
State Estimators (Multi-Sensor Data Fusion)

```
module system = RTAModule1 || RTAModule2 || RTAModule3;
```

RTA-Protected Robotics Mission



Safe Exploration



Darpa Demo in Collaboration with UPenn.

Rigorous Simulations

- Simulated the surveillance system for 104 hours.
- Total distance : ≈ 1500KM.
- Total disengagement (from AC to SC): 109.
- Total crashes : 34.
- Scheduling issues: 31.

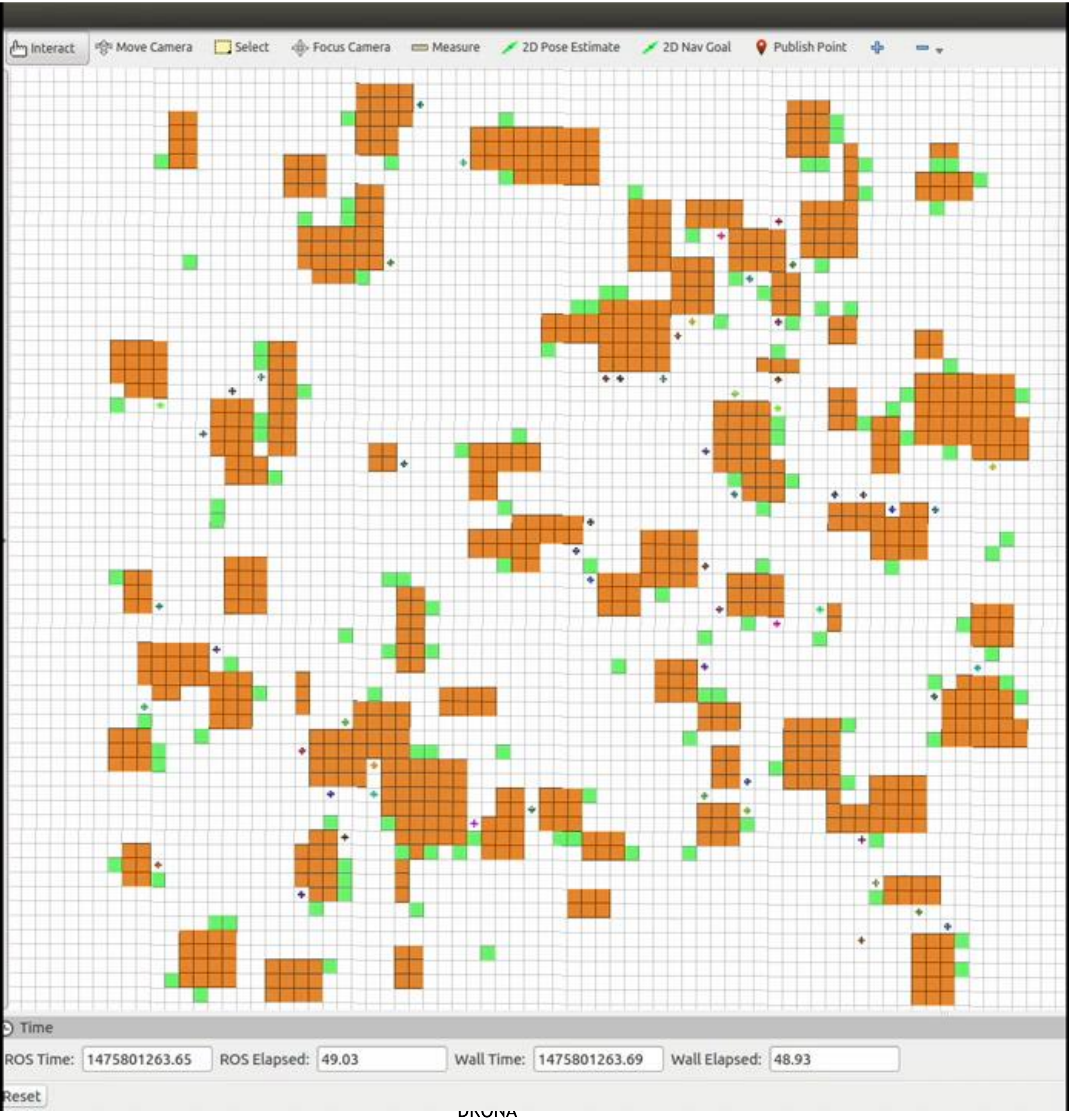
Conclusion

Two challenges: (1) Reactivity (2) Untrusted Components.

Solution: Combining design-time approaches like programming languages and verification (software and controller) with runtime assurance.

We can get the desired “end-to-end correctness”*

*under certain assumption ☺



ROS Time: 1475801263.65 ROS Elapsed: 49.03 Wall Time: 1475801263.69 Wall Elapsed: 48.93