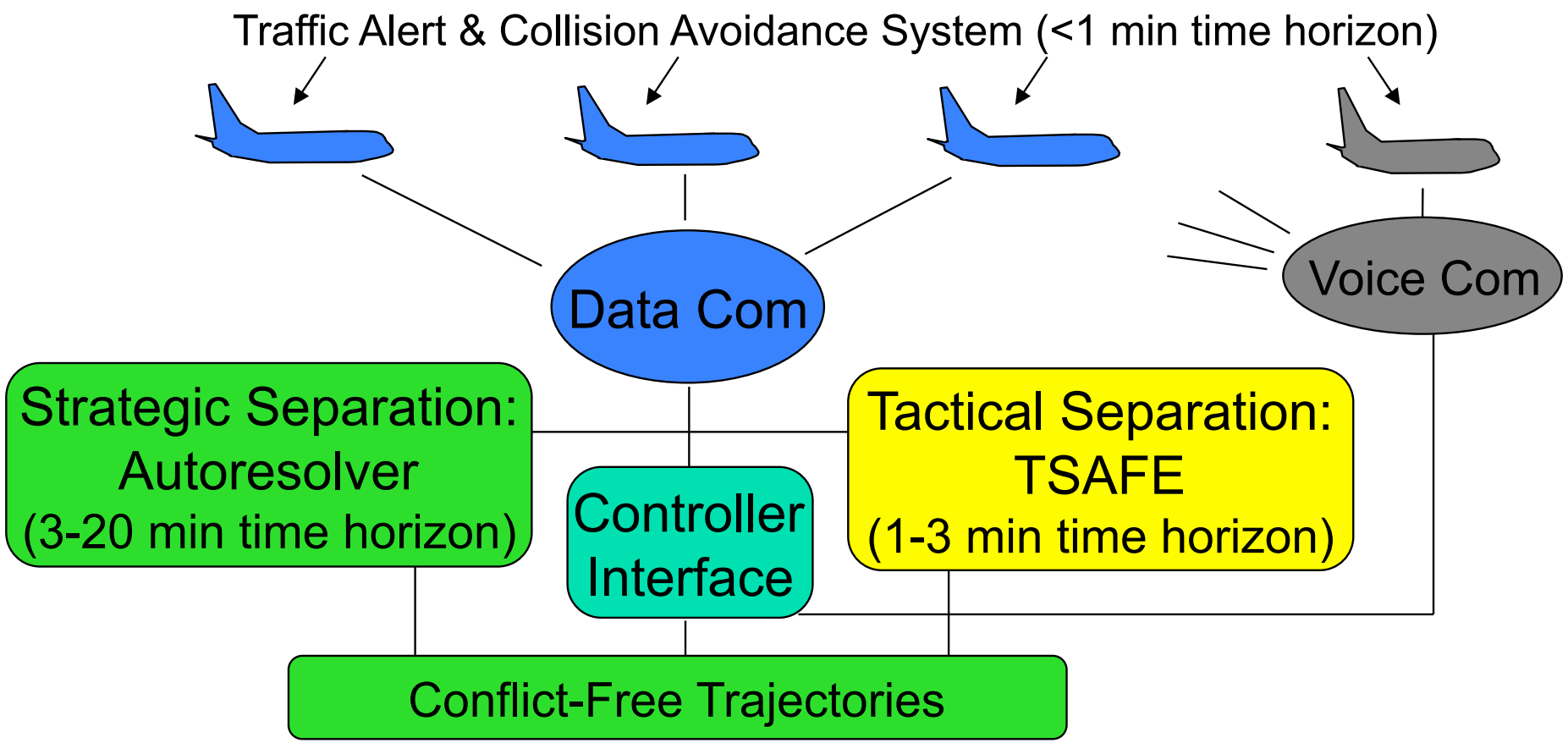
Runtime Verification (vs Model Checking, Simulation)

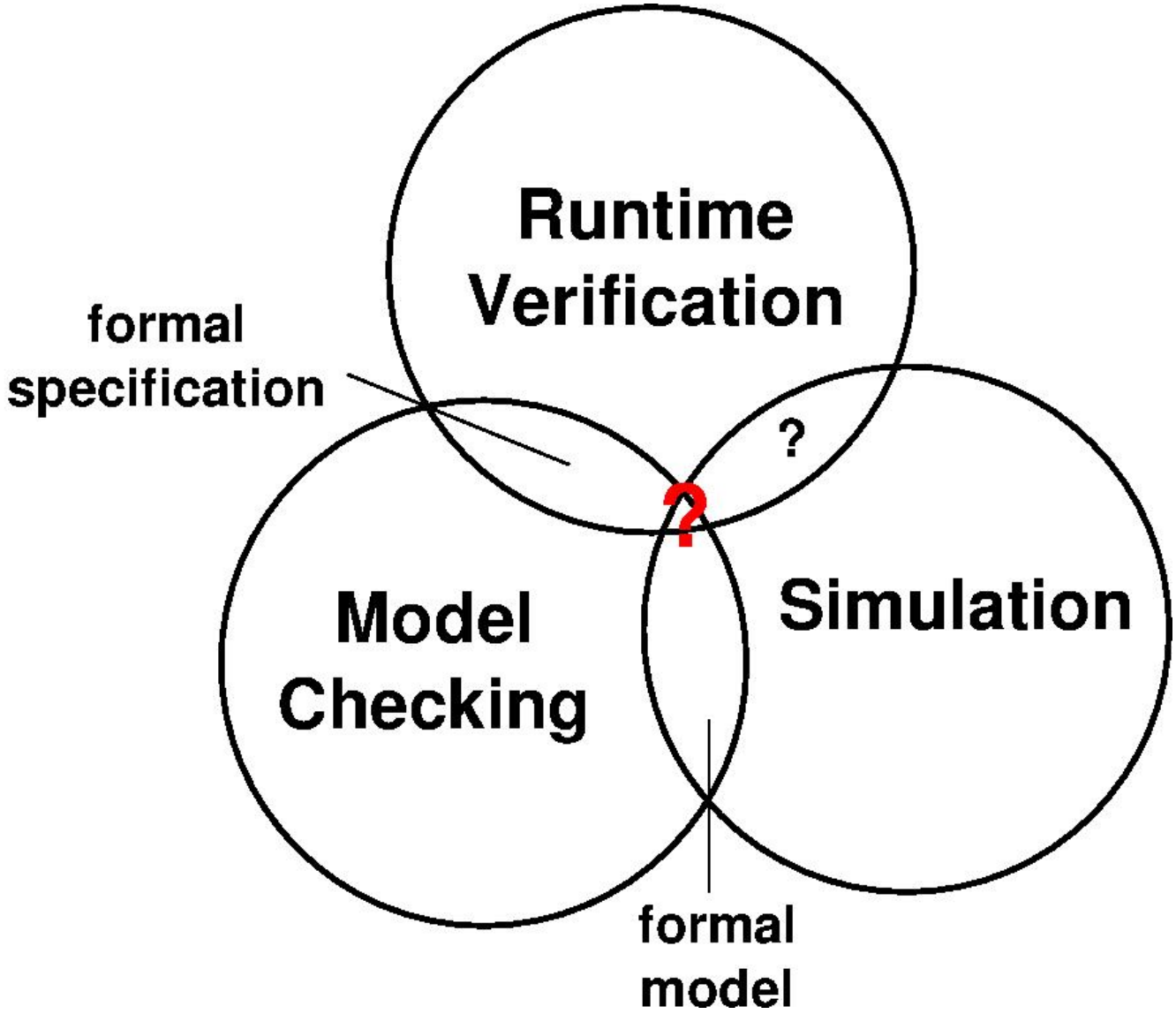
Kristin Yvonne Rozier
Iowa State University

ECNU Summer School
August 23/26, 2021

Motivation: Verification of the Automated Airspace Concept[1]



[1] H. Erzberger, K. Heere, Algorithm and operational concept for resolving short-range conflicts, Proc. IMechE G J. Aerosp. Eng. 224 (2) (2010) 225243


Runtime
Verification
formal
specification
?
?
Model
Checking
Simulation
formal
model

RV is a Semi-Formal Method...

"Formal Methods" are mathematically rigorous techniques for the specification, design, and verification of systems.

Formal Specification:

- System requirements (properties) in mathematical logic

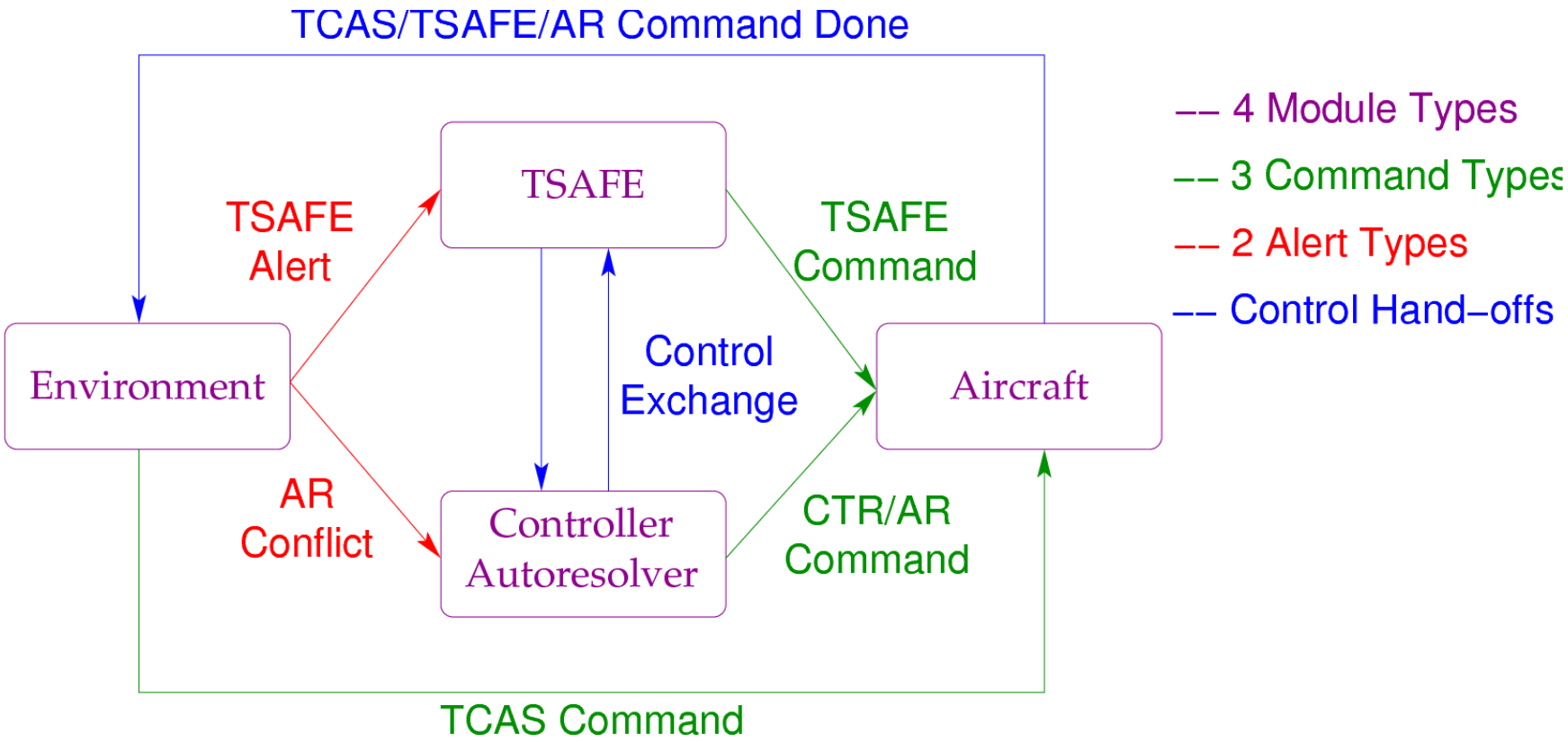
Formal Verification:

- Check that requirements always hold in a system (specified in a logical language)
- Examine the entire state space (all possible inputs)
- Provide absolute assurance of a correctness or safety property

Given formal definitions of what a system does (M) and what it should do (φ), formal methods can be used to show that M *satisfies* φ.

Intuitively, the system does what you think it should do and nothing else.

System Modeling: Overview



- Treat system components like black boxes
- Abstract like-components i.e., multiple planes

Model Checking vs Runtime Verification

Model Checking (MC)

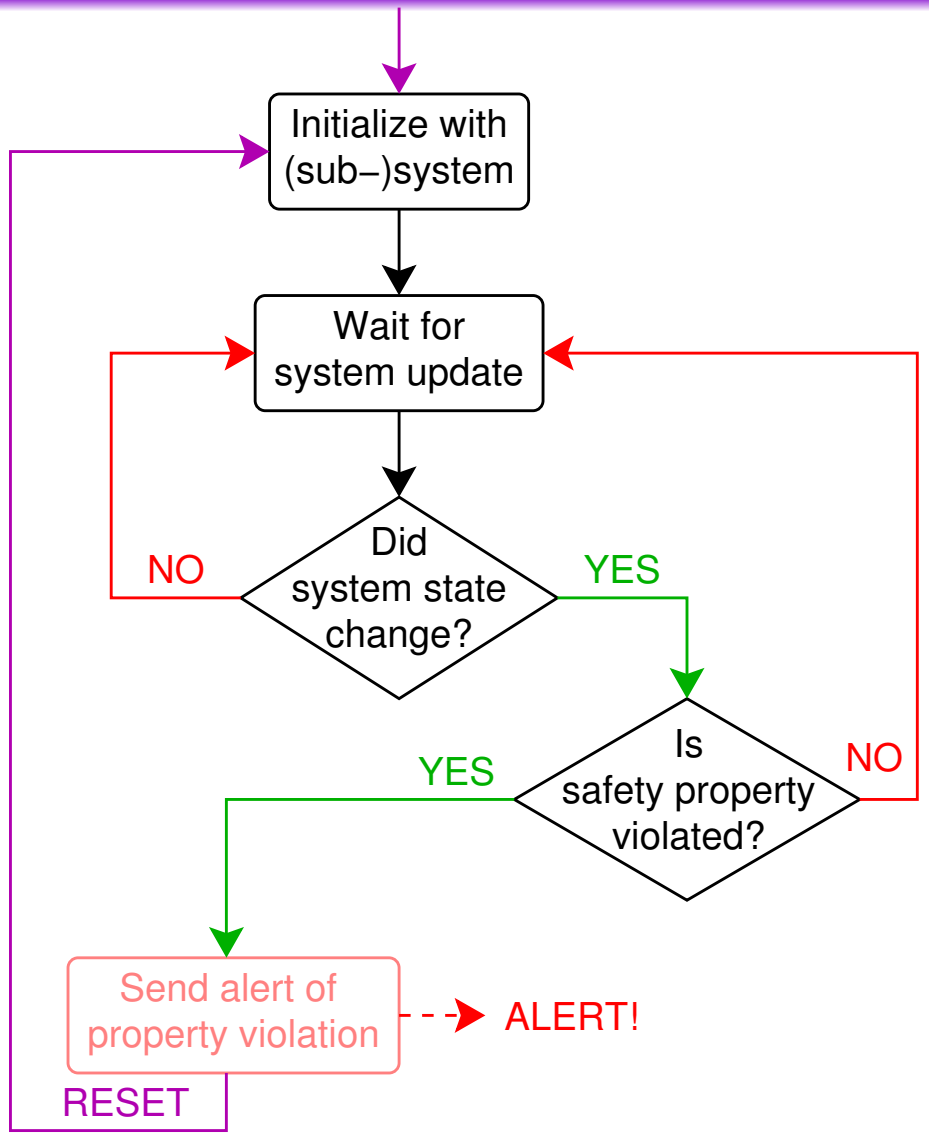
- Checks **all possible runs** of the system
- Needs an accurate system model as input
- Performed at design time for system **revision/debugging**
- Computationally challenging; runs can take hours or days
- Purpose: design refinement/debugging

Runtime Verification (RV)

- Checks only **the current run** of the system
- Needs **no system model**; the system is its own model
- Performed during system operation, usually for **mitigation triggering**[2]
- Must be computationally efficient, even real time
- Purpose: execution understanding; NOT debugging

[2] Other applications include black-box recording, sensor filtering, diagnostics, post-mission analysis, and aiding (but NOT contributing) fault-tolerance, replanning

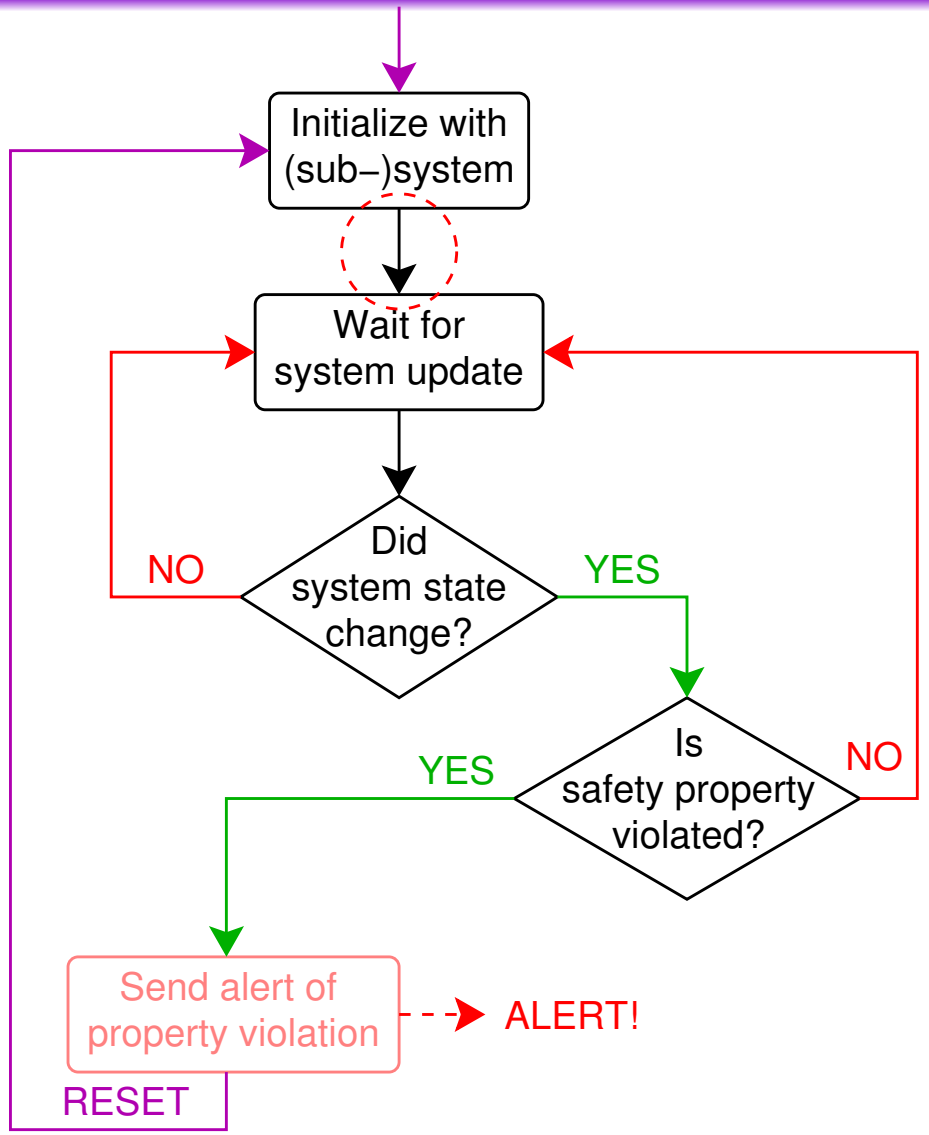
Runtime Monitoring



Research: state-of-the-art

- can check complex temporal properties *very* efficiently
- low overhead
- real-time
- enables resets, exits from infinite loops, etc.

Runtime Monitoring



Research: state-of-the-art

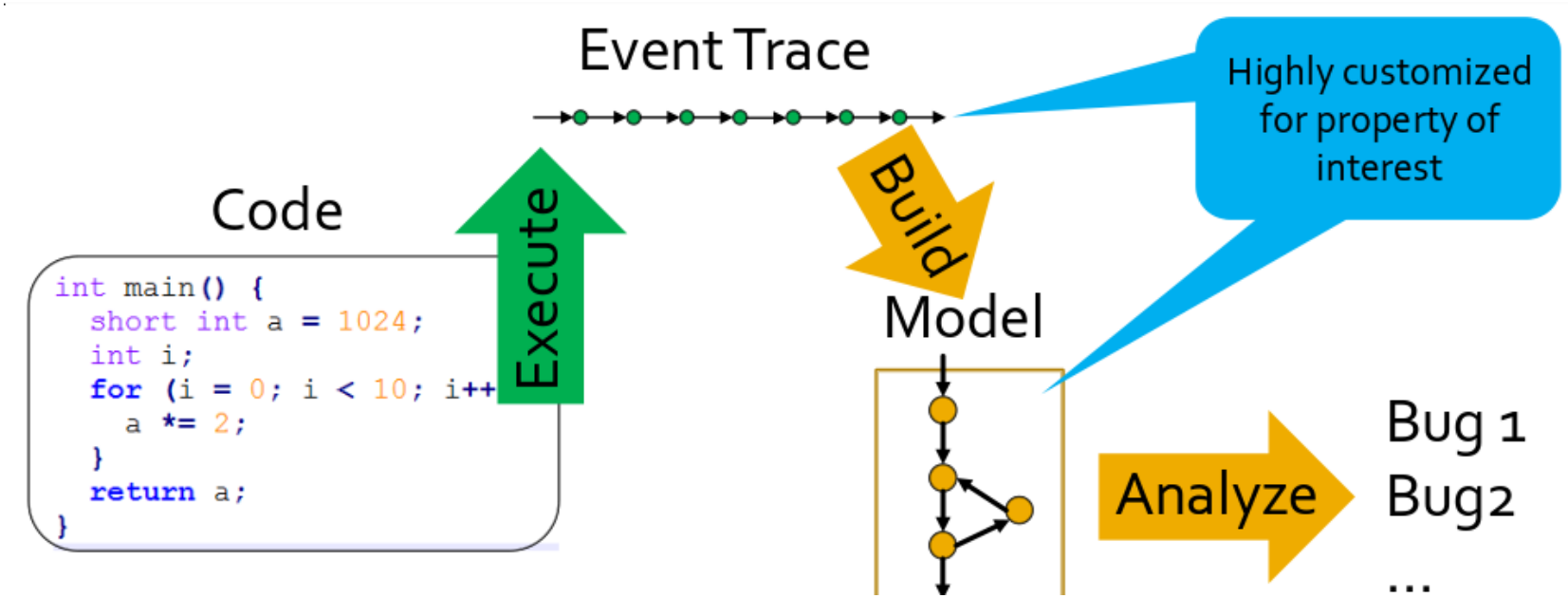
- can check complex temporal properties *very* efficiently
- low overhead
- real-time
- enables resets, exits from infinite loops, etc.
- requires instrumentation to send state variables to monitor

From Simulation to Runtime Verification and Back: Connecting Single-Run Verification Techniques[3]

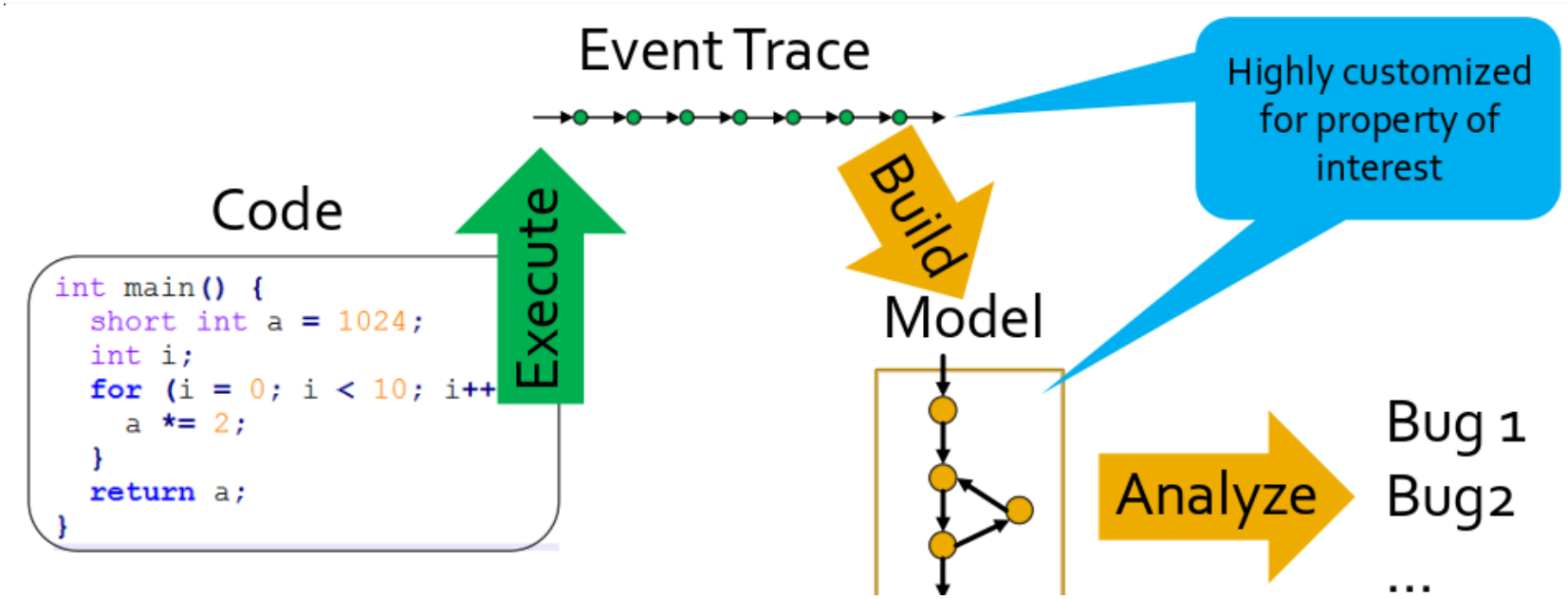
Runtime Verification is a single-run (non-exhaustive) (semi-formal) verification technique.

[3] Kristin Yvonne Rozier. From Simulation to Runtime Verification and Back: Connecting Single-Run Verification Techniques. In 2019 Spring Simulation Conference (SpringSim19). Tucson, Arizona, April 29 May 2, 2019.

Simulation or Runtime Verification?



Simulation or Runtime Verification?



4

[4] Grigore Rosu and Klaus Havelund, 2001, https://www.runtimeverification.com/presentations/

Simulation

focus on:
discrete-event stochastic simulation

Simulation[5]

Definition:
A **discrete-event simulation model** is defined by three attributes:

- **stochastic** – at least some of the system state variables are random;
- **dynamic** – the time evolution of the system state variables is important;
- **discrete-event** – significant changes in system state variables are associated with events that occur at discrete time instances only.

[5] [PL06] Leemis, L. M., and S. K. Park. 2006. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ.

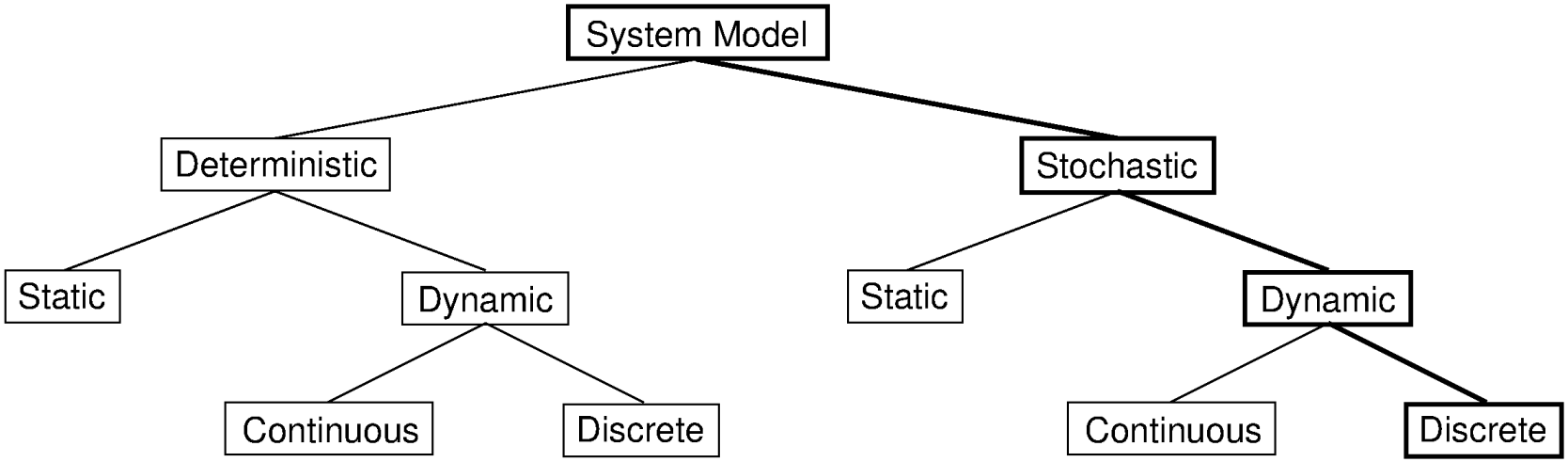


Figure: Characterization of a simulation system model in the form of a tree diagram. We focus on the right-most branch of the tree.

Developing a System Model[6]

Algorithm 0: **Developing a System Model**. Iterate through Steps 2–6 until a valid computational model (executable hardware or software) has been created: "as simple as possible, but never simpler."

1. Decide system goals and objectives, e.g., Boolean analysis (which faults need to be detected and which fault signatures to consider)
2. Build a **conceptual model**: state variables and their relationships.
3. Turn the conceptual model into a **specification model**, e.g., through collecting and analyzing data or otherwise deriving a representative model of relevant system behaviors

[6] *developing a discrete-event simulation model* in [PL06] Leemis, L. M., and S. K. Park. 2006. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ.

Developing a System Model[7]

Algorithm 1: **Developing a System Model**. Iterate through Steps 2–6 until a valid computational model (executable hardware or software) has been created: "as simple as possible, but never simpler."

4. Turn the specification model into a **computational model**, in the form of executable hardware or software.
5. Verify that the computational model is correct.
6. Validate that the computational model is consistent with the system under test.

[7] *developing a discrete-event simulation model* in [PL06] Leemis, L. M., and S. K. Park. 2006. *Discrete-event simulation: A first course*. Pearson Prentice Hall Upper Saddle River, NJ.

Related to Algorithm 1 Step 3's conceptual model, the following two definitions hold for both simulation and runtime verification[8]

Definition:
A **system state** is a complete characterization of the system at an instance in time, usually represented as an assignment to the complete set of system variables.

[8] Versions of these appear as Definitions 5.1.1 and 5.1.2 in PL06 as well as nearly every paper on runtime verification.

Definition: An **event** is an occurrence that changes the system state, e.g., by altering the assignment to the system variables. Only an event can result in a state change.

Definition:
A **fault** is a deviation between the behavior in a system execution and the expected behavior, as defined by safety requirements.

- can occur in software, hardware, or a combination
- can be a system state that should be unreachable
- can be a path through multiple system states that should not be followed

A fault might lead to a failure, but not necessarily.[9] Differently, an error is a mistake made by a human that results in a fault and possibly in a failure.

[9] Leucker, M., and C. Schallhart. 2009. A brief account of runtime verification. The Journal of Logic and Algebraic Programming vol. 78 (5), pp. 293303.

Runtime Verification

focus on: *online, stream-based discrete-time runtime verification*

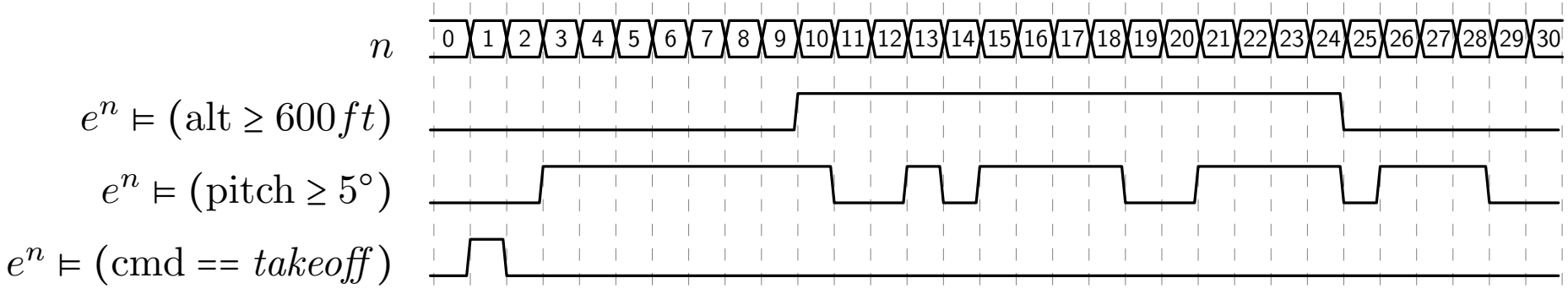
Definition:
Online, stream-based discrete-time runtime verification is defined by three attributes:

- **online** – the RV engine executes at the same time (synchronously or asynchronously) with the system under verification, during deployment, observing the current execution and part of its history
- **stream-based** – the output of the RV engine is a stream of $\langle time, verdict \rangle$ tuples, evaluating for every time t in the finite system execution whether the execution starting at time t satisfies the monitored requirement (as opposed to outputting a single such verdict for the total execution from start to finish);
- **discrete-time** – fault signatures are described by temporal logic formulas (or, equivalently, automata) over discrete time instances only, such as sensor signals.

Asynchronous Observers (aka event-triggered)

- *evaluate with every new input*
- 2-valued output: {true; false}
- resolve φ *as early as possible* (a priori known time)
- for each clock tick, may resolve φ for clock ticks prior to the current time n if the information required for this resolution was not available until n

Asynchronous Observers Example



$$\text{ALWAYS}_{[5]}(\text{pitch} \geq 5°)$$

0	(false,0)	8	(true,3)
1	(false,1)	9	(true,4)
2	(false,2)	10	(true,5)
3	(␣, ␣)	11	(false,11) Resynchronized!
4	(␣, ␣)	12	(false,12)
5	(␣, ␣)	13	(␣, ␣)
6	(␣, ␣)	14	(false,14) Resynchronized!
7	(␣, ␣)	15	(␣, ␣)

Comparably to discrete-event stochastic simulation runtime verification reasons about systems that are:

- **stochastic** – at the very least the environment is always a stochastic component of the system that we have to address, and the system faults RV aims to detect are largely random failures;
- **dynamic** – RV reasons over temporal traces of system execution;
- **discrete-event** – while there are some RV specification logics that reason over continuous time (e.g., Signal Temporal Logic), the sensor signals that serve as input to RV engines are inherently discrete.

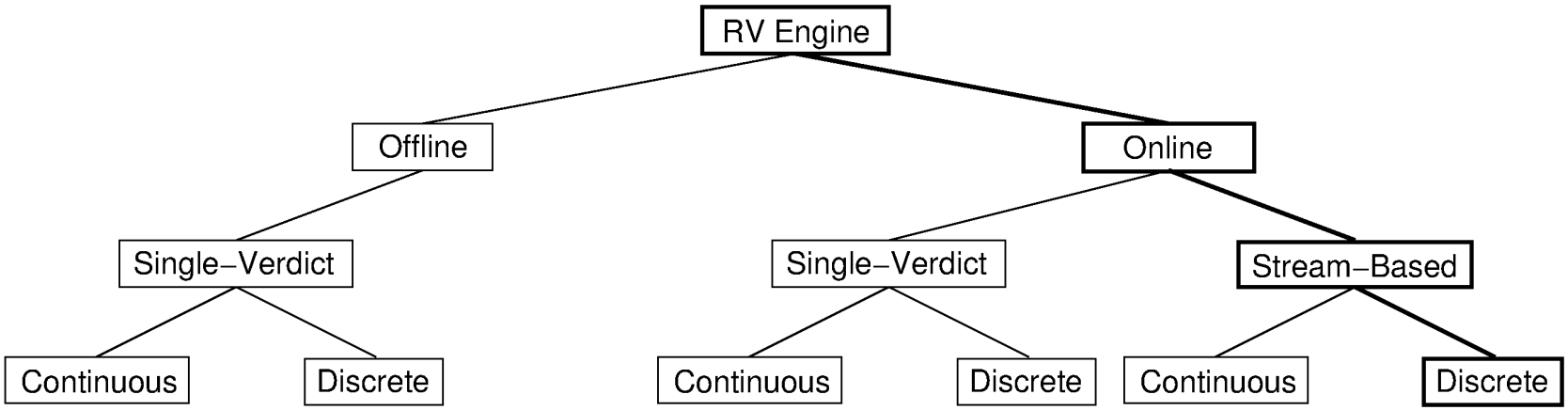


Figure: Characterization of a runtime verification engine paradigm in the form of a high-level tree diagram We focus on the right-most branch of the tree

Algorithm 2[10]: Developing a Runtime Verification Engine. Iterate until reaching required robustness (verification and validation) targets

1. **Specifying (Un)Desired System Behavior** – (Alg. 1, Steps 1-3), defining verification objectives, system variables, correct level of abstraction, and faithful representation in a specification model.
2. **Producing a Monitor from a Specification** – (Alg. 1, Steps 4-6), make executable runtime verification engine, (in software or programmable hardware), formally proving its correctness, and validating this engine.
3. **Connecting a Monitor to a System** – execution of RV engine running in parallel with the system depends upon connecting input signals

[10] Bartocci, E., Y. Falcone, A. Francalanza, and G. Reger. 2018. Introduction to runtime verification. In Lectures on Runtime Verification, pp. 133. Springer.

Common Objectives

- *correctness* (always behaving the way we expect)
- *safety* (not violating requirements)
- *performance-as-safety* (upholding minimum requirements for safe real-time operation such as responsiveness or throughput).

Computational Model Format

(corresponding to Algorithm 1, Step 4)

- reason about single executions of the system
  - called *traces* or *runs*
  - differ in the analysis performed on these
- aim for software implementations
- utilize same/similar fault models

Notion of Time

- can represent time and the transition between system states
- can be either event-triggered or time-triggered (event-triggered more common)
- RV doesn't represent clocks in the same way as simulations, but can usually define equivalences
  - counters
  - reliance on checks of external-to-the-RV-engine system clocks

Verification

- extensive testing
- comparison to expected outcomes defined by the human modeling the system
- RV allows for more rigorous, and therefore more automated, verification efforts, e.g., create RV test-cases with a priori known exact correct answers

Validation

- consistency checks: do changes in the specification result in the expected changes in the outputs?
- comparison with individual system runs on the real system for validation

Purpose

The purpose of simulation is insight [11] whereas the purpose of RV is fault detection [12].

[11] Leemis, L. M., and S. K. Park. 2006. Discrete-event simulation: A first course. Pearson Prentice Hall Upper Saddle River, NJ.

[12] Leucker, M., and C. Schallhart. 2009. A brief account of runtime verification. The Journal of Logic and Algebraic Programming vol. 78 (5), pp. 293303.

RV/Simulation

Simulation:

- design-time verification technique
- executed offline
- used to improve the system before deployment
- characterizing system performance, understanding subtle system features, discovering component interactions, improving responses, maximizing profit

RV:

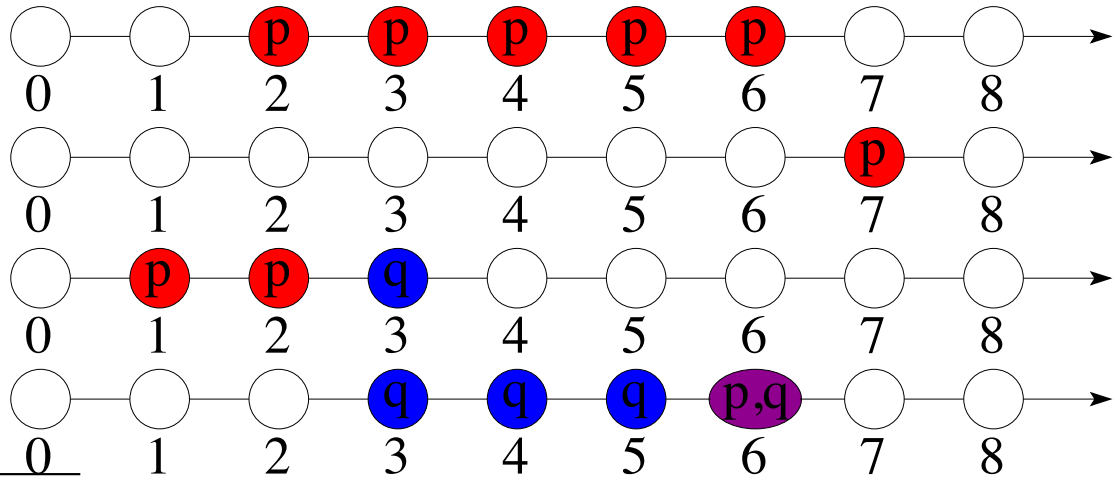
- runtime verification technique
- used to detect deviations from nominal operation online, in real time, to enable mitigation actions
- early-as-possible identification or prediction of faults or partial faults
- robustify system operation, ensure safety/compliance with requirements, enable certification

Modeling/Specification Language , e.g., MLTL [13]

Mission-Time Temporal Logic (MLTL) reasons about *integer-bounded* timelines:

- finite set of atomic propositions {p q}
- Boolean connectives: ¬, ∧, ∨, and →
- temporal connectives *with time bounds*:

Symbol	Operator
$\Box_{[2,6]} p$	ALWAYS$_{[2,6]}$
$\Diamond_{[0,7]} p$	EVENTUALLY$_{[0,7]}$
$p \mathcal{U}_{[1,5]} q$	UNTIL$_{[1,5]}$
$p \mathcal{R}_{[3,8]} q$	RELEASE$_{[3,8]}$



[13] T. Reinbacher, K.Y. Rozier, J. Schumann. "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems." TACAS 2014.

Outputs

Simulation:

- characterize system executions in the aggregate
- job- (or customer- or event-) averaged statistics: average delays, average interarrival times, or average service times
- time-averaged statistics: utilization

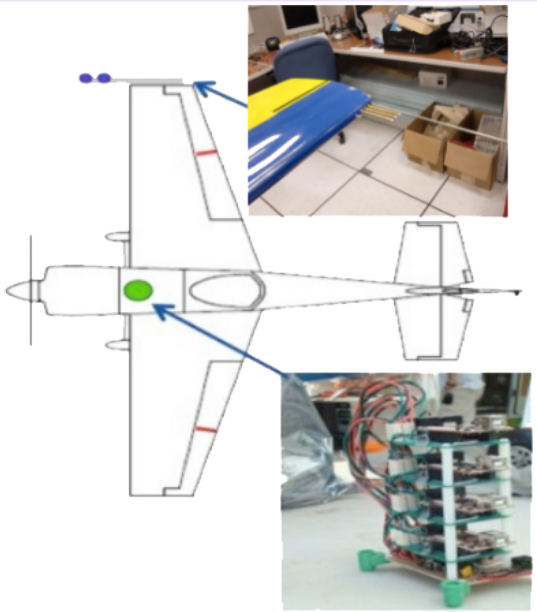
RV:

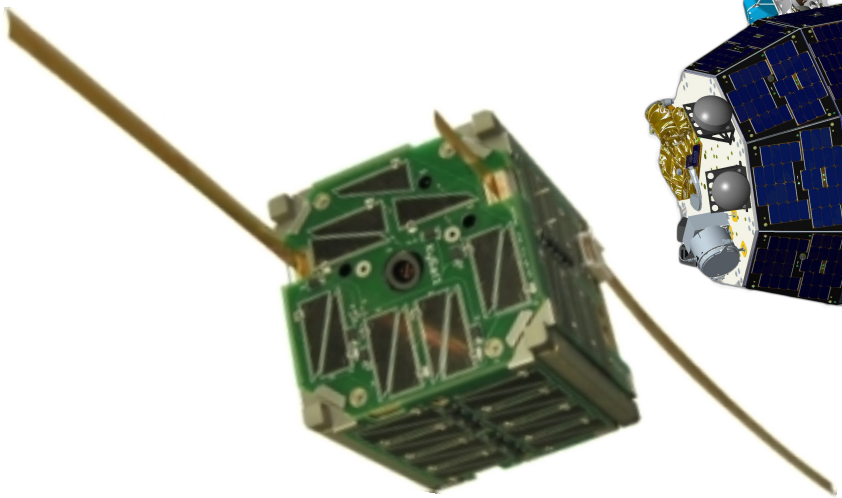
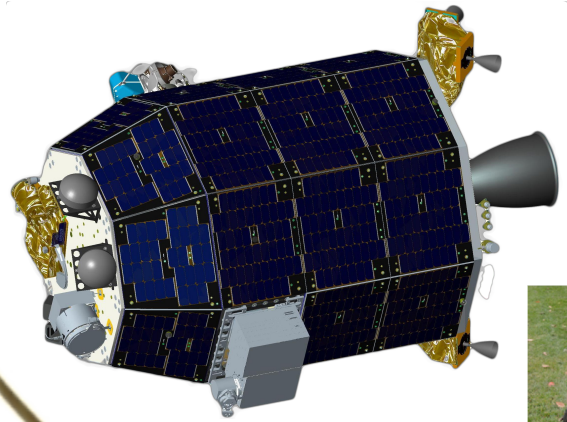
- checks one system execution (the current one)
- Boolean verdict

Opportunities for validation of simulation via RV: using RV for extracting statistics

Simulations could generate traces with Boolean-valued properties to validate RV frameworks

Autonomy: the Future of Aerospace (and Beyond . . .)




A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...
Crash of ESA's ExoMars Schiaparelli Lander

- October 19, 2016
- parachute deployed at:
  - altitude of 7.5 miles (12 km)
  - speed of 1,1075 mph (1,730 km/h)
- heat shield ejected at altitude of 4.85 miles (7.8 km)
- IMU miscalculated saturation-maximum period (by 1 sec)
- Navigation system calculated a *negative altitude*
  - premature release of parachute & backshell
  - firing of braking thrusters
  - activation of on-ground systems at 2 miles (3.7 km) altitude
- Crash at 185 mph (300 km/h)

A Recent Motivation...

Crash of ESA's ExoMars Schiaparelli Lander

Sanity Checks Relevant to this Mission:

- The altitude cannot be negative.
- The rate of change of descent can't be faster than gravity.
- The δ altitude must be within nominal parameters; it cannot change from 2 miles to a negative value in one time step.
- The saturation-maximum has an a priori known temporal bound.

These *sanity checks* could have prevented the crash.

Capability of such observations is *required for autonomy*.

Runtime Verification: Required for Autonomy & Future CPS

How do we fit RV into resources on-board already-flying CPS?

Satisfying Requirements

Satisfying Requirements

RESPONSIVE
REALIZABLE
UNOBTRUSIVE
Unit

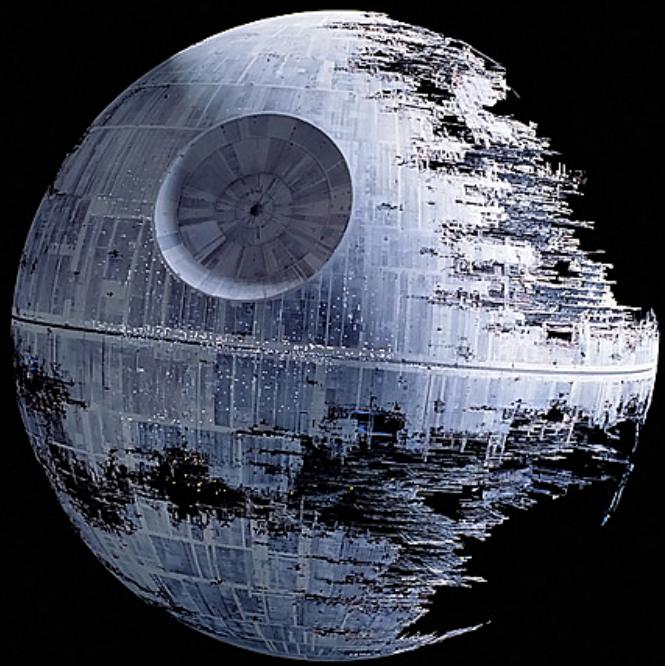
R2U2

Runtime Monitoring On-Board

Adding currently available runtime monitoring capabilities to the UAS would change its flight certification.

"Losing flight certification is like moving over to the dark side: once you go there you can never come back."

— Doug McKinnon,
NASA Ames' UAS Crew Chief



Requirements

Realizability:

- easy, *expressive* specification language
- *generic* interface to connect to a wide variety of systems
- *adaptable* to missions, mission stages, platforms

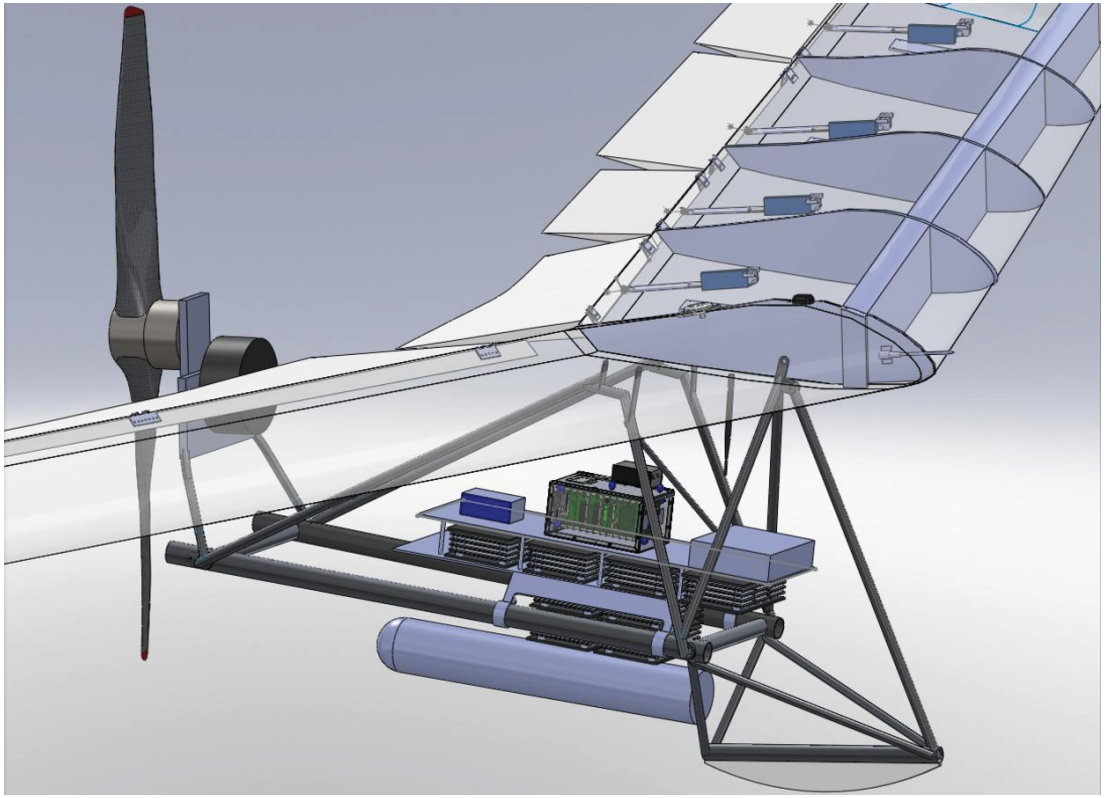
Responsiveness:

- *continuously monitor* the system
- *detect deviations* in *real time*
- *enable mitigation* or rescue measures

Unobtrusiveness:

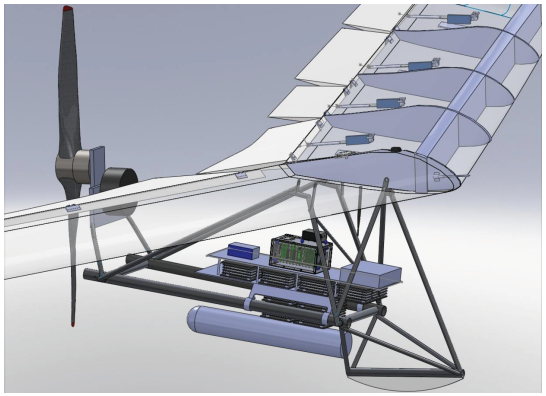
- *functionality*: not change behavior
- *certifiability*: avoid re-certification of flight software/hardware
- *timing*: not interfere with timing guarantees
- *tolerances*: obey size, weight, power, telemetry bandwidth constraints
- *cost*: use commercial-off-the-shelf (COTS) components

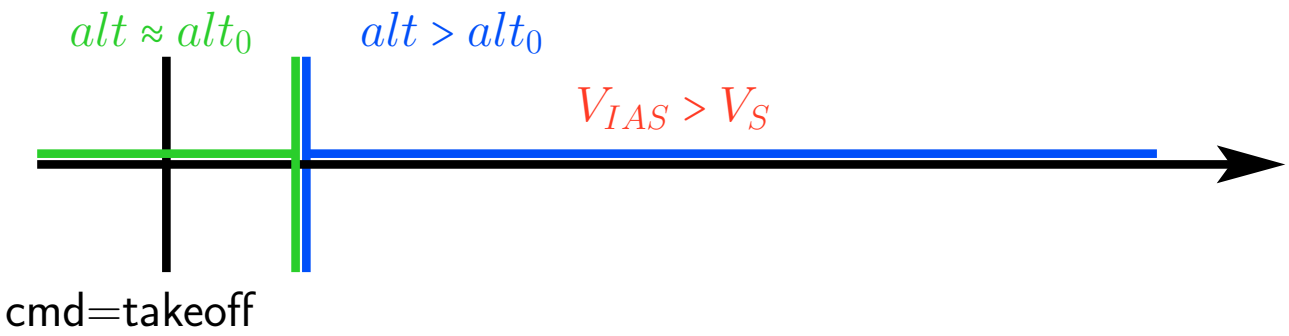
Example: NASA's Swift UAS



13 meter wingspan all-electric experimental platform

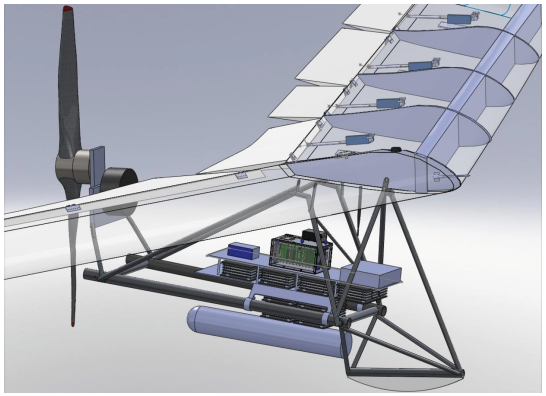
Runtime Observers for the Swift UAS

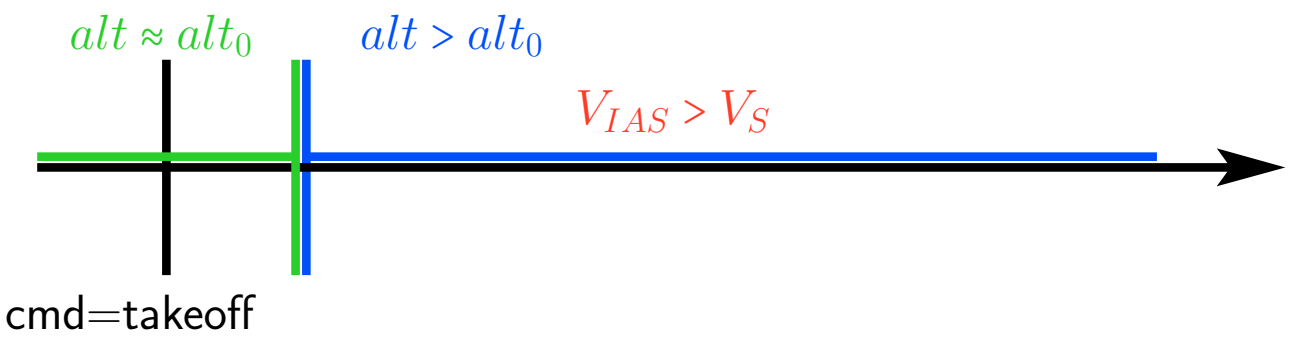




Whenever the Swift UAS is in the air, its indicated airspeed (V_{IAS}) must be greater than its stall speed V_S. The UAS is considered to be air-bound when its altitude *alt* is larger than that of the runway alt_0.

Runtime Observers for the Swift UAS





Whenever the Swift UAS is in the air, its indicated airspeed (V_{IAS}) must be greater than its stall speed V_S. The UAS is considered to be air-bound when its altitude *alt* is larger than that of the runway alt_0.

$$\text{ALWAYS}((alt > alt_0) \rightarrow (V_{IAS} > V_S))$$

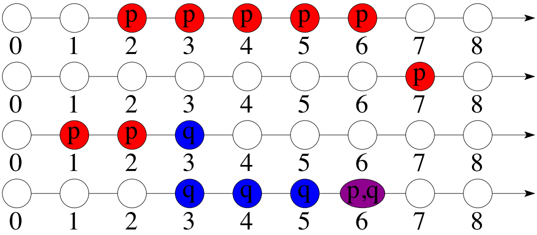
Encoding Timelines: Linear Temporal Logic

Mission-time LTL (MLTL) reasons about *bounded* timelines:

- finite set of atomic propositions {p q}
- Boolean connectives: ¬, ∧, ∨, and →
- temporal connectives *with time bounds*:

Symbol	Operator
$\Box_{[2,6]} p$	ALWAYS$_{[2,6]}$
$\Diamond_{[0,7]} p$	EVENTUALLY$_{[0,7]}$
$p \mathcal{U}_{[1,5]} q$	UNTIL$_{[1,5]}$
$p \mathcal{R}_{[3,8]} q$	RELEASE$_{[3,8]}$

Timeline



Mission-bounded LTL is an over-approximation for mission time τ

Runtime Monitoring for the Swift UAS

The precision of the position reading P_{GPS} from the GPS subsystem depends on the number of visible GPS satellites N_{sat}.

Runtime Monitoring for the Swift UAS

The precision of the position reading P_{GPS} from the GPS subsystem depends on the number of visible GPS satellites N_{sat}.

$$
\begin{aligned}
&\Box(\\
&\Box(N_{sat} == 1) \quad \rightarrow P_{GPS} \leq P^1_{GPS} \quad \wedge \\
&\Box(N_{sat} == 2) \quad \rightarrow P_{GPS} \leq P^2_{GPS} \quad \wedge \\
&\Box(N_{sat} == 3) \quad \rightarrow P_{GPS} \leq P^3_{GPS} \quad \wedge \\
&\Box(N_{sat} \geq 4) \quad \rightarrow P_{GPS} \leq P^+_{GPS})
\end{aligned}
$$

Runtime Monitoring for the Swift UAS

After receiving a command (cmd) for takeoff, the Swift UAS must reach an altitude of 600ft within 40 seconds.

Runtime Monitoring for the Swift UAS

After receiving a command (cmd) for takeoff, the Swift UAS must reach an altitude of 600ft within 40 seconds.

$$\square((\text{cmd} == \text{takeoff}) \rightarrow \Diamond_{[0,40s]}(alt \geq 600\ \text{ft}))$$

Runtime Monitoring for the Swift UAS

All messages sent from the guidance, navigation and control (GN&C) component to the Swift actuators must be logged into the on-board file system (FS). Logging has to occur before the message is removed from the queue. In contrast to the requirements stated above, this flight rule specifically concerns properties of the flight software.

Runtime Monitoring for the Swift UAS

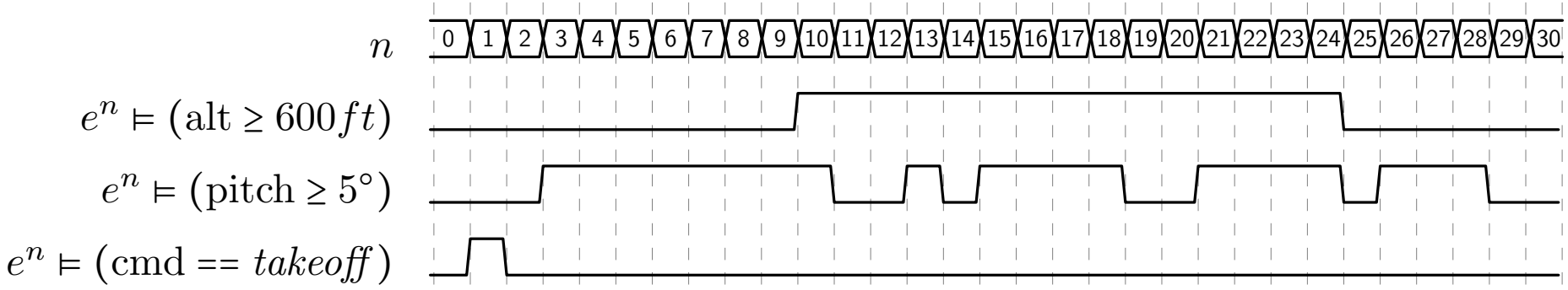
All messages sent from the guidance, navigation and control (GN&C) component to the Swift actuators must be logged into the on-board file system (FS). Logging has to occur before the message is removed from the queue. In contrast to the requirements stated above, this flight rule specifically concerns properties of the flight software.

$$\Box((\text{addToQueue}_{\text{GN\&C}} \wedge \Diamond \text{removeFromQueues}_{\text{Swift}}) \rightarrow \neg\text{removeFromQueues}_{\text{Swift}} \; \mathcal{U} \; \text{writeToFS})$$

Asynchronous Observers (aka event-triggered)

- *evaluate with every new input*
- 2-valued output: {true; false}
- resolve φ *as early as possible* (a priori known time)
- for each clock tick, may resolve φ for clock ticks prior to the current time n if the information required for this resolution was not available until n

Asynchronous Observers Example



$$\text{ALWAYS}_{[5]}(\text{pitch} \geq 5°)$$

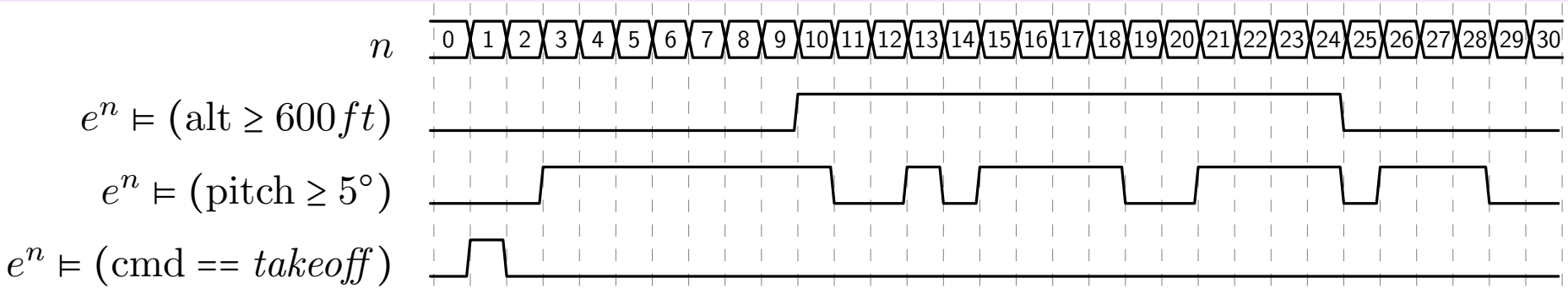
0	(false,0)	8	(true,3)
1	(false,1)	9	(true,4)
2	(false,2)	10	(true,5)
3	(␣, ␣)	11	(false,11) Resynchronized!
4	(␣, ␣)	12	(false,12)
5	(␣, ␣)	13	(␣, ␣)
6	(␣, ␣)	14	(false,14) Resynchronized!
7	(␣, ␣)	15	(␣, ␣)

Synchronous Observers (aka time-triggered)

- update continuously
- 3-valued output: {true; false; maybe}
- small hardware footprints
  ($\leq$ 11 two-input gates/operator)

Synchronous observers update at every tick of the system clock
. . . enabling probabilistic system diagnosis!

Synchronous Observers Example

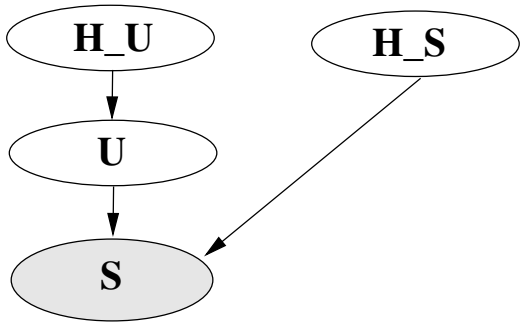


$$\text{ALWAYS}_{[5]}((\text{alt} \geq 600ft) \wedge (\text{pitch} \geq 5°))$$

0	(false,0)	8	(false,8)
1	(false,1)	9	(false,9)
2	(false,2)	10	(maybe,10)
3	(false,3)	11	(false,11)
4	(false,4)	12	(false,12)
5	(false,5)	13	(maybe,13)
6	(false,6)	14	(false,14)
7	(false,7)	15	(maybe,15)

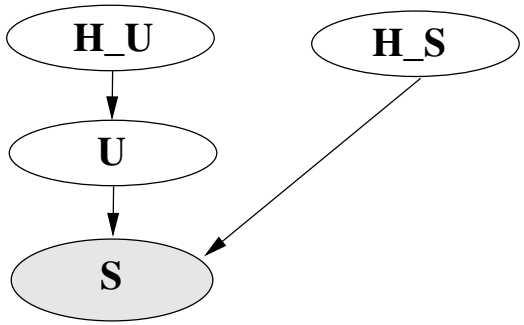
Bayesian Reasoning

- Our Bayesian Networks (BNs) contain
  - (observable) sensor nodes S
  - (unobservable) status nodes U
  - health nodes H_S, H_U
- Discrete sensor values & outputs of LTL/MTL formulas → S nodes
- *Posteriors* of the health nodes H_U, H_S reflect the *most likely* health status of the component



Bayesian Reasoning

- Our Bayesian Networks (BNs) contain
  - (observable) sensor nodes S
  - (unobservable) status nodes U
  - health nodes H_S, H_U



- Discrete sensor values & outputs of LTL/MTL formulas → S nodes

- *Posteriors* of the health nodes H_U, H_S reflect the *most likely* health status of the component

In our framework we do not use Dynamic BNs as temporal aspects are handled by the temporal observers.

R2U2: Realizable, Responsive, Unobtrusive[14]

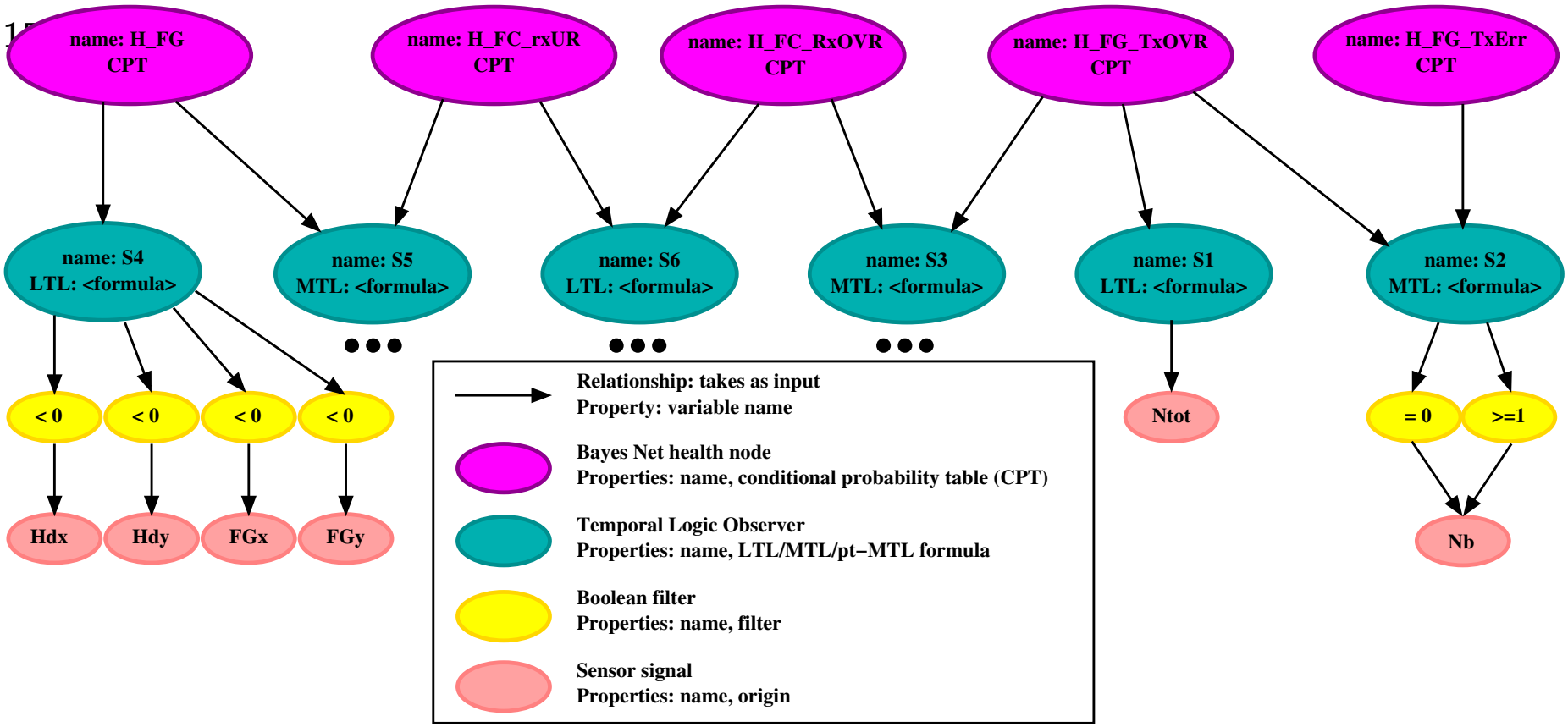
R2U2 specification format:

1. **Signal Processing**: Preparation of sensor readings
   - **Filtering**: processing of incoming data
   - **Discretization**: generation of Boolean outputs
2. **Temporal Logic (TL) Observers**: Efficient temporal reasoning
   1. **Asynchronous**: output $\langle t, \{0, 1\} \rangle$
   2. **Synchronous**: output $\langle t, \{0, 1, ?\} \rangle$

   - **Logics**: MTL, pt-MTL, Mission-time LTL
   - **Variables**: Booleans (from system bus), sensor filter outputs
3. **Bayes Nets**: Efficient decision making
   - **Variables**: outputs of TL observers, sensor filters, Booleans
   - **Output**: most-likely status + probability

[14] Kristin Yvonne Rozier, and Johann Schumann. "R2U2: Tool Overview." In International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES), held in conjunction with the 17th International Conference on Runtime Verification (RV), Kalpa Publications, Seattle, Washington, USA, September 13-16, 2017.

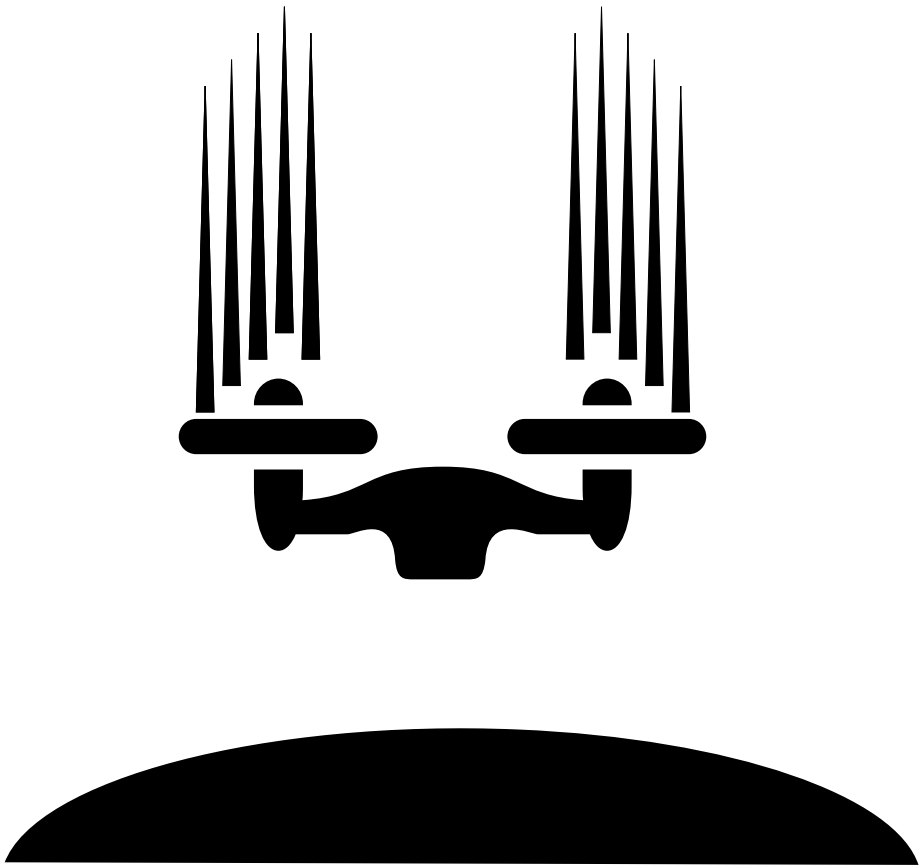
R2U2 Observation Tree (Specification)



[15] Kristin Yvonne Rozier, and Johann Schumann. "R2U2: Tool Overview." In *International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES)*, held in conjunction with the 17th International Conference on Runtime Verification (RV 2017), Springer-Verlag, Seattle, Washington, USA, September 13–16, 2017.

Runtime Functional Specification Patterns[16]

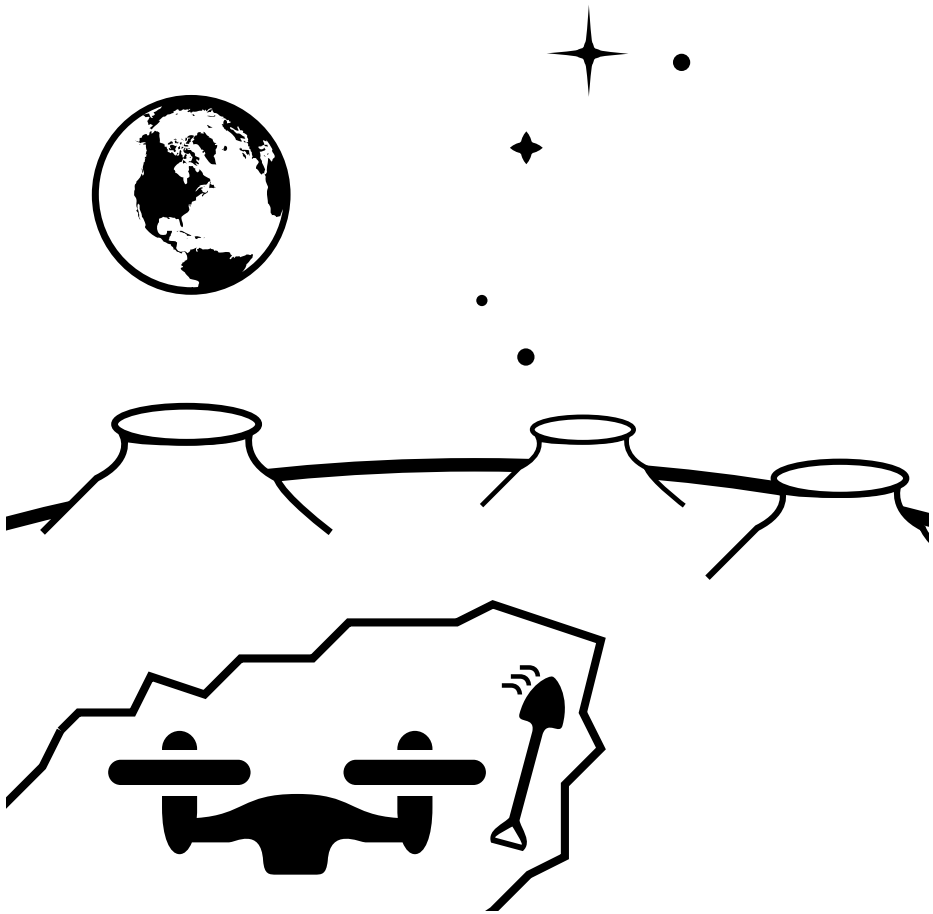
- **Rates**
- **Ranges**
- **Relationships**
- **Control Sequences**
- **Consistency Checks**



[16] K.Y.Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy." VSTTE, 2016.

Runtime Functional Specification Patterns[16]

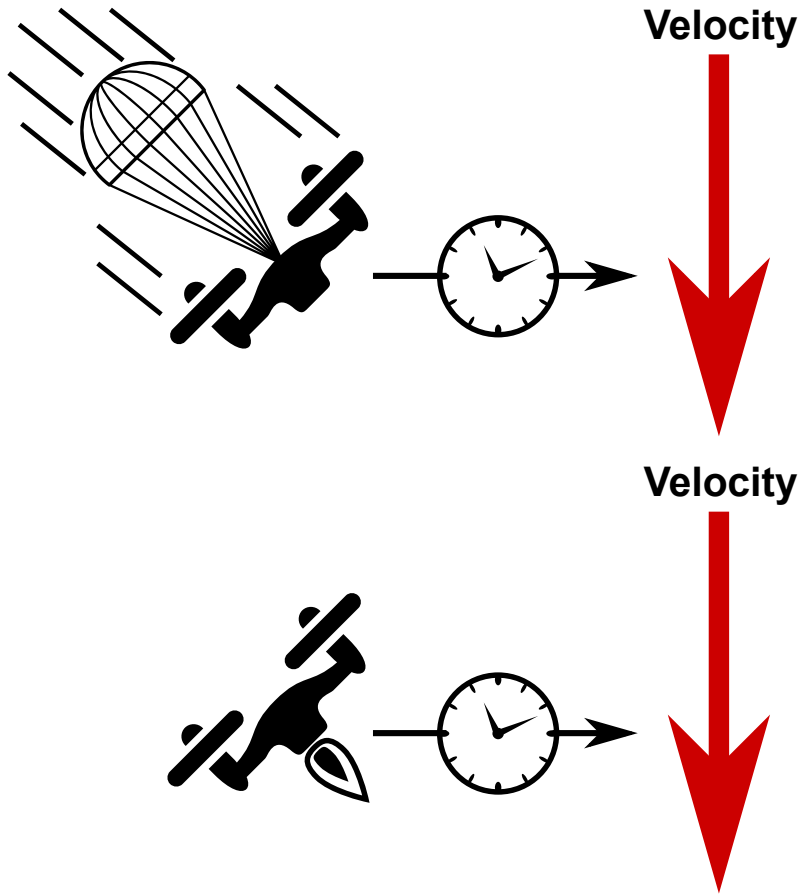
- Rates
- Ranges
- Relationships
- Control Sequences
- Consistency Checks



[16] K.Y.Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy." VSTTE, 2016.

Runtime Functional Specification Patterns[16]

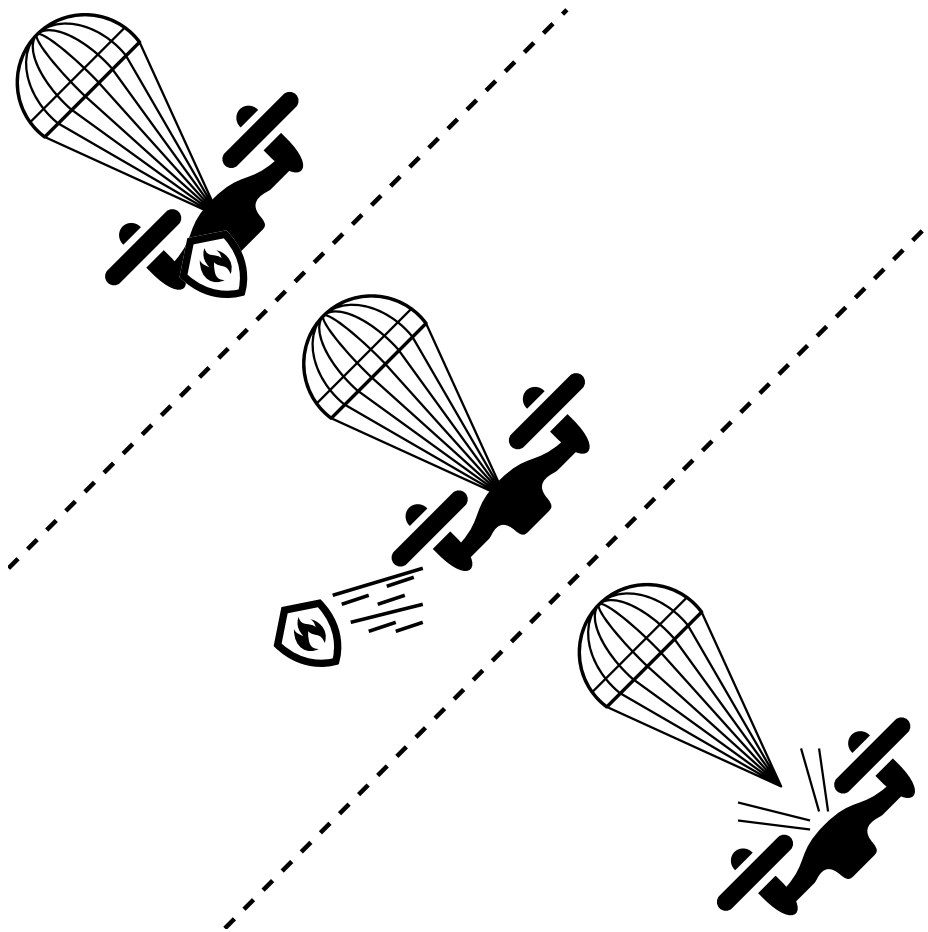
- Rates
- Ranges
- Relationships
- Control Sequences
- Consistency Checks



[16] K.Y.Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy." VSTTE, 2016.

Runtime Functional Specification Patterns[16]

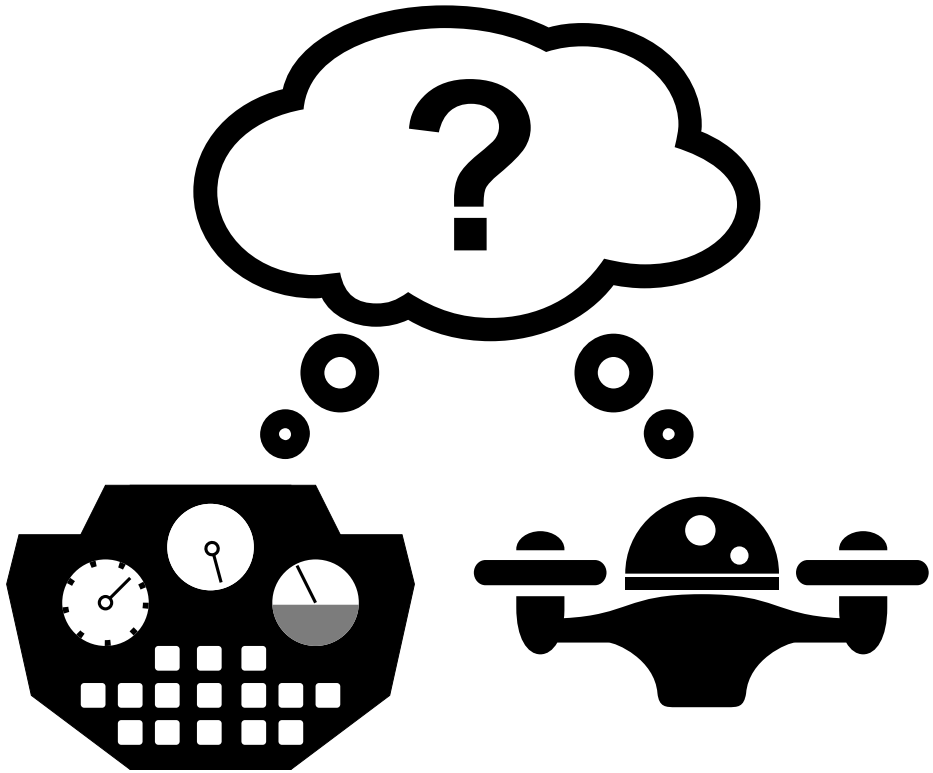
- **Rates**
- **Ranges**
- **Relationships**
- **Control Sequences**
- **Consistency Checks**



[16] K.Y.Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy." VSTTE, 2016.

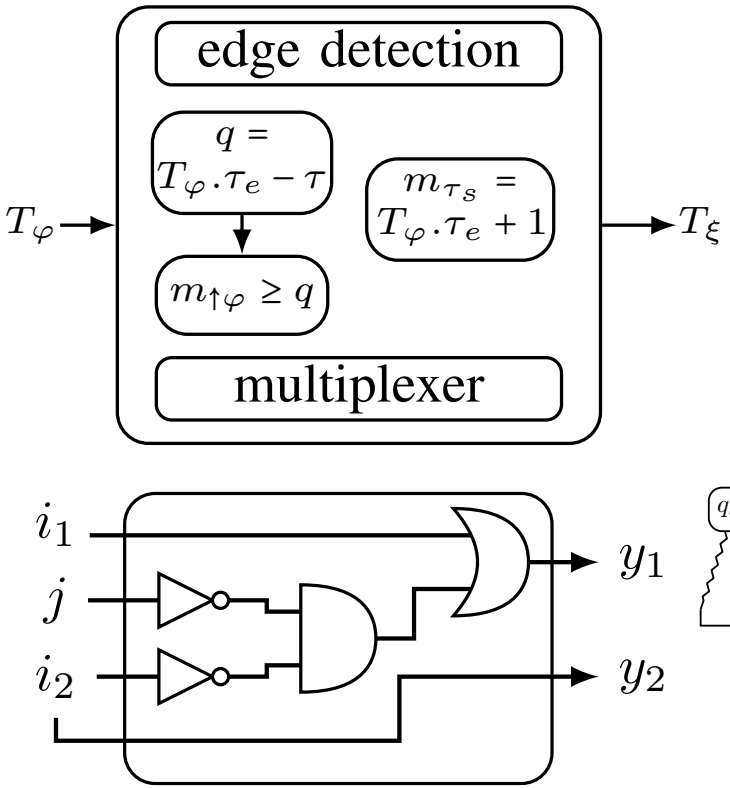
Runtime Functional Specification Patterns[16]

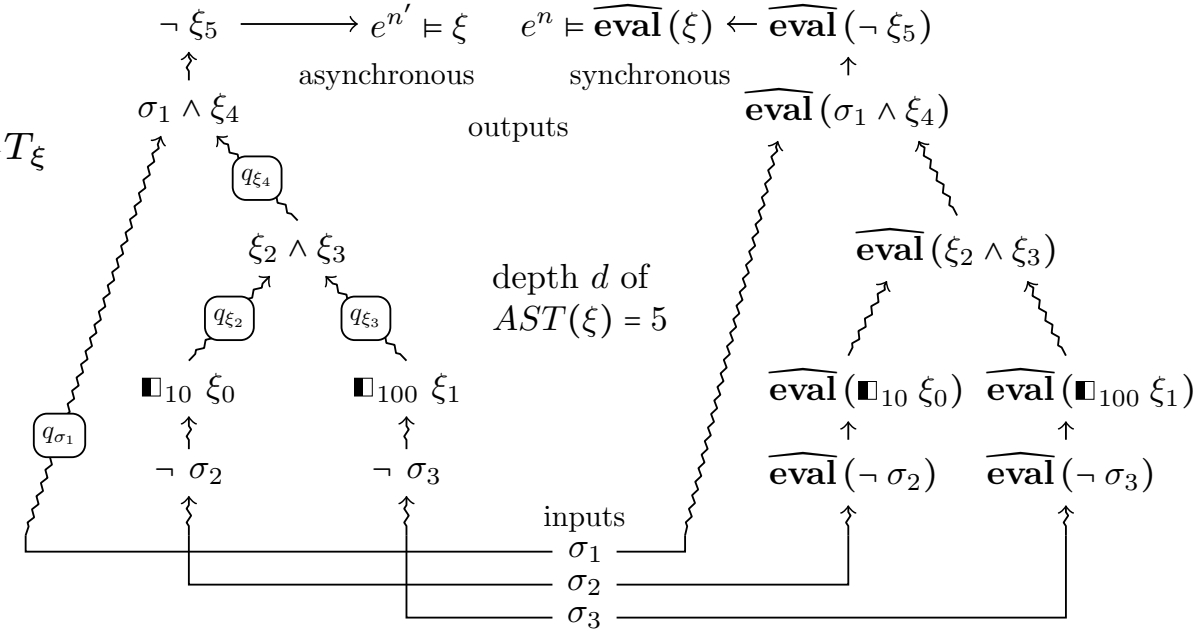
- Rates
- Ranges
- Relationships
- Control Sequences
- Consistency Checks



[16] K.Y.Rozier. "Specification: The Biggest Bottleneck in Formal Methods and Autonomy." VSTTE, 2016.

FPGA Implementation of Temporal Observers[17]

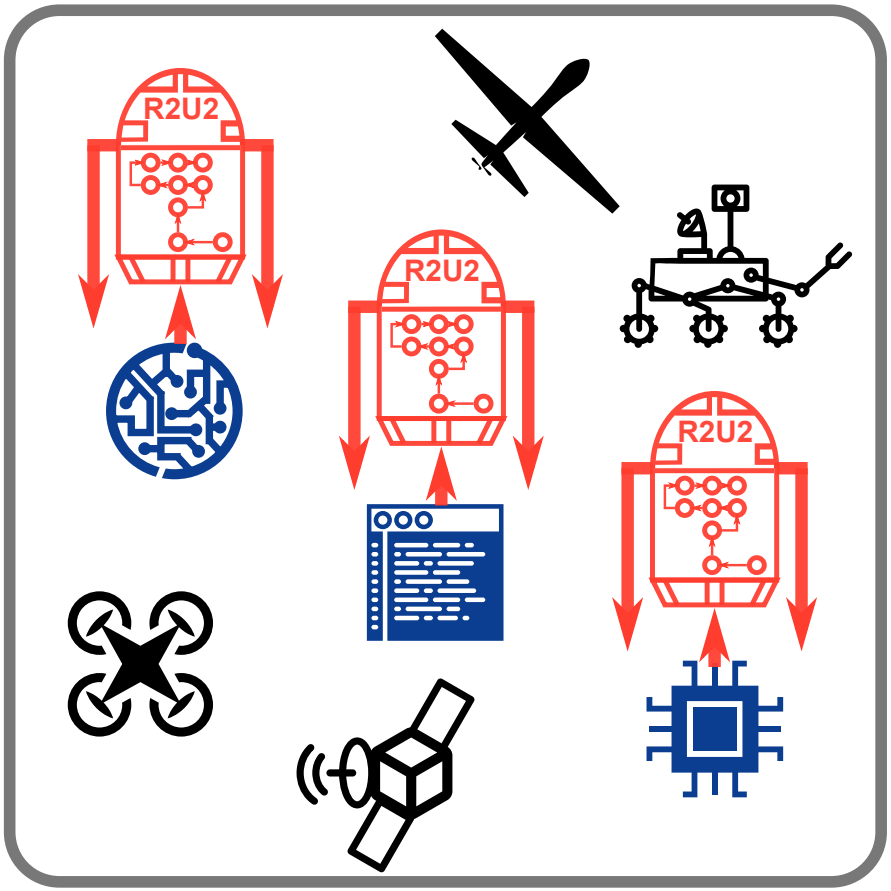




- asynchronous observers: substantial hardware complexity
- synchronous observers: small HW footprint

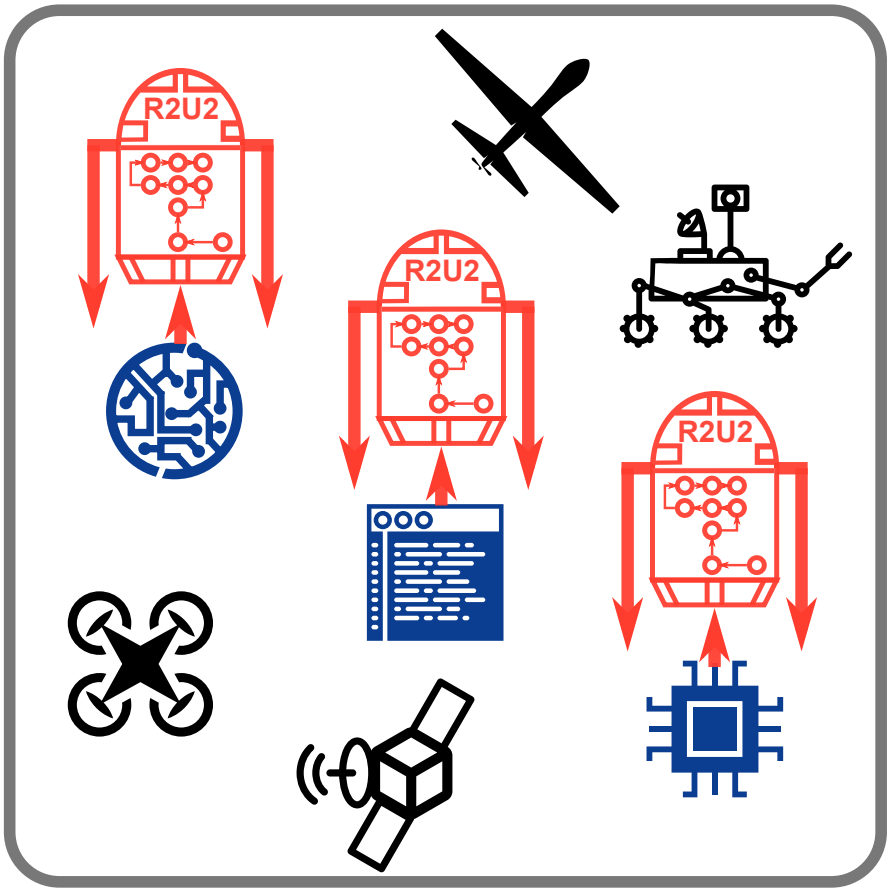
[17] Thomas Reinbacher, Kristin Y. Rozier, and Johann Schumann. "Temporal-Logic Based Runtime Observer Pairs for System Health Management of Real-Time Systems." In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 8413 of Lecture Notes in Computer Science (LNCS), pages 357–372, Springer-Verlag, April, 2014.

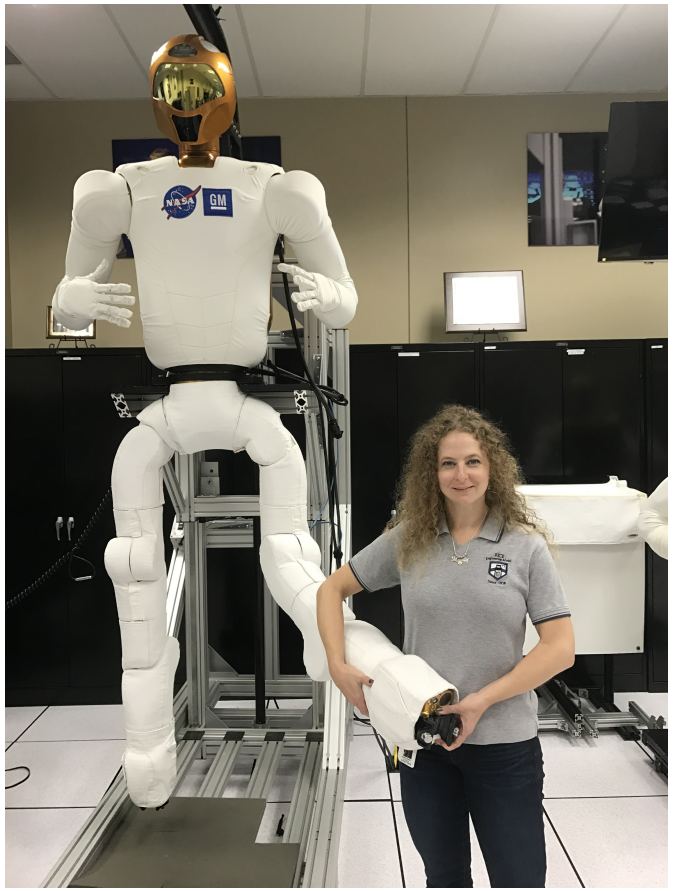
Multi-Platform, Multi-Architecture Runtime Verification of Autonomous Space Systems[18]



[18] NASA ECF Award

Multi-Platform, Multi-Architecture Runtime Verification of Autonomous Space Systems[18]

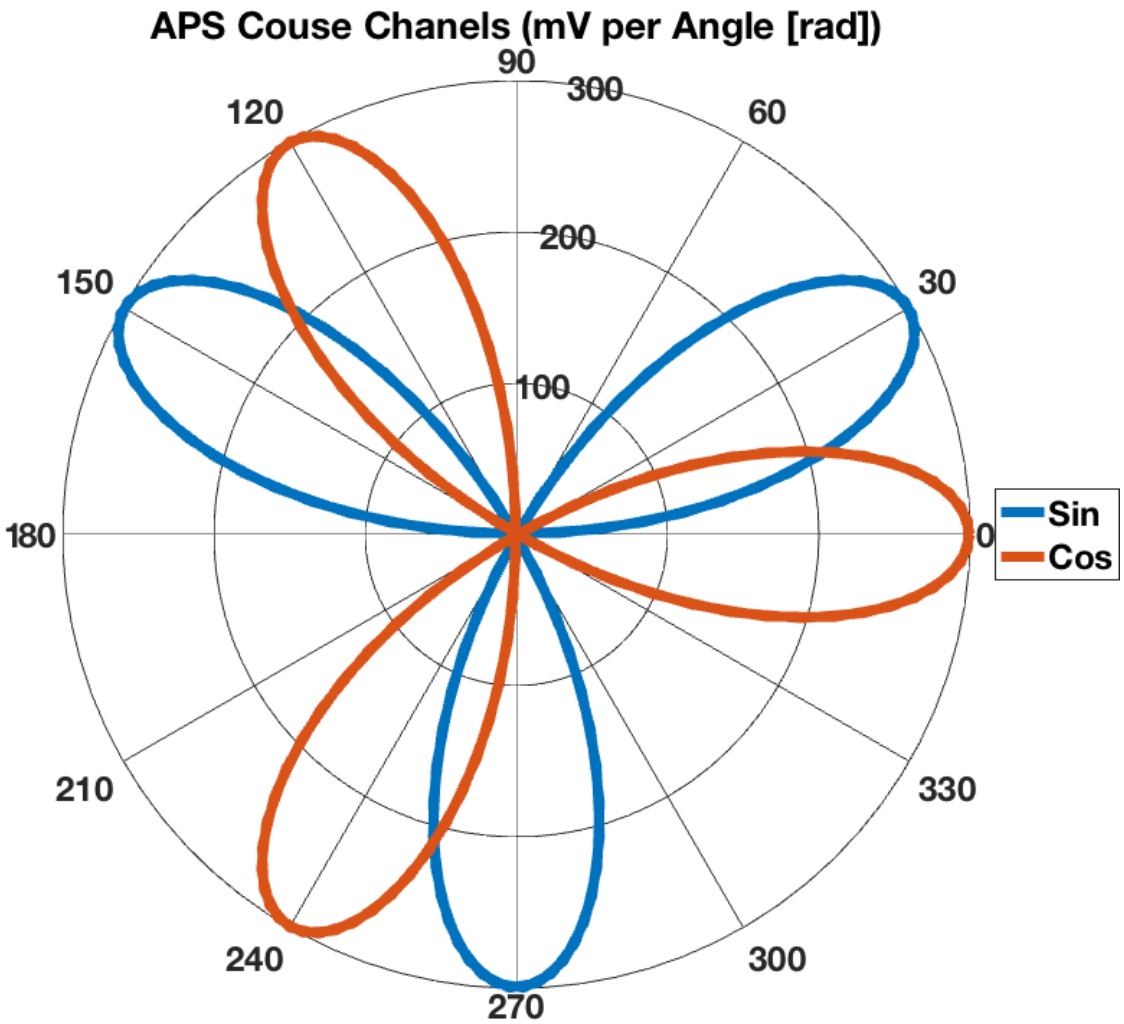




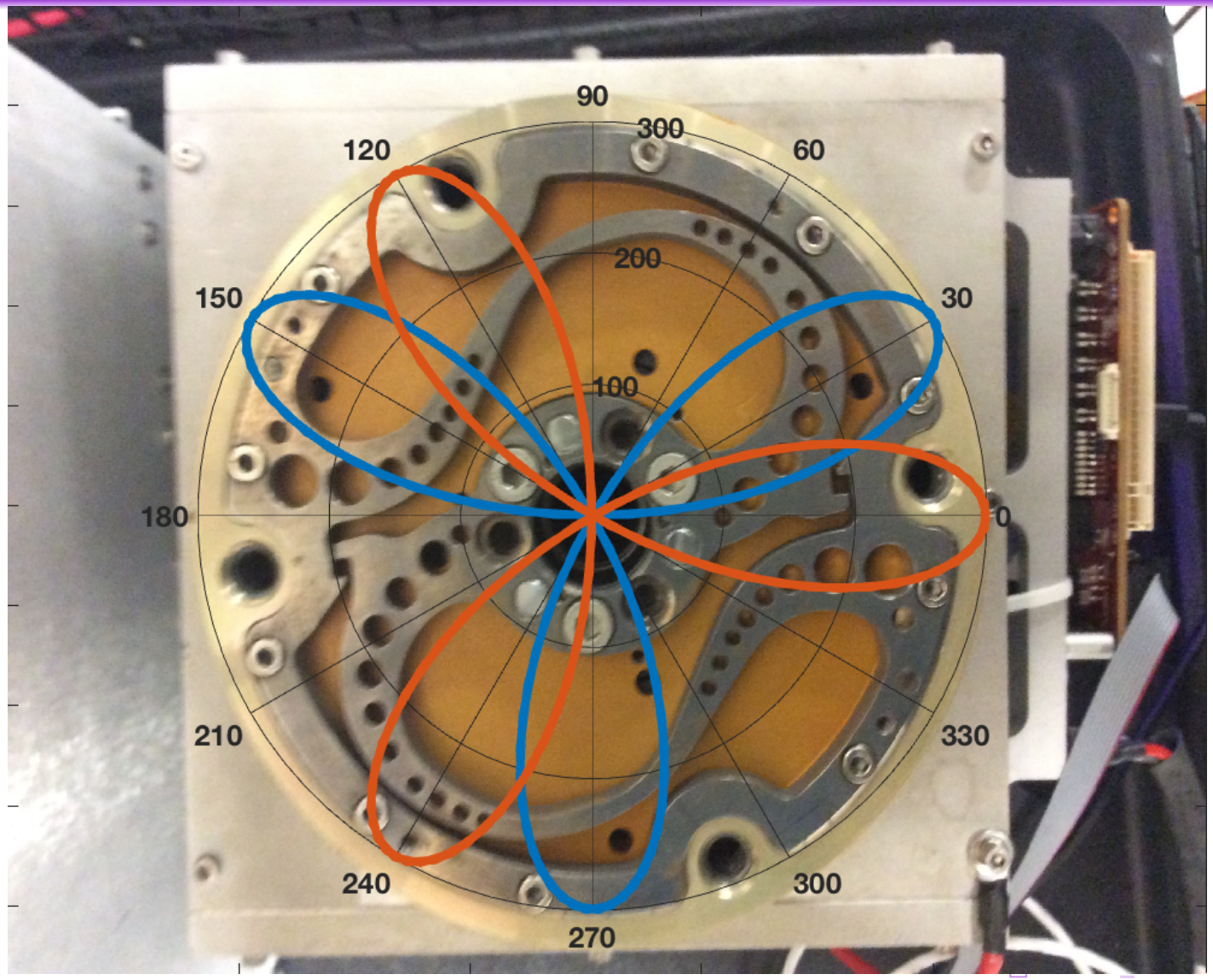
[18] NASA ECF Award

Robonaut2

Robonaut2's Knee



Robonaut2's Knee



Goals to Work Toward

Fault description: (1) identify when a "switch" happens from 1 of 3 positions (as it is at a discrete point during operation), and (2) to identify on the joint level which APS is at fault.
(1) is indicated by φ_1: do APS1 and APS2 disagree
(2) is indicated by the other two MLTL specs: φ_2, φ_3
If φ_1 is triggered but not φ_2 or φ_3 then we have a different fault; trigger standard error handling

Goal 1: detect this fault 100% of the time with no false positives

Goal 2: disambiguate between 3 actions:

1. Reinitialize assuming APS1 is bad
2. Reinitialize assuming APS2 is bad
3. No action: either there is no fault or a different fault has occurred

Goal 3: there is a precursor to this error whose cause is not known?

MLTL Specifications

Do APS1 and APS2 disagree by a large margin (2 radian threshold): indicates that there is a fault

$$THRESHOLD = (2.094 \pm 0.03 rad)$$

2.094 is the 120 separation; 0.03 is the range of the fine position sensing in APS

$$V_{threshold} = |r2.left\_leg.joint0.APS1 - r2.left\_leg.joint0.APS2| > (2.064)$$

$$\varphi_1 = G_{[0,3]}(V_{threshold})$$

Assumption: all faults occur in known transition modes so we can test the monitor with generated error traces for those scenarios

MLTL Specifications

Encoder drift fault occurs and encoder position agrees with APS2 (indicates fault occurred and APS1 is wrong)

$$AGREE_{Enc-APS2} = |r2.left\_leg.joint0.APS2 - r2.left\_leg.joint0.EncPos| < 0.01rad$$

Assumption: this can be refined to represent encoder drift over time but this should be a good indication of agreement in general

$$\varphi_2 = [r2.left\_leg.joint0.FaultEncPos \wedge G_{[0,3]}(AGREE_{Enc-APS2})] \rightarrow APS1_{WRONG}$$

If there is disagreement but *not* encoder drift fault then assume APS2 is wrong:

$$\varphi_3 = G_{[0,3]}(V_{threshold}) \wedge !r2.left\_leg.joint0.FaultEncPos \rightarrow APS2_{WRONG}$$

Assumption: the two agreeing sensors are correct {EncPos, APS1, APS2}
Assumption: all encoder faults are detected in r2.left_leg.joint0.FaultEncPos

http://temporallogic.org/research/R2U2/R2U2-on-R2_demo.mp4

Lifting Runtime Monitoring

Runtime Monitoring

"R2U2 breaks our taxonomy; it is entirely application driven."
— Giles Reger, 11/13/2018[19]

[19] Falcone, Ylis, Sran Krsti, Giles Reger, and Dmitriy Traytel. "A taxonomy for classifying runtime verification tools." In International Conference on Runtime Verification, pp. 241-262. Springer, Cham, 2018.

Lifting Runtime Monitoring

Temporal Fault Disambiguation

Runtime Monitoring

"R2U2 breaks our taxonomy; it is entirely application driven."
— Giles Reger, 11/13/2018[19]

[19] Falcone, Ylis, Sran Krsti, Giles Reger, and Dmitriy Traytel. "A taxonomy for classifying runtime verification tools." In International Conference on Runtime Verification, pp. 241-262. Springer, Cham, 2018.

Lifting Runtime Monitoring

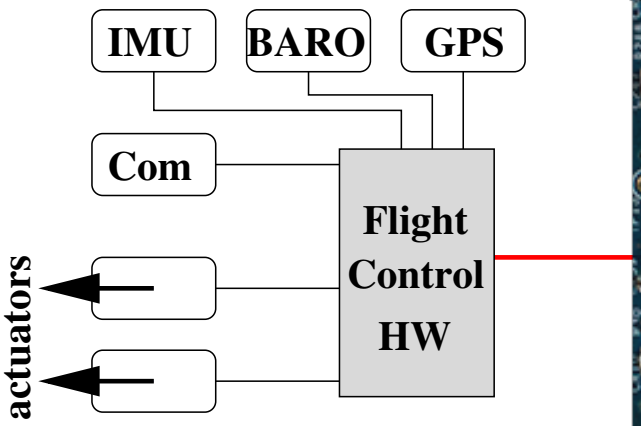
Temporal Fault Disambiguation

Runtime Monitoring

"R2U2 breaks our taxonomy; it is entirely application driven."
— Giles Reger, 11/13/2018[19]

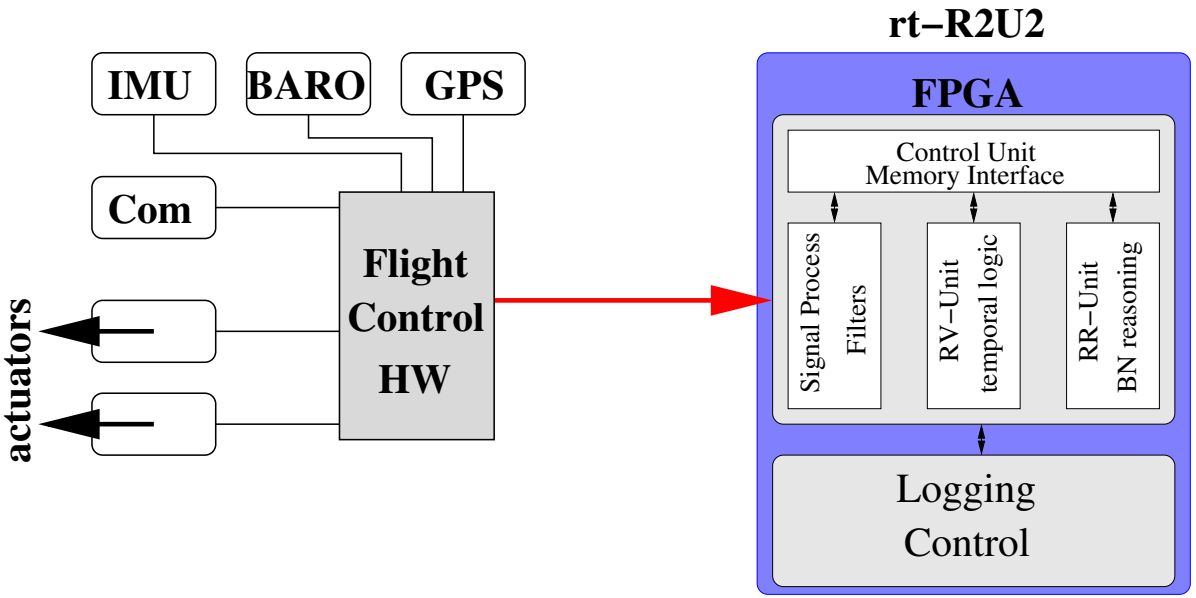
[19] Falcone, Ylis, Sran Krsti, Giles Reger, and Dmitriy Traytel. "A taxonomy for classifying runtime verification tools." In International Conference on Runtime Verification, pp. 241-262. Springer, Cham, 2018.

Hard- and Software Architecture: Resource Estimation



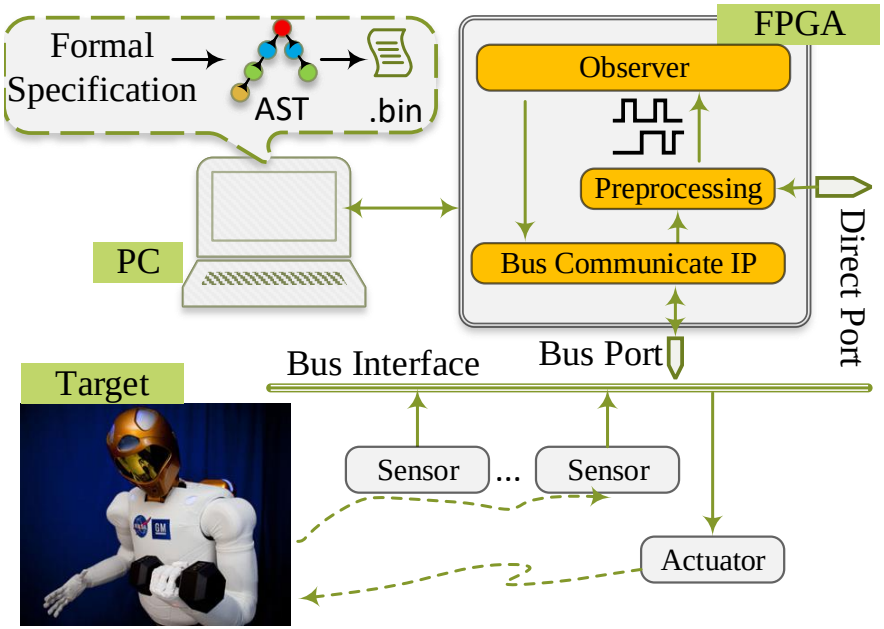
- How do we fit in the resources left over?
- Choose between 3 R2U2 implementations:
  - Hardware: FPGA
  - Software: C emulation of FPGA
  - Software: Object-oriented C++

Hard- and Software Architecture: Resource Estimation

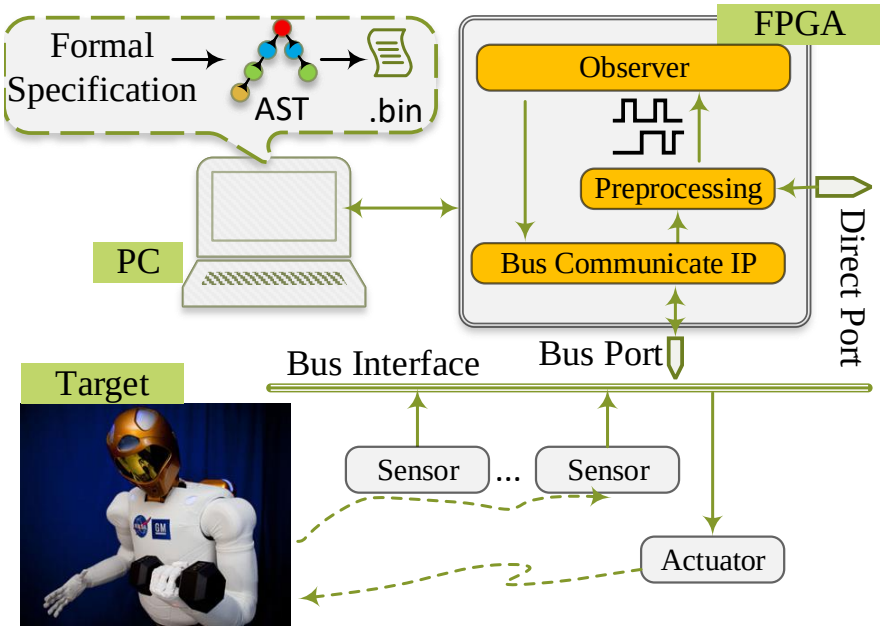


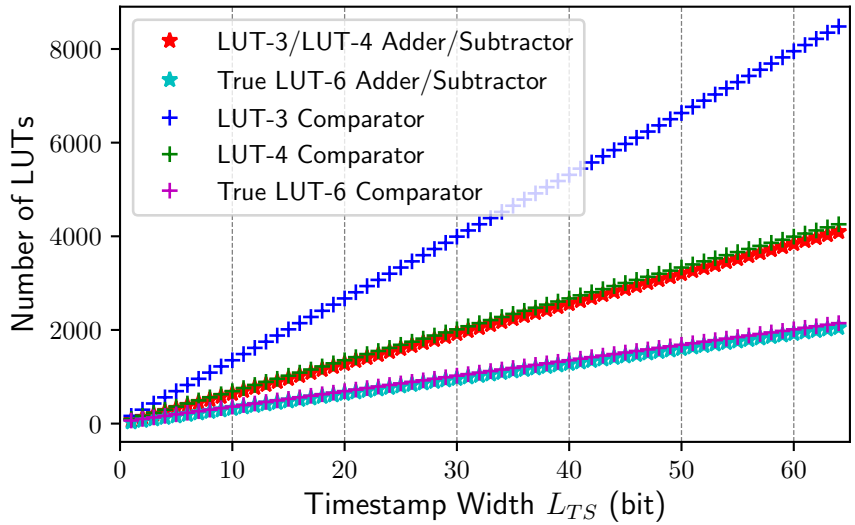
- How do we fit in the resources left over?
- Choose between 3 R2U2 implementations:
  - Hardware: FPGA
  - Software: C emulation of FPGA
  - Software: Object-oriented C++

Resource Estimation and Improved Encoding Algorithms



Resource Estimation and Improved Encoding Algorithms





R2U2: Realizable, Responsive, Unobtrusive[20]

R2U2 specification format:

1. **Signal Processing**: Preparation of sensor readings
   - **Filtering**: processing of incoming data
   - **Discretization**: generation of Boolean outputs
2. **Temporal Logic (TL) Observers**: Efficient temporal reasoning
   1. **Asynchronous**: output $\langle t, \{0, 1\} \rangle$
   2. **Synchronous**: output $\langle t, \{0, 1, ?\} \rangle$

   - **Logics**: MTL, pt-MTL, Mission-time LTL
   - **Variables**: Booleans (from system bus), sensor filter outputs
3. **Bayes Nets**: Efficient decision making
   - **Variables**: outputs of TL observers, sensor filters, Booleans
   - **Output**: most-likely status + probability

[20] Kristin Yvonne Rozier, and Johann Schumann. R2U2: Tool Overview. In International Workshop on Competitions, Usability, Benchmarks, Evaluation, and Standardisation for Runtime Verification Tools (RV-CUBES), held in conjunction with the 17th International Conference on Runtime Verification (RV), Kalpa Publications, Seattle, Washington, USA, September 13-16, 2017.