Formal Methods for Space at NASA Langley

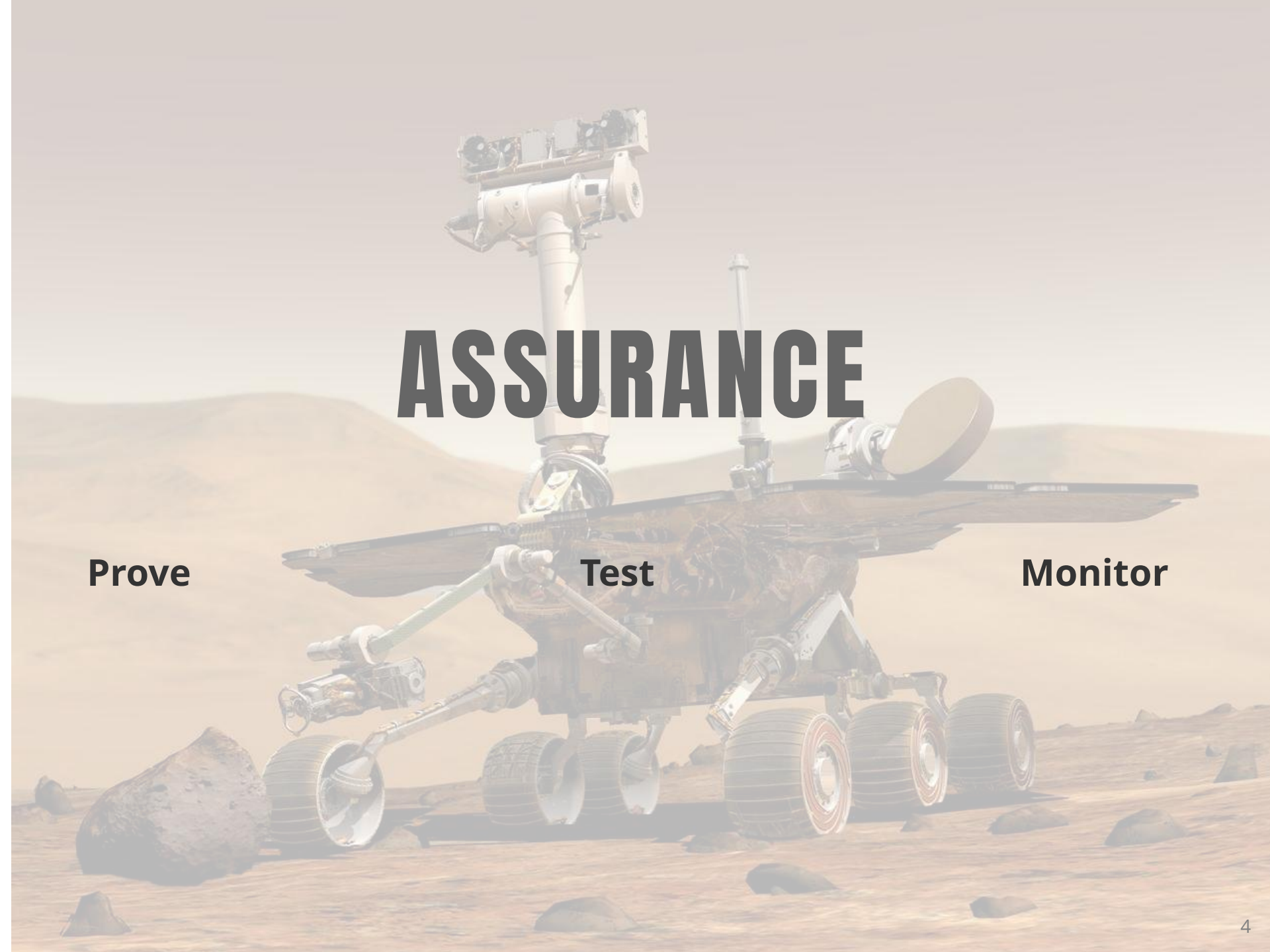
Alwyn E. Goodloe
NASA Langley Research Center
Cesar Munoz
NASA Langley Research Center
Ivan Perez
KBR/NASA Ames Research Center

FM at NASA Langley Research Center

- NASA Langley is a research center
  - Langley is the oldest NASA center historical focus on aeronautics research
- NASA Langley's formal methods (FM) group dates to the 1970s with pioneering work on fault-tolerant avionics
  - Draper SIFT
  - Lamport Byzantine Generals
  - Formally verified clock synchronization protocols
- Two decades of applying FM to verify algorithms enabling safe airspace operations
  - Aircraft separation
  - PVS theorem prover models
    - Sophisticated mathematical library
- Runtime Verification Framework Copilot/Ogma
- Verification of communication protocols
- Plan Execution Interchange Language (PLEXIL)

Why Runtime Verification

- Mission-critical and safety-critical systems often require a high degree of assurance
- Formal verification proves a correctness property holds for every execution of a program
  - Most software is too large , and verification requires very specialized workforce
- Testing demonstrates correctness property holds on specific test cases
- Runtime verification (RV) detects if a correctness property is violated during execution and invokes procedures to steer the system into a safe state
  - A form of dynamic system verification



ASSURANCE

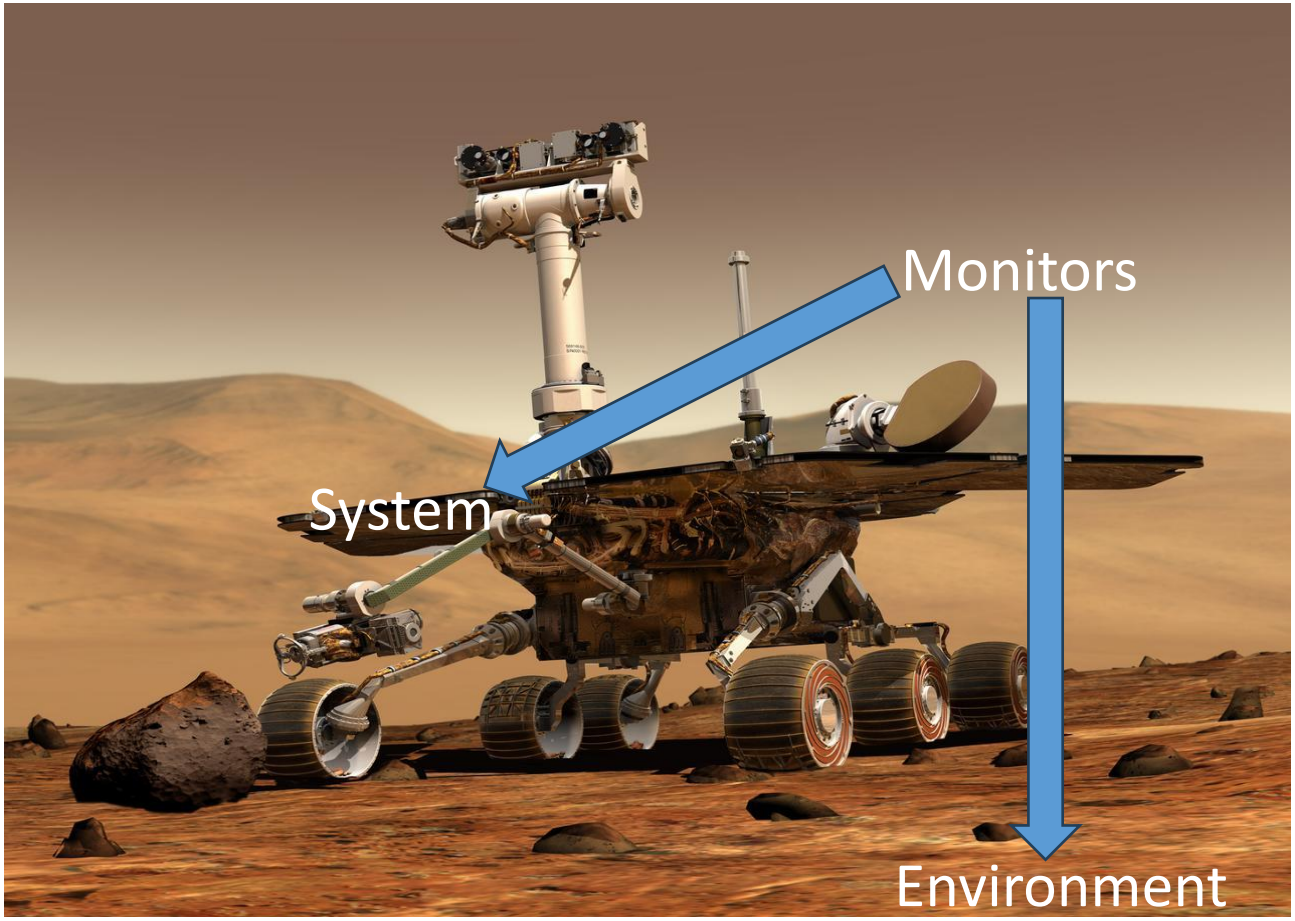
Prove **Test** **Monitor**

Foundations of RV

- Given a specification ϕ of the property we want to check
  - Specification logics: linear temporal logics (LTL), regular expressions, …
- A trace τ of the execution capturing information about the state of a system under observation (SUO)
  - System must be instrumented to capture the trace
- An RV monitor checks for language inclusion $\tau \in \mathfrak{L}(\phi)$
  - Accept all traces admitting ϕ
  - We do this online, but offline analysis is possible

RV frameworks synthesize monitors from specifications

RV in Practice



https://photojournal.jpl.nasa.gov/catalog/PIA04413

What's What

Copilot:

An extensible, high-level language to specify the properties.

Ogma:

A tool to facilitate incorporating monitors into an existing system.

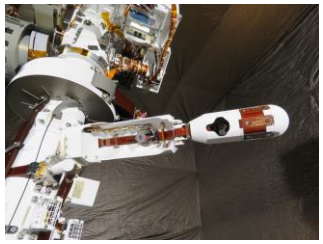
RV Engineer Checklist

- Specify the property to be checked
- Identify the trace to be captured
- Synthesize a monitor that checks the property using an RV framework
- Create handler that steers the system to a safe state when the property is violated
- Install monitor and handler

Copilot

- Copilot is a language and runtime verification framework targeting hard real-time safety-critical systems
  - Collaboration between NASA Langley and NASA Ames
- Stream based specification language
  - Similar to Lustre and LOLA
- Employs sampling rather than extensive code instrumentation
  - Appropriate for monitoring safety of CPS systems
- Copilot specifications are translated into MISRA C99 monitors or to BlueSpec and Verilog for implementation in FPGAs
- Effort started in 2008 as a research program
  - Galois and the National Institute of Aerospace (NIA)
- Copilot and Ogma are NASA software engineering tools
  - Adapted NASA Software Engineering development processes
  - Open source
  - Monitors classified as “Mission Support Software” and flown on NASA flights

Stream Language



Internal Module

		...
12 C	11.75 C	...
{ x: 75 , c: True }	{ x: 88 , c: False }	...
T = 0	T = 1	...

Copilot

Copilot Language

```
spec =
  trigger "heatoff" prop1 [arg medaAirTemperature]


prop1 :: Stream Bool
prop1 = alwaysBeen 0 3 condition

condition :: Stream Bool
condition = temperatureDiff > 25

temperatureDiff :: Stream Int32
temperatureDiff = medaAirTemperature - medaLastTemperature

medaAirTemperature :: Stream Int32
medaAirTemperature = extern "temperature" Nothing

medaLastTemperature :: Stream Int32
medaLastTemperature = [0] ++ medaAirTemperature
```

Copilot Language

```
spec =
  trigger "heatoff" prop1 [arg medaAirTemperature]


prop1 :: Stream Bool
prop1 = alwaysBeen 0 3 condition

condition :: Stream Bool
condition = temperatureDiff > 25

temperatureDiff :: Stream Int32
temperatureDiff = medaAirTemperature - medaLastTemperature

medaAirTemperature :: Stream Int32
medaAirTemperature = extern "temperature" Nothing

medaLastTemperature :: Stream Int32
medaLastTemperature = [0] ++ medaAirTemperature
```

Copilot Language

```
spec =
  trigger "heatoff" prop1 [arg medaAirTemperature]

prop1 :: Stream Bool
prop1 = alwaysBeen 0 3 condition

condition :: Stream Bool
condition = temperatureDiff > 25

temperatureDiff :: Stream Int32
temperatureDiff = medaAirTemperature - medaLastTemperature

medaAirTemperature :: Stream Int32
medaAirTemperature = extern "temperature" Nothing

medaLastTemperature :: Stream Int32
medaLastTemperature = [0] ++ medaAirTemperature
```

Call ‘heatoff’ every time that ‘prop1’ is true

Copilot Language

```
spec =
  trigger "heatoff" prop1 [arg medaAirTemperature]

prop1 :: Stream Bool
prop1 = alwaysBeen 0 3 condition

condition :: Stream Bool
condition = temperatureDiff > 25

temperatureDiff :: Stream Int32
temperatureDiff = medaAirTemperature - medaLastTemperature

medaAirTemperature :: Stream Int32
medaAirTemperature = extern "temperature" Nothing

medaLastTemperature :: Stream Int32
medaLastTemperature = [0] ++ medaAirTemperature
```

True if 'condition' has always been true for the last 4 samples

Copilot Language

```
spec =
  trigger "heatoff" prop1 [arg medaAirTemperature]


prop1 :: Stream Bool
prop1 = alwaysBeen 0 3 condition

condition :: Stream Bool
condition = temperatureDiff > 25

temperatureDiff :: Stream Int32
temperatureDiff = medaAirTemperature - medaLastTemperature

medaAirTemperature :: Stream Int32
medaAirTemperature = extern "temperature" Nothing

medaLastTemperature :: Stream Int32
medaLastTemperature = [0] ++ medaAirTemperature
```

Boolean condition

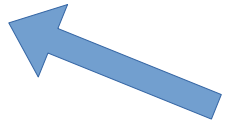
External data

Copilot: Structs

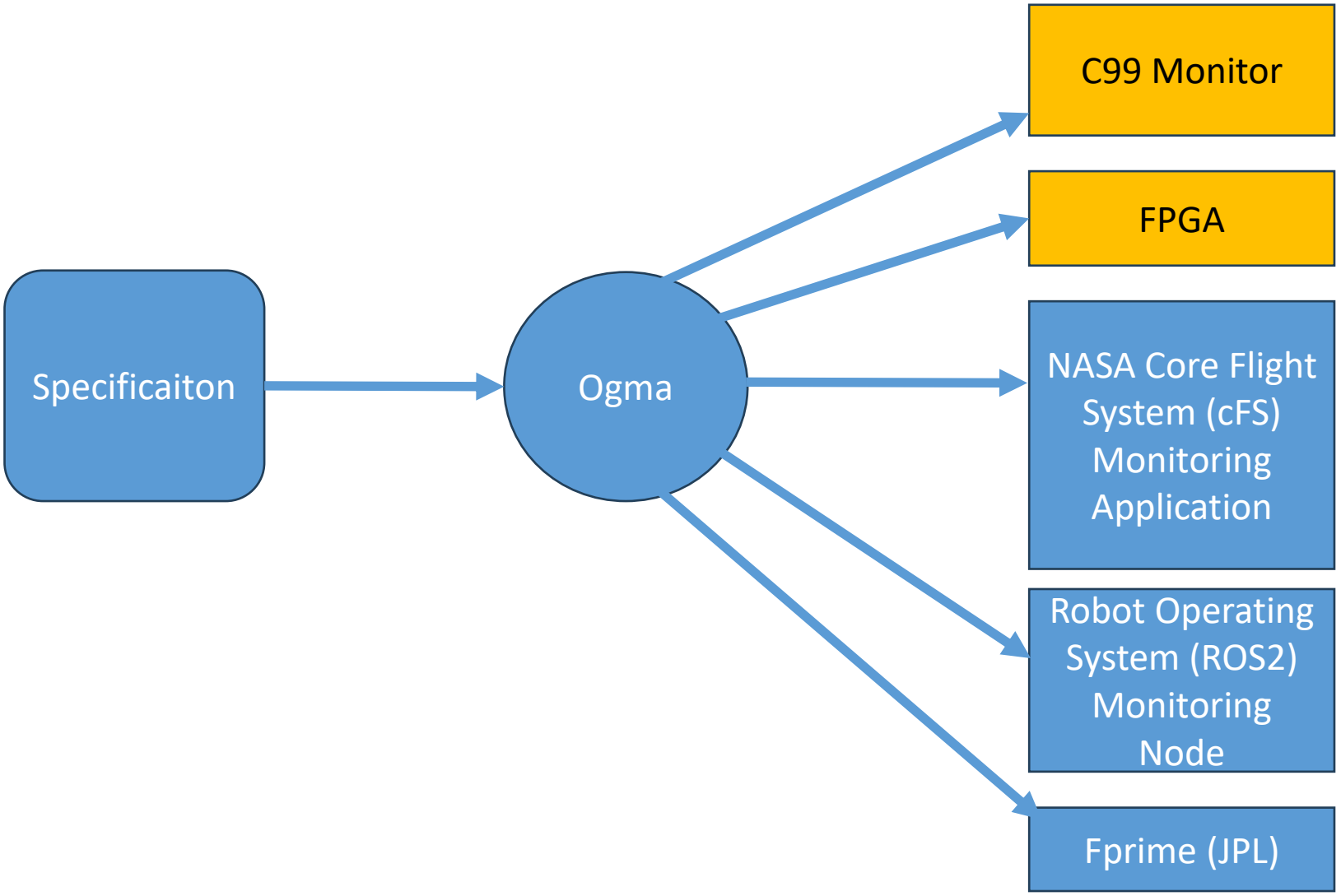
```
data Volts = Volts
  { numVolts :: Field "numVolts" Word16
  , flag     :: Field "flag"     Bool
  }

voltage :: Stream Volts
voltage = extern “voltage” Nothing

prop :: Stream Bool
prop = voltage # numVolts > 200 && voltage # flag
```

Copilot: Arrays

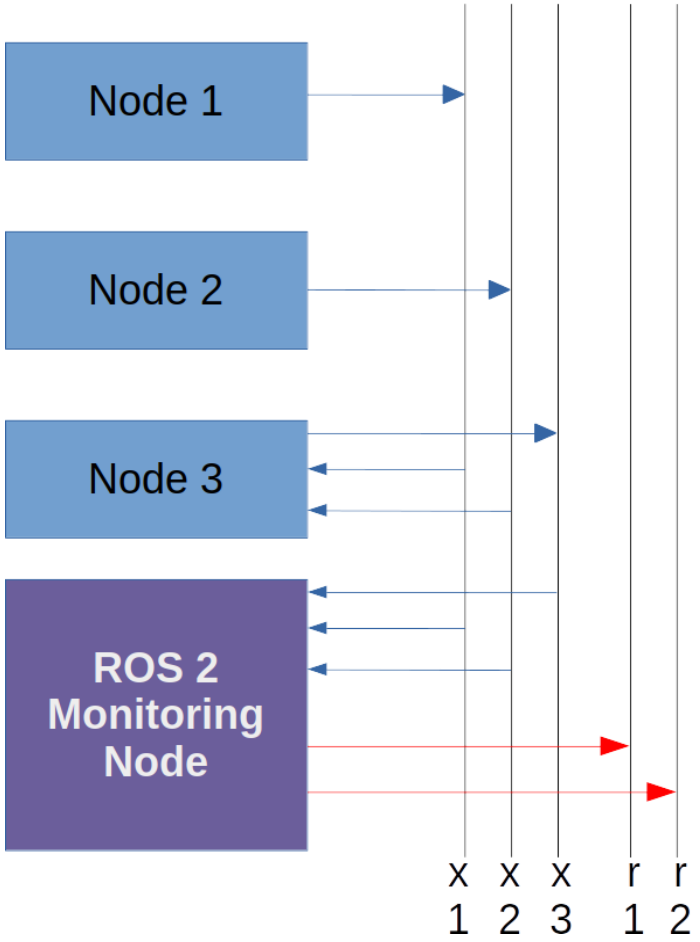
```
data Volts = Volts
  { numVolts :: Field "numVolts" Word16
  , flag     :: Field "flag"     Bool
  }

voltages :: Stream (Array 2 Volts)
voltages = extern “voltage” Nothing

prop :: Stream Bool
prop =
 (voltages!0) # numVolts > (voltages!1) # numVolts
```



Ogma


C99 Monitor
FPGA
Specificaiton
Ogma
NASA Core Flight System (cFS) Monitoring Application
Robot Operating System (ROS2) Monitoring Node
Fprime (JPL)

Monitoring RoS



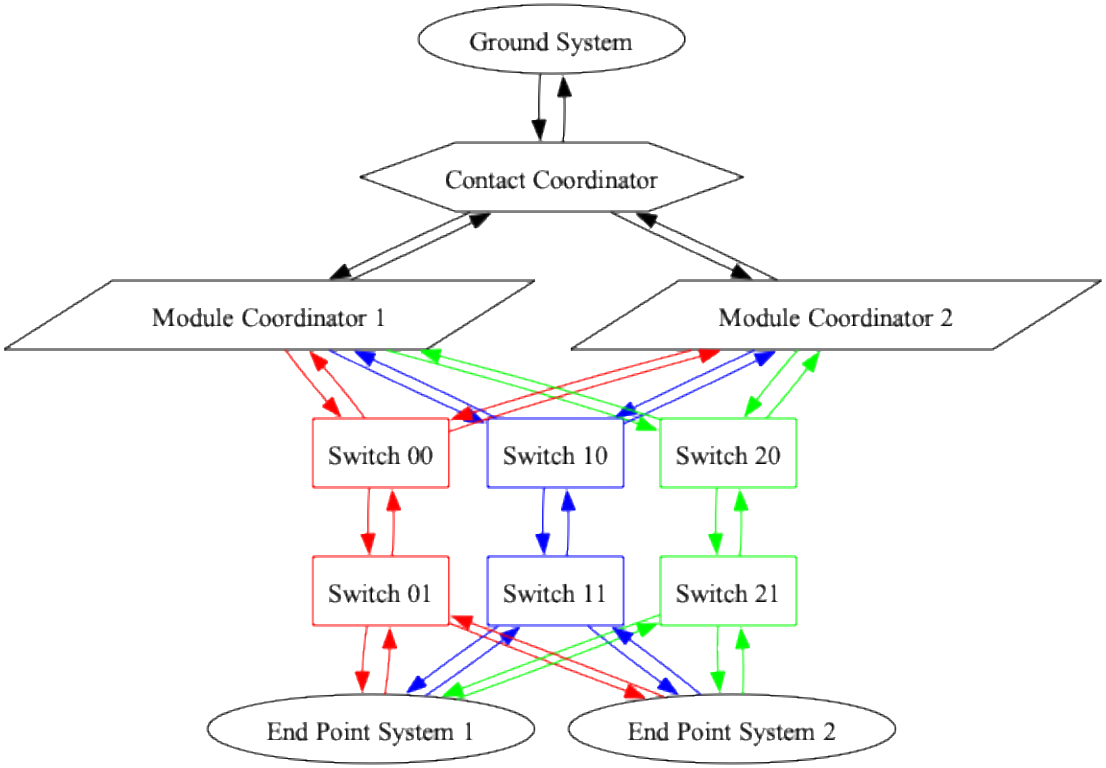
Reconfigurable Networks in Space

- As space missions become more complex and longer duration, the avionics are becoming complex distributed systems
  - Expected to operate without significant downtime or human management
  - Very long durations
- System architects are adopting Ethernet variants for networking
  - Time-Triggered Ethernet (TTE)
  - Avionics Full-Duplex Switched (AFDX) Ethernet
- Design tradeoffs favor determinism and fault tolerance
  - Static network configuration enables predictable behavior
- Fixed number of network elements
- Each node maintains one or more configuration files
- Changes to the network configuration requires files be updated
- Given the long duration of space missions, how best to do the update?

Ways to Update Config Files

- Preplan and store all conceivable configurations
  - Consumes many resources
  - Assumes it is possible to plan for any scenario
- Gound based controllers could manually upload each file
  - Known to be error prone
- Have astronauts manually update files
  - Not practical and very error prone
- Develop a protocol that is robust to faults and failures to reconfigure the network
  - NASA researchers have developed such a protocol
- Such a protocol will need to have undergone extensive analysis
  - Failure can endanger the spacecraft and/or humans onboard!!

Network Communication Structure



Roles of Nodes in System

- Ground System (GS)(Root of Trust) – Starting point of all commands
  - Symmetrically omissive/Fail Silent
- Contact Coordinator (CC) – Direct connection to ground. Orchestrates distribution of messages
  - Byzantine/Fail Arbitrary
- Module Coordinator (MC) – “Regional” coordinator orchestrating distribution of messages to a particular fault containment region (FCR)
  - Only one active per region
  - Byzantine/Fail Arbitrary
- Switch (SW) – Nodes that route messages, but can execute protocol commands
  - Asymmetric Omissive/Fail Arbitrary
- End System Participant (ES) - The end system nodes only accept and respond to the protocol
  - Byzantine/Fail Arbitrary

Protocol Structure

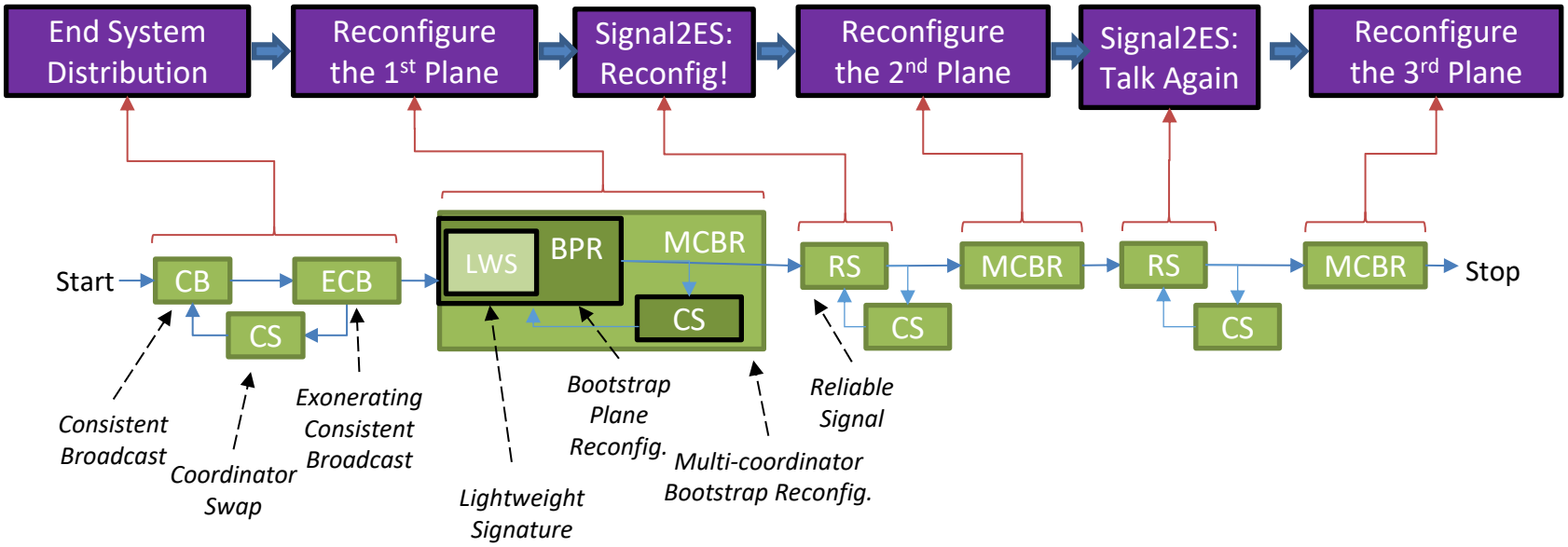
- The protocol is composed from a set of primitives
- Primitives are either cryptographic operations or small protocols
- Primitive protocols follow the same pattern:
  - Ground system sends command to coordinator
  - Coordinator running the protocol will broadcast commands to receiving nodes
  - Receiving nodes receive command and perform an operation and send a reply to coordinator
  - Ground system requests an acknowledgement
  - Coordinator gathers acks from participants and sends reply to ground system
  - Protocol on ground evaluates and acts on the information in the ack it receives

Protocol Primitives

- Coordinator Swap (CS) – Swap out/in nodes designated as coordinators
- Bootstrap Plane Reconfiguration (BPR) – One coordinator directs the reconfiguration of switches in a plane
- Multi-Coordinator Bootstrap (MBPR) – Reconfigure switches using all available coordinators
- Reliable Bootstrap Reconfiguration (RBR) – Disable planes when MBR fails
- Consistent Broadcast (CB) – Configure end systems
- Exonerating Consistent Broadcast (ECB) – Identify faults when configuring end systems
- Reliable Signal (RS) – Send value to nodes and verify that coordinator sent a values all members of a group of participants

Protocol Steps

Flow of Top-Level Operations



Flow of Primitives that Comprise Top-Level Operations

Assuring the Protocol (Work inProgress)

- For the primitives, developers identified several correctness properties:
  - Authenticity
  - Validity
  - Verifiability
- Informal proofs of correctness of individual primitives
- Developers wanted to increase confidence that there was no unintended harmful interactions among the primitives when composed into a larger protocol
- They asked us to help answer this question
- We are reporting on WORK IN PROGRESS

Modeling Process

- Construct an abstract model of the system
  - Model network elements (switches, connections)
  - Model protocol primitives and their composition
    - Abstraction requires tradeoffs
- We are building two models using different tools making different tradeoffs
  - One model naturally synchronous, network layout changed easily, and easily model arbitrary node failures
  - The other model has more fidelity in modeling network elements, but with fewer nodes and more difficult to change configuration
- Properties to be checked
  - Do primitives interfere with each other?
  - Failure modes
  - Currently in early stages

What We Model

- Network elements - Switches, coordinators, end systems, parallel network planes
- Coordinator Swap (CS) – Swap out/in nodes designated as coordinators
- Consistent Broadcast (CB) – Configure end systems
- Exonerating Consistent Broadcast (ECB) – Identify faults when configuring end systems
- Reliable Signal (RS) – Send value to nodes and verify that coordinator sent values to each node

Maude

- Maude is a high-level specification language
  - Developed at SRI and UIUC
  - Algebraic specification language
  - Term rewriting
- Maude is a typed language where the types are called *Sorts*
  - Object oriented
- Equations create equivalent classes and substitute one equal term with another
  - eq $t = t'$
- Rewriting rules transform terms in ways that do not necessarily substitute one term for another
  - Rewriting is a logic of concurrent change
  - rl $t_1\ t_2\ \ldots t_n\ \rightarrow t'_1\ t'_2\ \ldots t'_m$.
  - crl $t_1\ t_2\ \ldots t_n\ \rightarrow t'_1\ t'_2\ \ldots t'_m$ if $e = e'$.

Protocol Models

- Each protocol is modeled as a state machine executing at a node
  - State machines defined for GS, CC, MC, SW, and ES
  - Each state is a rule in the model
- Model abstracts away implementation details
- Protocols are simplified to configure one node in system
- Must limit state explosion

Queue in Maude

```
fmod QUEUE {X :: TRIV} is

  sort NeQueue{X} Queue{X} .
  subsort NeQueue{X} < Queue{X} .
  op empty : -> Queue{X} [ctor] .
  op enqueue : Queue{X} X$Elt -> NeQueue{X} [ctor] .
  op dequeue : NeQueue{X} -> Queue{X} .
  op first : NeQueue{X} -> X$Elt .
  op isEmpty : Queue{X} -> Bool .

  eq dequeue(enqueue(empty,E)) = empty .
  ceq dequeue(enqueue(Q,E)) = enqueue(dequeue(Q),E) if Q =/= empty .
  eq first(enqueue(empty,E)) = E .
  ceq first(enqueue(Q,E)) = first(Q) if Q =/= empty .
  eq isEmpty(empty) = true .
  eq isEmpty(enqueue(Q,E)) = false .
  eq isEmpty(enqueue(empty,E)) = false .
```

Channels

- Channel is an object comprised of a queue and identifiers for the end points

< A : Channel | queue : Q , in : B, out : C >

- All nodes in the network are connected by pairs of unidirectional channels
- Messages are sent and received by putting them into and removing the from channel queues

Packets and Switches

- Packets have source, destination, msg type, and payload
- op < _ _ _ | _ > : Address Address MsgType Payload -> Packet
- Switches move packets
- Keep more than one routing table at each switch
  - Config swap
- Each table maps addresses to channel IDs
- Switch is a map of maps :
- op sw-routingtable :_ : Map{Nat , Map{Address,Oid}} -> Attribute .
- When routing a packet, select the routing table and look at the destination of the packet in an inbound queue and lookup the channel to put that packet in
  - (RT [SelRT]) [pi-dst(first(QI))

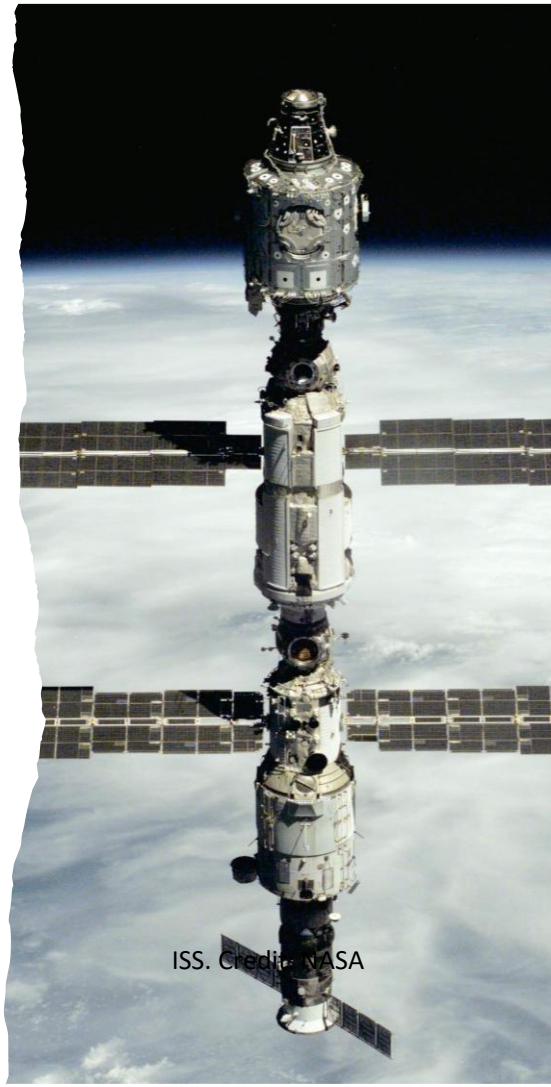
Next Steps

- Explore different classes of faults
- Likely fault scenarios for MC and ES:
  - Fail Omissive – a device fails to send or receive an arbitrary number of packets
  - Fail-Inconsistent – One set of receivers gets correct messages and another gets detectably incorrect
  - Fail Arbitrary - device is free to generate arbitrary packets at arbitrary points in time. Device can fail inconsistently
- Explore statistical model checking

Plan Execution Interchange Language (PLEXIL)

K10 Rover. Credit: NASA/Ames

- A **plan execution language** is a specialized language for specifying control strategies that command and monitor a variety of systems such as spacecrafts, robots, instruments, and habitats.
- **PLEXIL** is a *NASA-developed* plan execution language for representing plans for **automation**, as well a technology for executing these plans on real or simulated systems.



ISS. Credit: NASA

Why PLEXIL

"*Mars missions will see unavoidable communication delays of up to 20 minutes each way, as well as periodic communication blackout of up to two weeks*" (State-of-the-Agency for EIO)

- Autonomous plan execution is **required**.
- ***Verifiable correct planning and plan execution*** is essential for safety, autonomy, adaptability of spacecraft operations on highly uncertain and hazardous environments.

PLEXIL has been used on several NASA projects, e.g., Ocean Worlds Autonomy Testbed for Exploration Research and Simulation (OceanWATERS), Lunar Atmosphere and Dust Environment Explorer (LADEE), Drilling Automation for Mars Exploration (DAME), Deep Space Habitat and Habitat Demonstration Unit (DSH/HDU), and Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS).

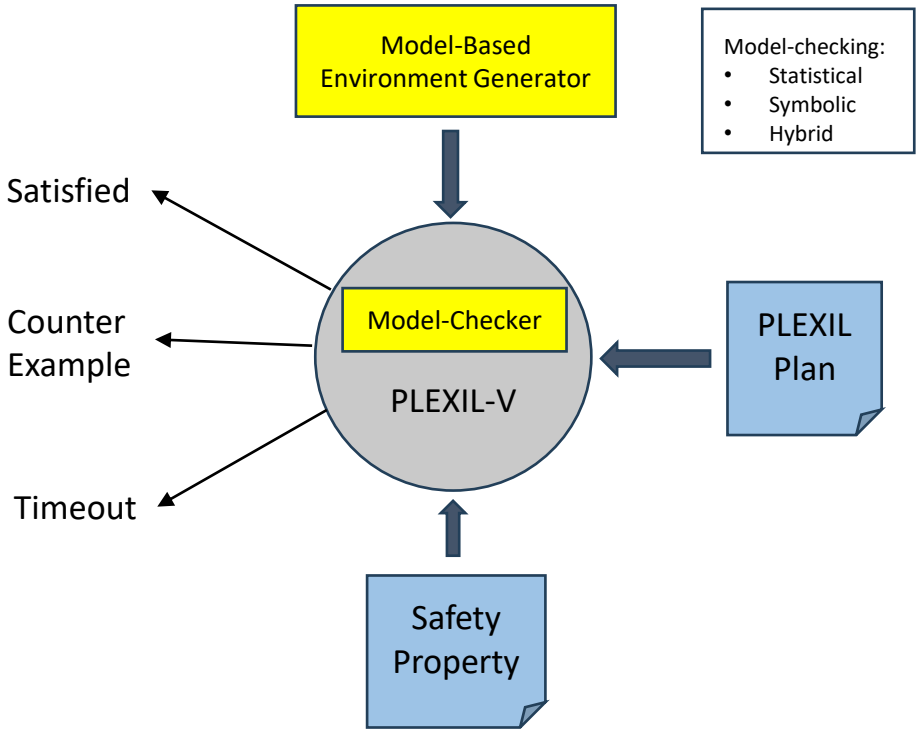
PLEXIL-V

PLEXIL-V is a NASA-LaRC developed formal operational semantics of PLEXIL, which is freely available under NASA Open Source Agreement.

PLEXIL-V provides a reference implementation of the PLEXIL executive.

PLEXIL-V uses theorem-proving and model-checking for the formal verification of plans and plan executions.

Verifiable Correct Plan Execution



FM at NASA JPL

- JPL is a space flight center focusing on robotic exploration missions
  - MARS Rovers
  - Deep space probes
- Small FM team whose members are often embedded with flight software developers
- SPIN model checker developed and applied to high-profile missions such as mars rovers
- Runtime Verification conducted offline
  - RV applied to telemetry data
  - PyContract, TraceContract

FM at NASA Ames

- **Robust Software Engineering** branch
- Inference Kernel for Open Static Analyzers (IKOS)
  - Abstract interpretation of C programs
- Formal Requirements Elicitation Tool (FRET)
  - Transforms structured English into formal specification
  - Generate tests and monitors (via Copilot)
  - Sophisticated user friendly interface
- Safe Deep Neural Networks (SafeDNN)
  - Assurance of neural networks
- CoCoSim: Contract based Compositional verification of Simulink models
- Java PathFinder (JPF)
  - Model checking and symbolic execution

Questions?

Contact Information:

Alwyn Goodloe.
a.goodloe@nasa.gov

Ivan Perez
ivan.perezdominguez@nasa.gov

Cesar Munoz
cesar.a.munoz@nasa.gov