Runtime Verification of Hard Realtime Systems with Copilot: A Tutorial

Ivan Perez

KBR @ NASA Ames Research Center

Alwyn E. Goodloe

NASA Langley Research Center

Frank Dedden

System F Computing

RV Motivation

- Formal verification proves a correctness property for every execution of a program correct
  - Most software is too large and requires very specialized workforce
- Testing demonstrates correctness 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

RV in Practice
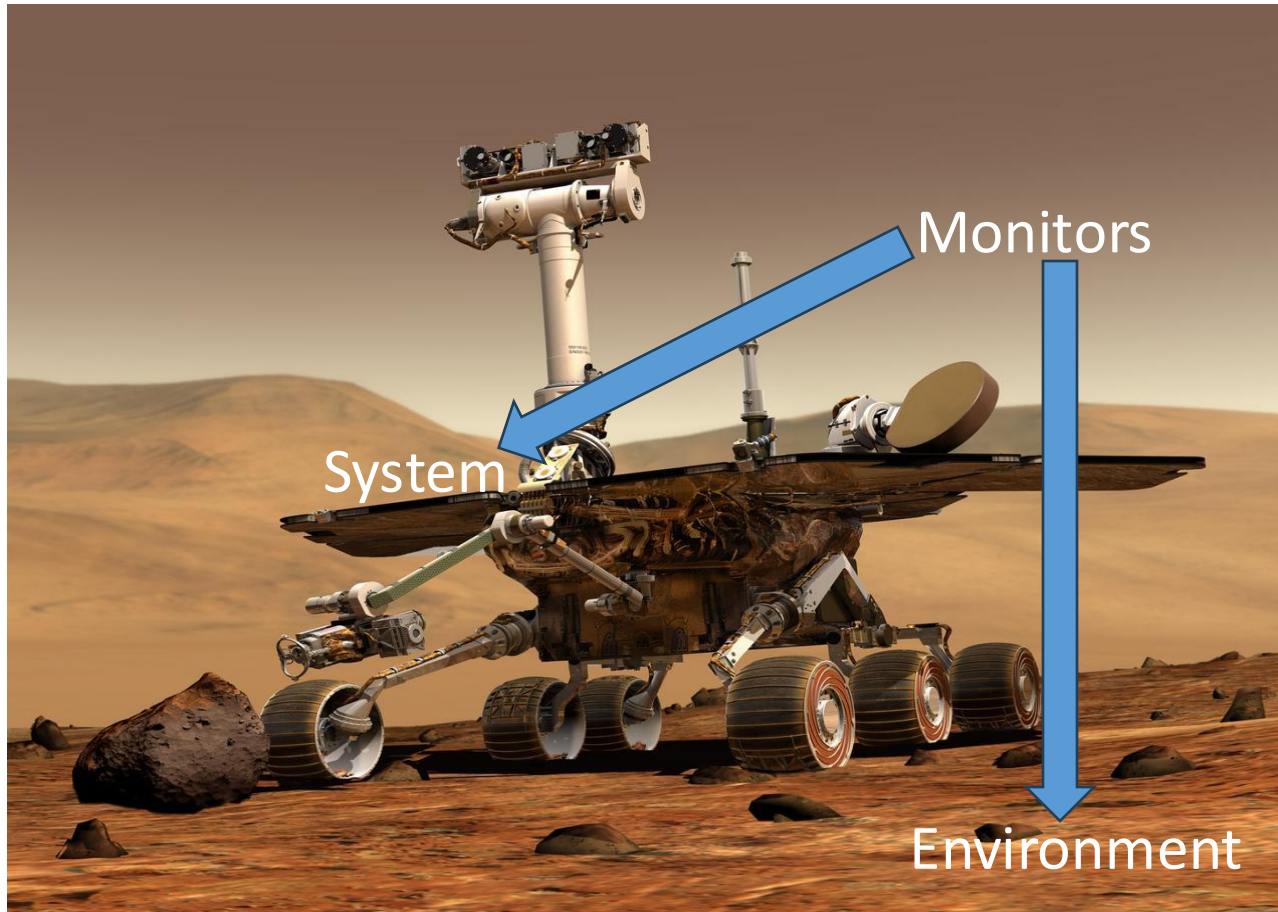
Foundations of RV

- 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

RV frameworks synthesize monitors from specifications

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

Copilot

- Copilot is a language and runtime verification framework targeting hard real-time safety-critical systems
- 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 has evolved into a NASA software engineering tool
  - Adapted NASA Software Engineering development processes
  - Open source
  - Monitors classified as " Mission Support Software" and flown on NASA flights

Copilot Language

- Copilot language implemented as a Haskell Embedded Domain Specific Language (EDSL)
- Users can be productive learning learning Copilot without having to learn Haskell
- Users can write many useful specifications/programs using only a small set of Copilot combinators
- There is an expanding library of predefined combinators to aid in the writing concise specifications
- The Copilot language has been used for general purpose programming of embedded systems
    - Not just for RV

Questions?

Contact Information:

Copilot Language Operators

- Stream language – constants, logic, and arithmetic operations are lifted to stream level

- (++) :: [a] -> Stream a -> Stream a

- xs ++ s   - prepends list xs to stream s

- drop :: Int -> Stream a -> Stream a

- drop n s - skips the first n values in the stream

- Special constructs for input (sampling) and output (triggers)

- Copilot specs must be causal – stream values cannot depend on future values
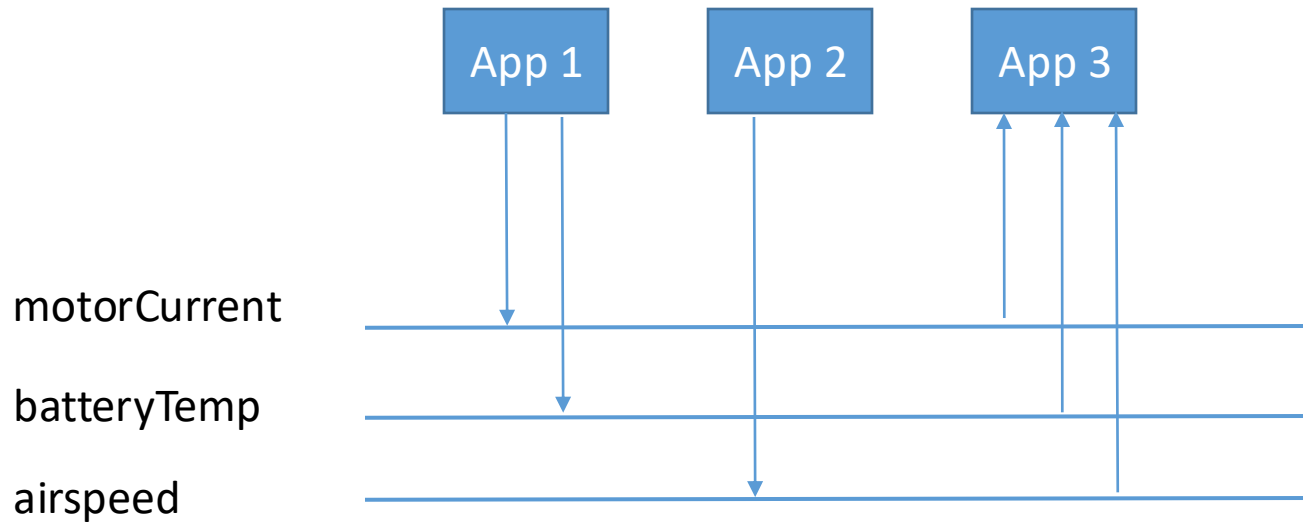
Developing Flight-ready Applications

- NASA Flight Software normally uses NASA's Core Flight System framework or the F' framework.
  - CFS / cFE missions include: Lunar Reconnaissance Orbiter (LRO, 2009), Global Precipitation Measurement (GPM, 2014), VIPER (2024), the Gateway Space Station (development ongoing, launch planned after 2025).
  - F' missions include: Mars Ingenuity Helicopter
  - ROS also used in robotics at NASA.
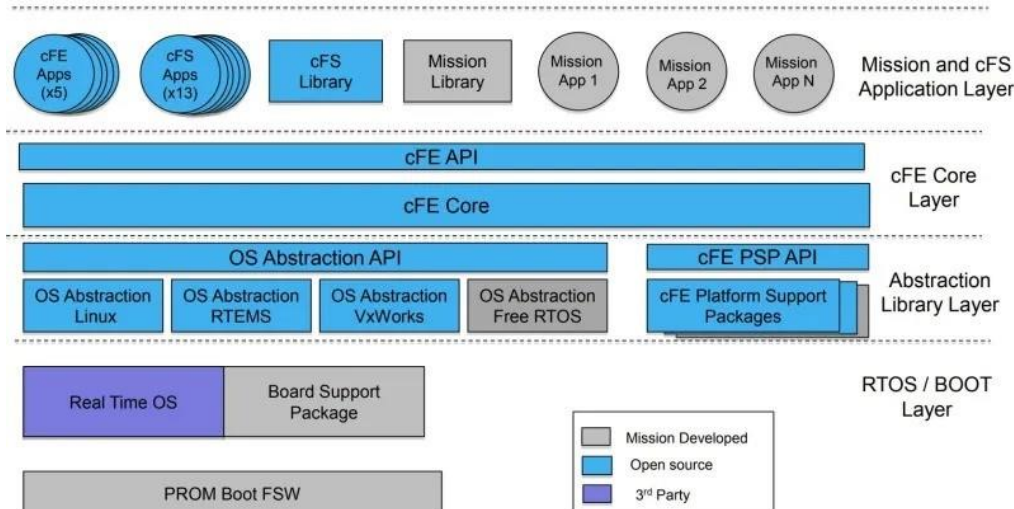
Developing Flight-ready Applications

- These frameworks provide abstractions for component modularization and communication (port-based, software bus).
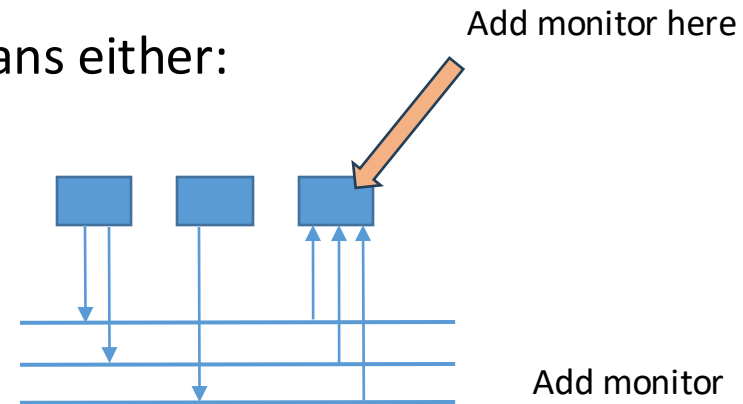
Developing Flight-ready Applications

• Adding monitors to such architecture means either:

   • Instrumenting existing components to inspect and monitor their internal state.

    • Create monitoring applications that treat components as black boxes and listen to data using the mechanisms available to other applications. Violations may trigger recovery mechanism or be published in the SW bus.
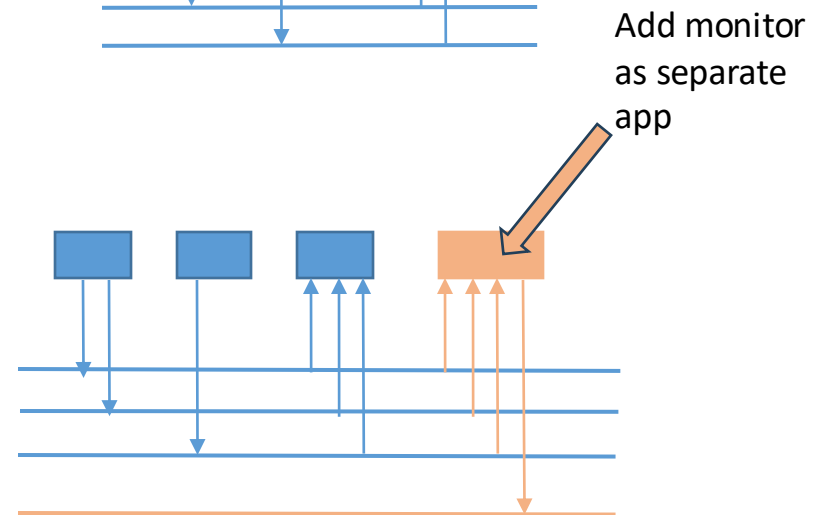
Add monitor here

Add monitor as separate app

Developing Flight-ready Applications

- Errors instrumenting the system for monitoring or adding monitoring applications may lead to failures of the mission as a whole.

- We developed Ogma, a tool that auto-generates complete, ready-to-run monitoring components/applications, using Copilot under the hood to implement the core of the monitors.

- **Ogma reduces the changes of introducing errors in the code that plugs the monitors to the final system.**

High level language → OGMA → Monitoring Application
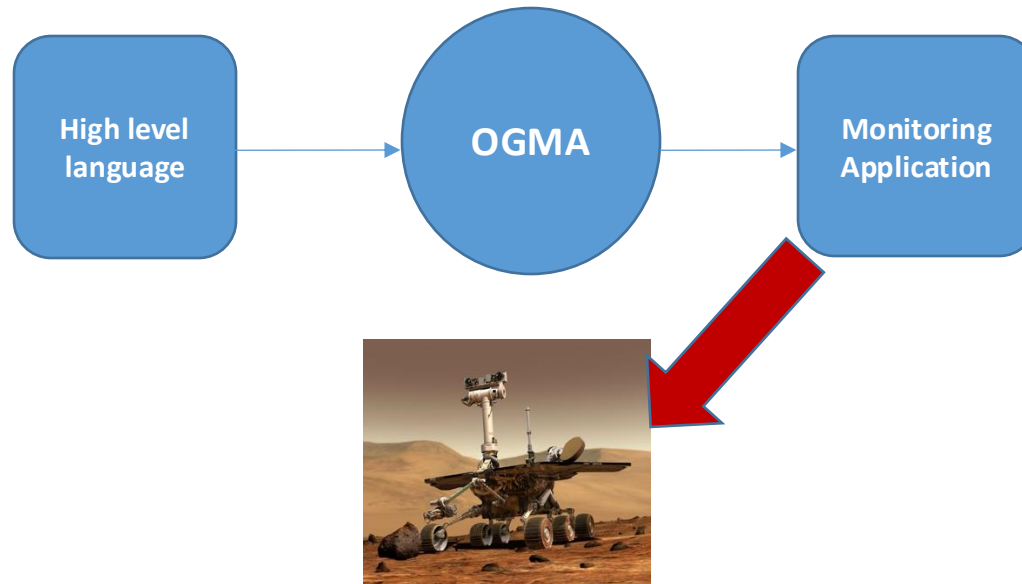
Improving Assurance for Flights

- Engineers within NASA were asking if Copilot could be used to help protect UAS flights against certain unsafe actions
    - Unsafe mode changes
    - Loss of separation
- The monitors would be classified as critical flight software
- Subject to review and approval by NASA flight software safety officials
- The monitors would need to meet criteria for NASA Class C software
- The framework that generated the monitors can be NASA Class D
- Team Copilot decided to accept this challenge while remaining an open-source release

RV You Can Trust Your Life With

- Sound software engineering practices should be applied throughout the software development life-cycle
  - Robust testing with continuous integration
- RV framework should support validating the specification
- RV framework should support analysis of the specification
- RV framework should should generate monitors that run in constant time/constant space and conform to standards such as MISRA
- RV framework should support the verification that the monitors are equivalent to the specification

Software Process Improvement

- NASA Software Engineering Requirements NPR-7150.2C provides necessary guidance
  - Support from center software assurance group
  - Requires organizational buy-in
  - Imposes structure on the development process
- Required documentation
  - Software Plan
  - Coding standard
  - Software Requirements
  - Software Design and Description
  - Work Plan
  - …

Validating the Specification

- Validating that you got the specification right is a critical aspect of assurance

- Copilot framework has integrated support for specification validation

- An interpreter lets users execute the specification without compiling to C and integrating with the SUO

- Theorem proving tools integrated into the framework to determine if the specification satisfies desired properties
  - SMT and model checking

- Support to integrate with simulations of the system being monitored

Testing

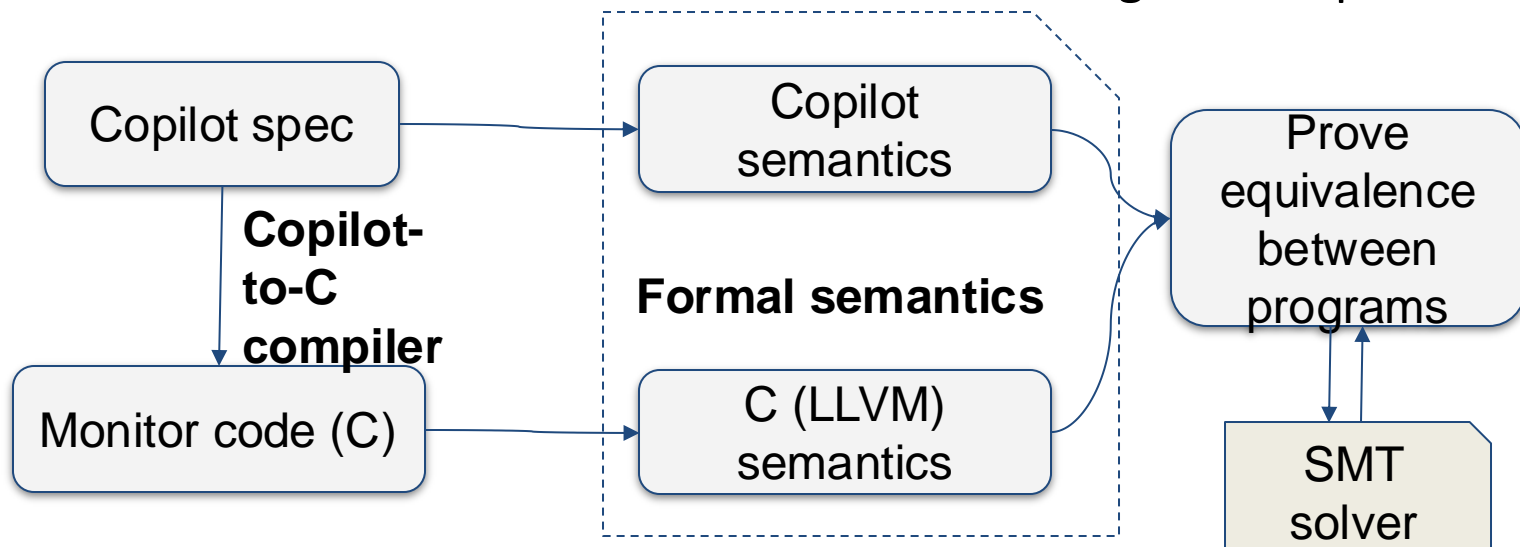- Randomized property based testing using QuickCheck
- Tests generate random Copilot specifications and check that
  - Interpreter follows language semantics
  - C code run
  - C code generated follows language semantics
  - Automated reasoning tools prove specified properties
  - Verify that modules satisfy specified invariants
- Tests run before any code is merged into main line (CI/CD)
- All key libraries have tests

Copilot Monitors + Formal Verification

- CopilotVerifier runs alongside Copilot, proving a proof of correctness for a generated C monitor with respect to its original specification.

- CopilotVerifier uses SMT-backed formal methods to prove that the low-level C code has the same behaviors as the high-level spec.

```
Copilot spec  ──────────────►  Copilot semantics  ──────┐
     │                                                    │
     │  Copilot-                                          ▼
     │  to-C                         Formal semantics    Prove
     │  compiler                                          equivalence
     ▼                                                    between
Monitor code (C) ──────────►  C (LLVM) semantics  ──────►  programs
                                                           ▲ │
                                                           │ ▼
                                                          SMT
                                                          solver
```

19

Certifiably Trustworthy Monitors

- We don't just want a proof of correctness, we want an *explanation* of why it is correct.

- Ongoing work: make Copilot Verifier able to generate evidence that would convince someone auditing a Copilot monitor.

- This should be convincing enough to use in Class-C NASA missions.

Goal 1: The generated C code is traceable to low-level requirements.

↓

Strategy 1: Establish a bisimulation relation between the spec and C code

↓

Goal 2: The initial values of the two programs are the same.

↓

Goal 3: Discharge goal to SMT solver (Z3)

Body of Assurance Evidence

- Following prescribed NASA software development processes

- Generated monitors are demonstrably constant time and constant space so they are free of many vulnerabilities

- Extensive suite of tests for each module

- Formal proofs that the C monitors are equivalent to the high-level specification

- Traceability from specification to monitor code and vice versa

- Support for end-to-end formal argument: proofs and tests that you have the right specification and proofs that the monitor being generated indeed implements the specification

Open Source Release

- We did an open-source release of Copilot
  - More hoops to jump thru than an internal release
- Security reviews to ensure we weren't exposing any sensitive information
- Everyone who had worked on the project had to sign releases
- The license of every library used must be documented and reviewed by legal
- NASA's open-source release process took almost 18 months
- Thanks to being open source have found users and contributors
  - Industry, academia, and hobbyists
- New contributions require additional steps:
  - Contributor License Agreement
  - Compliance with our process:
    1) File issue to document change, 2) Issue is formally assigned to contributor 3) Commits mention issue 4) Commit history is "clean" and rebased on top of HEAD, 5) All tests pass.

Sample Copilot Specification

Generating Fibonacci Sequence: 0,1,1,2,3,5,8,13,21,34,.....

```
fib :: Stream Word32
fib = [0,1] ++ (fib + drop 1 fib)
```
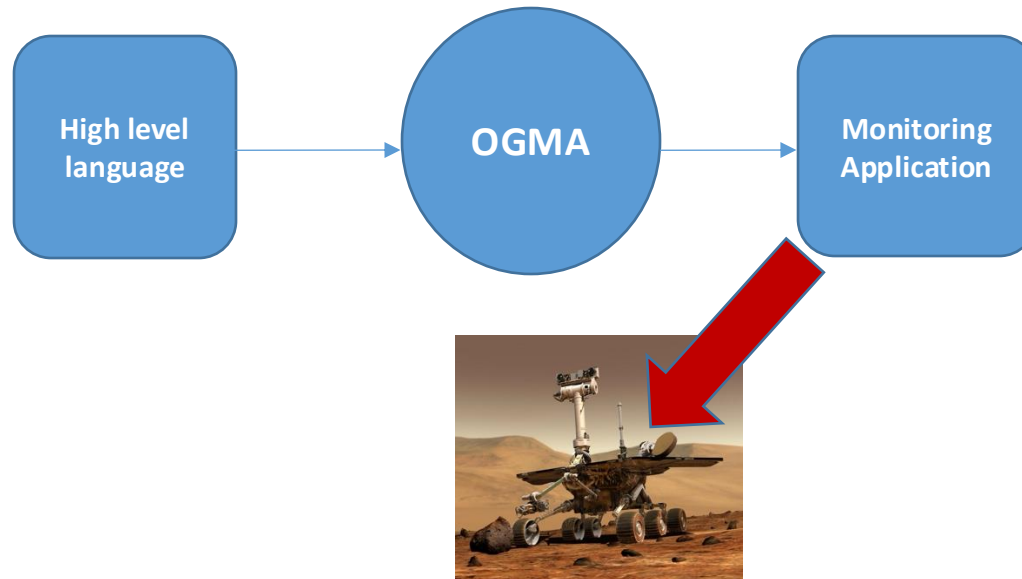
Developing Flight-ready Applications

- Errors instrumenting the system for monitoring or adding monitoring applications may lead to failures of the mission as a whole.

- We developed an application that auto-generates complete monitoring components, using Copilot under the hood to implement the core of the monitors.

Developing Flight-ready Applications

- Errors instrumenting the system for monitoring or adding monitoring applications may lead to failures of the mission as a whole.

- We developed Ogma, a tool that auto-generates complete, ready-to-run monitoring components/applications, using Copilot under the hood to implement the core of the monitors.
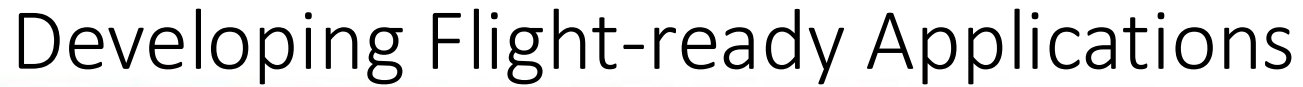
Developing Flight-ready Applications

- OGMA supports NASA cFS (Class A), JPL's F', the Robot Operating System, and standalone. Properties can be provided direclty via CLI, in separate files, or in files produced by MBSE tools.

```
ros-user@ros-laptop$ ogma ros -e 'mod x 2 /= 0' --variable-db variable-db --variables vars --app-name copilot --app-target-dir .
ros-user@ros-laptop$ find ros -type f | sort
ros/src/copilot/CMakeLists.txt
ros/src/copilot/package.xml
ros/src/copilot/src/copilot_logger.cpp
ros/src/copilot/src/copilot_monitor.cpp
ros/src/copilot/src/.keep
ros/src/copilot/src/monitor.c
ros/src/copilot/src/monitor.h
ros/src/copilot/src/monitor_types.h
ros-user@ros-laptop$ 
```

High-Assurance RV

- We aim to provide an end-to-end assurance argument
- Unit tests for each module
- Provide a range of specification logics
- Validate that the specification expresses the desired property
  - Interpret, simulate, prove
- Support fault-tolerant monitors
- Generate C99 monitors
- Generate proof that the monitors are bisimilar to high-level specification
- Automate integration with SUO
  - FRET, CFS, F', Frigate, ….

Developing Flight-ready Applications

- Ogma-produced applications can be directly integrated in the target system, as well as in existing simulation tools.