An Assured Open-Source Framework for Runtime Verification

Alwyn E. Goodloe
NASA Langley Research Center

Ryan Scott
Galois Inc.

Ivan Perez
KBR @ NASA Ames Research Center

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
  - Can be done offline or online
- Roots in the monitoring of system logs (1970s) and Simplex architecture (1990s)

Foundations of RV

- Given a specification ϕ
  - Specification logics: linear temporal logics (LTL), regular expressions, …
- A trace τ obtained from a system under observation (SUO)
- System must be instrumented to capture the trace
  - This affects what properties you can specify
- An RV monitor checks for language inclusion $\tau \in \mathfrak{L}(\phi)$
  - Accept all traces admitting ϕ
- RV frameworks generate monitors from a specification

Copilot

- Copilot is a runtime verification framework aimed at hard real-time safety-critical systems
- Employs sampling rather than extensive code instrumentation
  - Appropriate for monitoring safety of CPS systems
- Stream based specification language similar to Lustre
- Implemented as a Haskell Embedded Domain Specific Language (EDSL)
- Copilot specifications are translated into MISRA C99 monitors or to BlueSpec and Verilog for implementation in FPGAs
  - Exploring adding capability to generate Rust monitors
- Effort started in 2008
  - NASA research award (NNL08AA19 order NNL08AD13T)
  - Galois and the National Institute of Aerospace (NIA)

Copilot: The Early Years

- Copilot was very much run like an academic research project
  - A small army of many gifted interns from US and Europe
  - Sketch an idea, create a prototype, write paper, repeat …
- Flight tests with edge aircraft
- Pioneered creation of Byzantine fault-tolerant monitors
- Monitors used in a research setting
  - We wrote papers
- At no point did we argue to the flight safety folks that our monitors were mitigating hazards as identified in hazard analysis

Fork in the Road

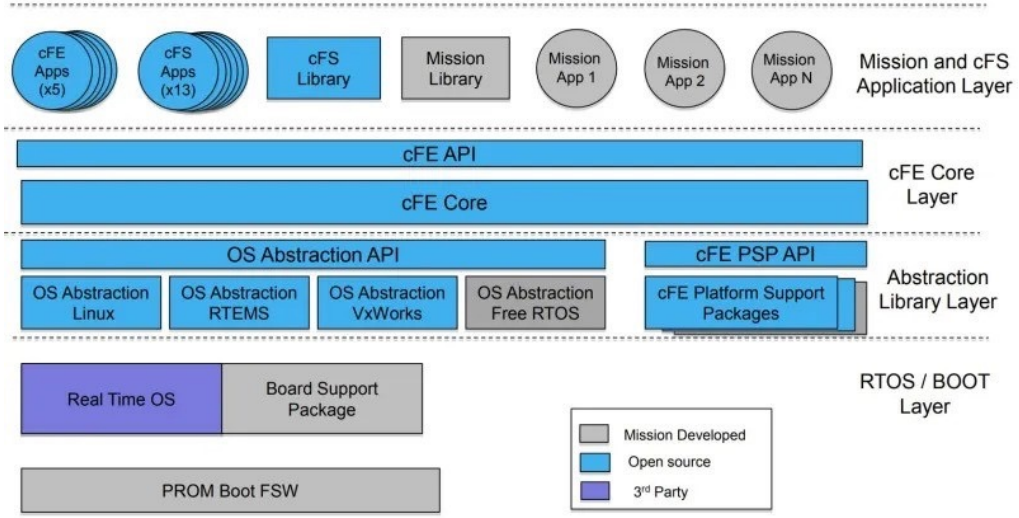
- We viewed Copilot as a very successful research tool
  - Great lessons learned
  - Good papers
- After years of hacking by, albeit amazingly gifted, students the code base was a bit of a mess
- In embedded systems, almost all the data we need to observe is kept in C-Structs or arrays and the framework didn't accommodate monitoring such data structures
  - Flight engineers found the framework too clunky to use so researchers were writing lots of custom code for each monitor
- Decision time:
  - Continue hacking away?
  - Rewrite the framework?

New Beginnings

- We decided to rewrite most of the framework
- We identified what needed to be removed
  - Poorly maintained open-source packages
  - Many “cool” never used features students had added
  - Dead code, poorly written code, ...
- Rewrote/restructured a large percentage of the code base
- Developed new backend for generating C code that could handle arrays and structs
- A challenge to translate nice-behaving Haskell structure into wild-west of C.

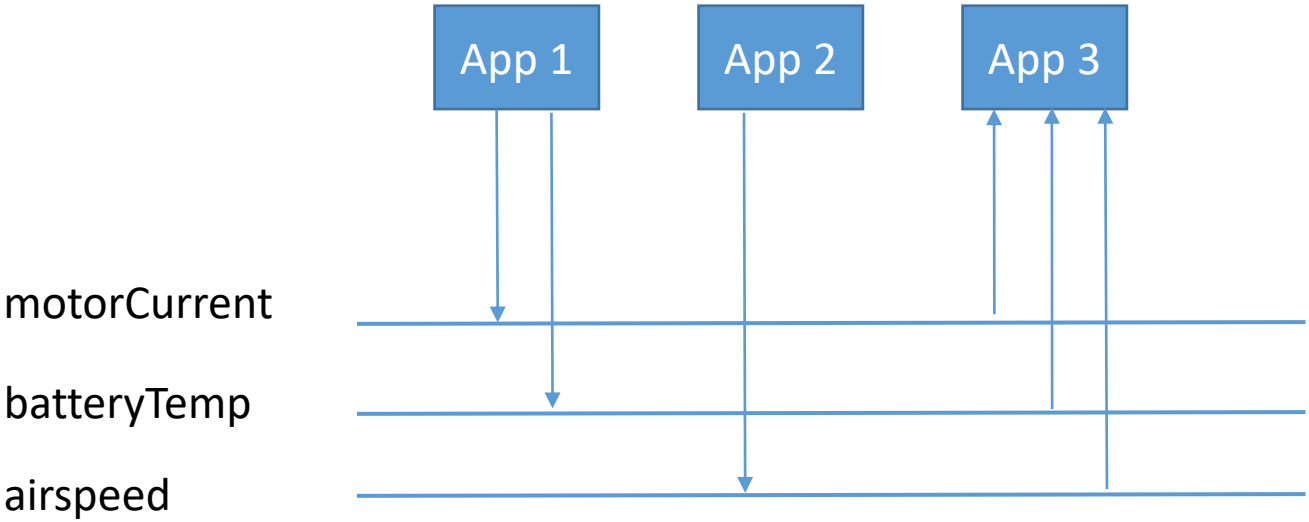
Developing Flight-ready Applications

- 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.



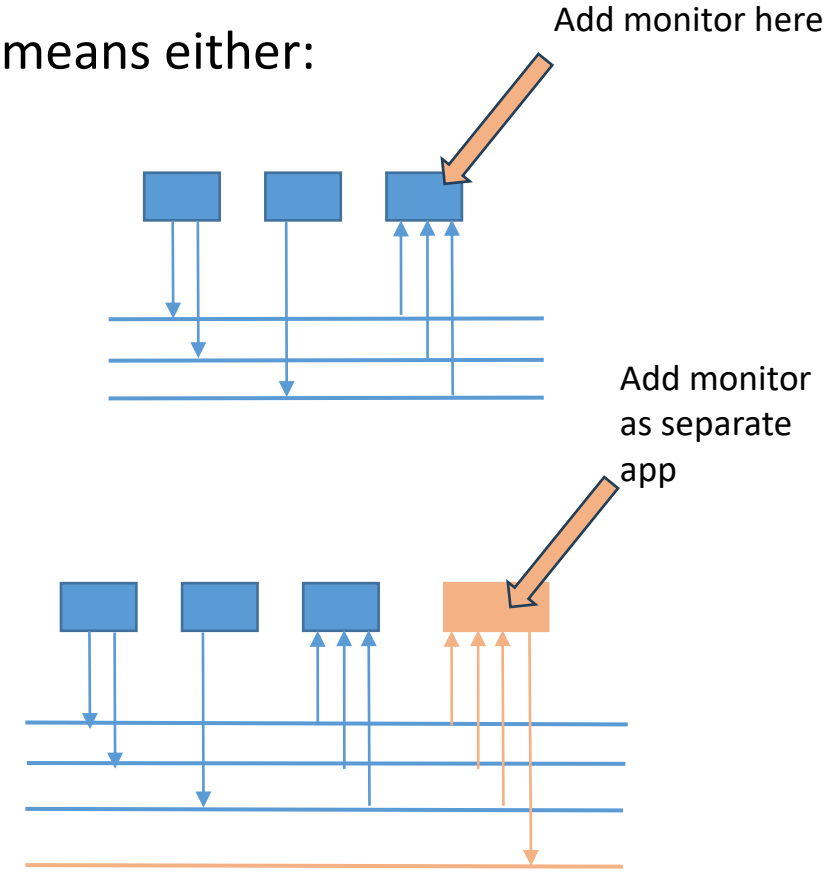
Developing Flight-ready Applications

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



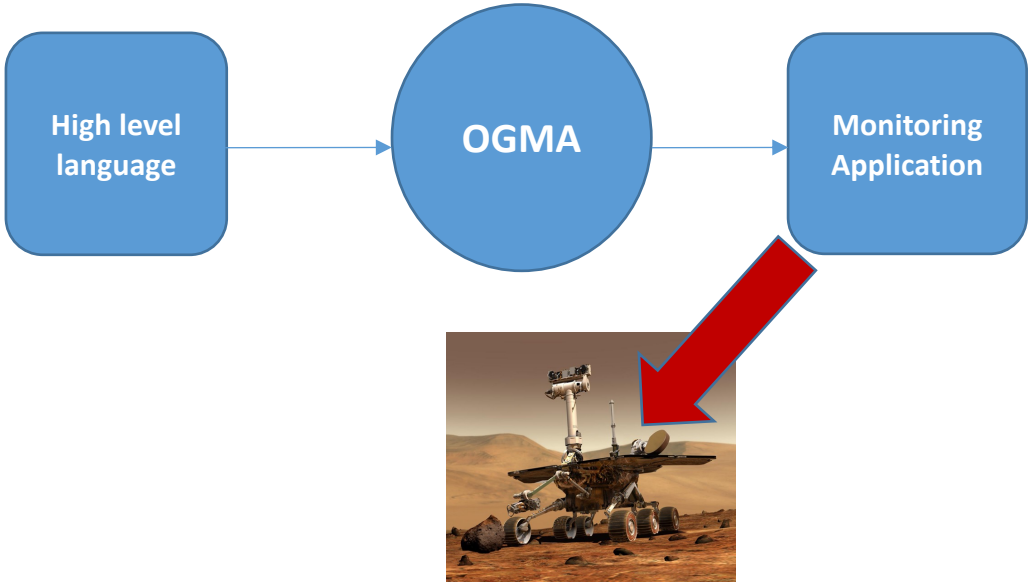
Developing Flight-ready Applications

- 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.



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.**



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

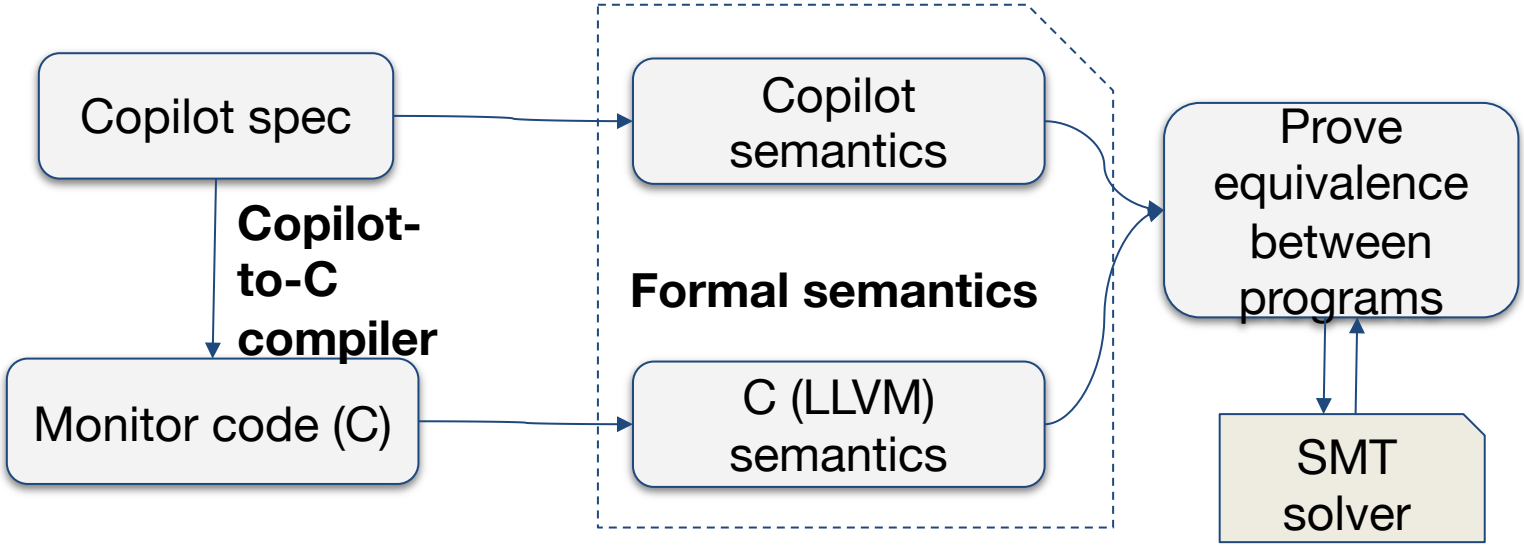
- Validating that you got the specification right is a critical aspect of assurance
- Copilot framework has integrated support for specification validation
- An interpreter lets users execute the specification without compiling to C and integrating with the SUO
- Theorem proving tools integrated into 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

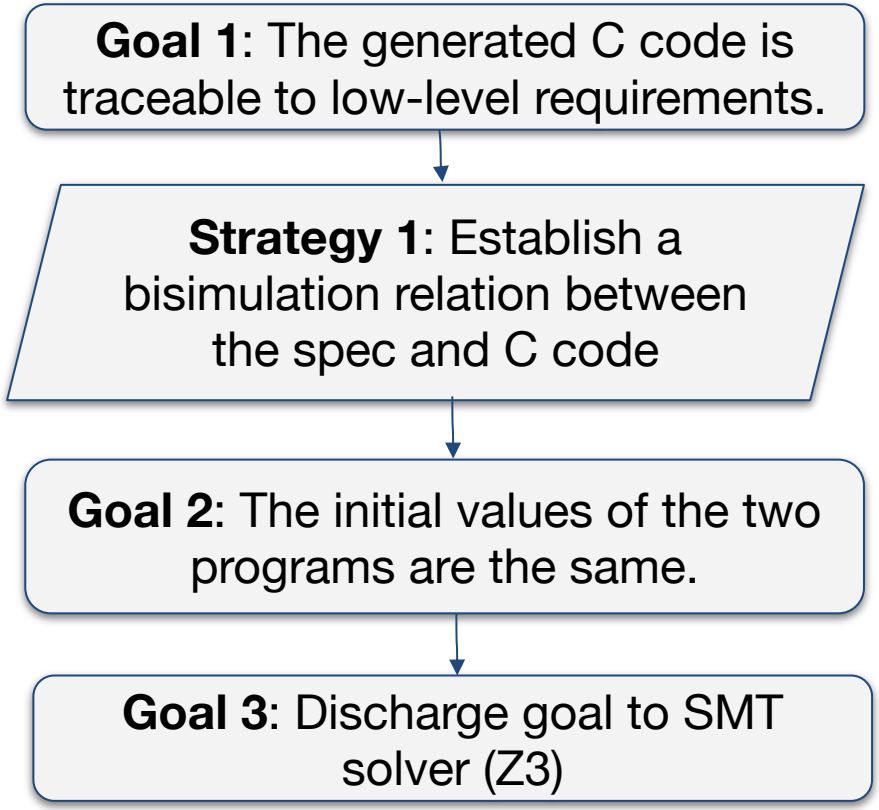
Copilot Monitors + Formal Verification

- 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.



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.



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.

Lessons

- Student work was relegated to being exploratory
- We could not have succeeded without having a full-time employee acting as development lead and a contractor to work on documentation
- Having users really helps the project mature and get funded
- Verification involved a mix of test and formal methods within a controlled development process
- Formal verification can be done on an industry schedule with the right team and setting meetable goals
- Formal verification requires a lot of education of the certification folks (an ongoing process)
- Open-source contributions need to be tightly controlled
- Nobody is going to do it for you and it is a long road, but totally worth it

Questions?

Contact Information:
Alwyn E. Goodloe
+1-757-864-5064
a.goodloe@nasa.gov

Copilot Language Operators

- Stream language – constants and arithmetic operations are lifted to stream level
  - Semantics of Haskell lazy lists
- (++) :: [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

Sample Copilot Specification

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

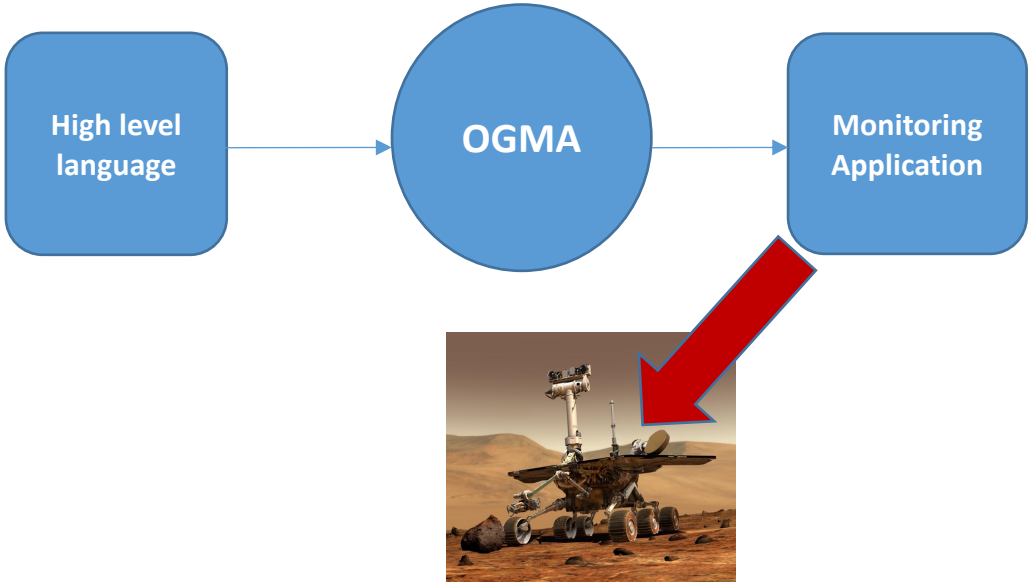
```
fib :: Stream Word32
fib = [0,1] ++ (fib + drop 1 fib)
```

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.