RDE Example

High-Assurance Cyber-Physical Systems

Digital Instrumentation & Control (DI&C)

Reactor Trip System (RTS)

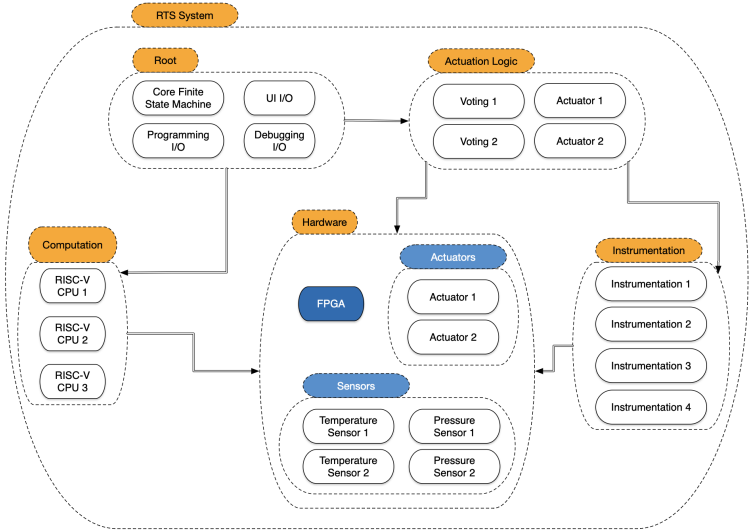
- The RTS is a demonstration DI&C system, which are basically sense-compute-control architectures that control safety-critical systems,
- have human-in-the-loop user interfaces,
- commonly have built-in self-test subsystems that must be able to self-test/assure a system while it is in operation,
- are fault-tolerant and have heterogenous components, where components are implemented using multiple techniques, sometimes by multiple teams, and have multiple redundant connectors,
- are built today within the NPP industry without software or COTS hardware, and
- the NRC does not want to see the industry repeat the technology mistakes of other nationally critical infrastructure industries.

NPP = nulcear power plant; COTS = commercial off the shelf

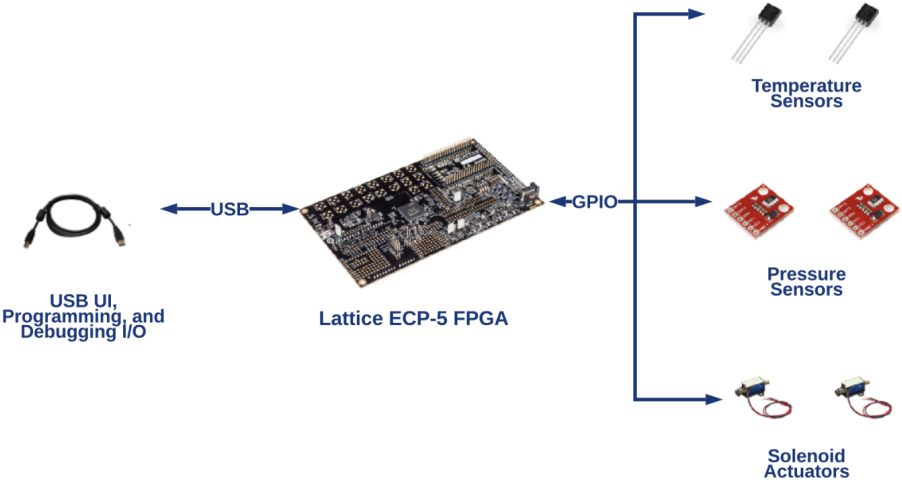
Base System Architecture

- Four redundant divisions of instrumentation, each containing identical designs:
  - Two instrumentation channels (Pressure and Temperature)
    - Sensor
    - Data acquisition and filtering
    - Setpoint comparison for trip generation
    - Trip output signal generation
- Two trains of actuation logic, each containing identical designs:
  - Two-out-of-four coincidence logic of like trip signals
  - Logic to actuate a first device based on an OR of two instrumentation coincidence signals
  - Logic to actuate a second device based on the remaining instrumentation coincidence signal

RTS Proposed Architecture Sketch



RTS Hardware



Functions to Be Implemented

1. Trip on high pressure (sensor to actuation)
2. Trip on high temperature (sensor to actuation)
3. Trip on low saturation margin (sensors to actuation)
4. Vote on like trips using two-out-of-four coincidence
5. Automatically actuate devices
6. Manually actuate each device
7. Select mutually exclusive maintenance and normal operating modes on a per division basis

8. Perform setpoint adjustment in maintenance mode
9. Configure the system in maintenance mode to bypass an instrument channel (prevent it from generating a corresponding active trip output)
10. Configure the system in maintenance mode to force an instrument channel to an active trip output state
11. Display pressure, temperature and saturation margin
12. Display each trip output signal state
13. Display indication of each channel in bypass
14. Periodic continual self-test of safety signal path (e.g., overlapping from sensor input to actuation output)

IEEE Standard 603-2018 — Characteristics to be demonstrated

1. [**CCC**] Completeness and consistency of requirements
2. [**Instrumentation independence**] Independence among the four divisions of instrumentation (inability for the behavior of one division to interfere or adversely affect the performance of another)
3. [**Channel independence**] Independence among the two instrumentation channels within a division (inability for the behavior of one channel to interfere or adversely affect the performance of another)
4. [**Actuation independance**] Independence among the two trains of actuation logic (inability for the behavior of one train to interfere or adversely affect the performance another)
5. [**Actuation correctness**] Completion of actuation whenever coincidence logic is satisfied or manual actuation is initiated
6. [**Self-text/trip independance**] Independence between periodic self-test functions and trip functions (inability for the behavior of the self-testing to interfere or adversely affect the trip functions)

How to get from Characteristics to Formally Verified System Properties

- characteristics are **precise, semi-formal English requirements** about an amorphous system
- in order to understand a characteristic, we must understand its constituent terms
- in order to verify a characteristic, we must translate in a traceable fashion semi-formal characteristics into semi-formal requirements, and hence to formal, checkable properties about the system
- properties are **decidable, checkable predicates** about systems, software, firmware, and hardware, which can be **checked by runtime verification (rigorous testing) and formal verification**

Queries	Level	Artifacts
What must it do?	Requirements	Lando, FRET
How do the parts interact?	Architecture	SysML
How will it do it?	Specifications	Cryptol
Is it built right?	Verification and correctness	Cryptol
What does it do?	Behavior	Verilog
Does it do the right thing?	Validation and testing	—
How do we build it?	Implementation	FPGA

Requirements

The RTS demonstrator treated requirements as formal, analyzable artifacts.

- **Lando** was used to rewrite natural language requirements into a **structured, semantic format**.
  - Enabled clarification, modularity, and alignment with domain concepts.
  - Served as a **bridge** between informal stakeholder intent and formal tools.
- **FRET (Formal Requirements Elicitation Tool)** captured precise, logic-based properties:
  - Expressed **assumptions** and **guarantees** over time and behavior.
  - Requirements were made **machine-checkable**, and suitable for **formal verification**.
  - Used as input to tools like **Cryptol** and **Frama-C**.

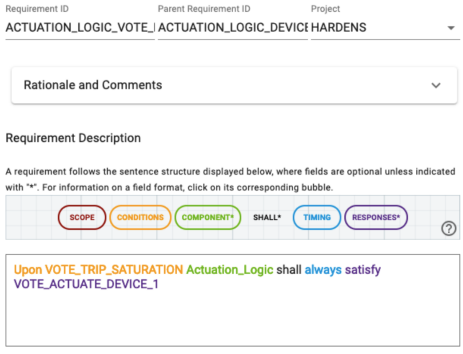
Together, Lando and FRET formed a pipeline from informal intent to formal verification.

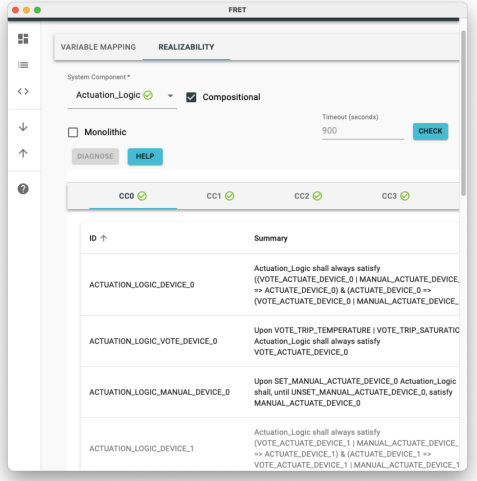
Requirements using FRET[1]

- FRET is a tool for writing, understanding, formalizing, and analyzing requirements.
- Users write requirements in an intuitive, restricted natural language, called FRETISH, with precise, unambiguous meaning.
- For a FRETISH requirement, FRET:
  1. produces natural language and diagrammatic explanations of its exact meaning,
  2. formalizes the requirement in future-time and past-time temporal logic, and
  3. supports interactive simulation of produced logic formulas to ensure that they capture user intentions.
- FRET connects to analysis tools by facilitating the mapping between requirements and models/code, and by generating verification code.

[1]This entire summary is straight from "Formal Requirements Elicitation with FRET" by Giannakopoulou, et al. REFSQ-2020.

STS FRET Editor and Realizability





Architecture

The RTS system architecture was modeled to support structure, safety, and traceability.

- The architecture captured the **decomposition of subsystems**:
  - Sensor logic, redundant voting, and actuation behavior
  - Clear separation of hardware and software responsibilities
- **SysMLv2** was used to express:
  - System components, interfaces, and data flows
  - Allocation of functions to hardware/software
  - Behavioral constraints and stakeholder interactions
- SysML enabled:
  - **Model-based traceability** across layers
  - Planning for **verification and validation**
  - Mapping from requirements to implementation

Architecture modeling provided a foundation for structured refinement and assurance.

Domain Engineering Model

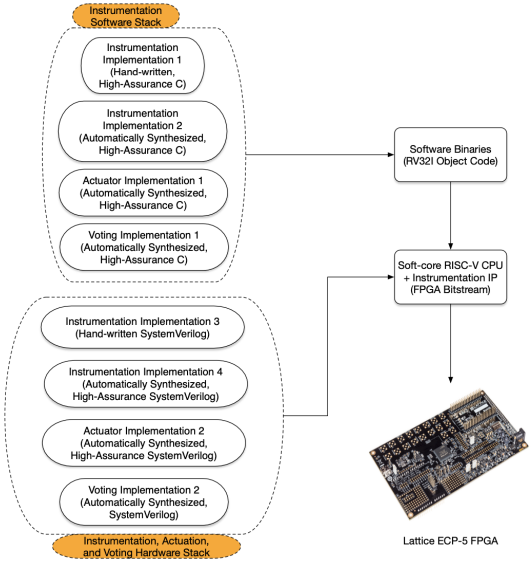
- A domain engineering model is a high-level, formal description of a domain and its properties.
- Think of it as a glossary of concepts (nouns, verbs, adjectives, adverbs) and their relations that is formalized in a machine interpretable fashion.
- Domain engineering models provide a semantics for requirements engineering.
- Such mechanizations are achieved using a variety of formal methods and technologies.
- Historically, formal methods like B and Event-B, RAISE, VDM, and Z are used for critical systems.
- In the modern day these are augmented by formal methods and tools like Alloy, Coq, Lando, and PVS.
- The domain engineering model for HARDENS is specified in **Lando and SysMLv2**. The background theory for the Lando is specified in Higher-Order Logic (HOL) in Coq and PVS.

Domain Engineering Model Example

```
subsystem RTS Hardware Artifacts

The physical hardware components that are a
part of the HARDENS RTS

demonstrator.


component USB Cable

A normal USB cable.

What kind of USB connector is on the start of
the cable?

What kind of USB connector is on the end of
the cable?


relation USB Cable inherit USB, Cable

…
```

```
component Temperature Sensor

A sensor that is capable of measuring the
temperature of its environment.

What is your temperature reading in Celsius
(C)?


component Pressure Sensor

A sensor that is capable of measuring the air
pressure of its environment.

What is your pressure reading in Pascal (P)?


component Solenoid Actuator

A solenoid actuator capable of being in an
open or closed state.

Close!

Open!


relation Temperature Sensor inherit Sensor

relation Pressure Sensor inherit Sensor

relation Solenoid Actuator inherit Actuator
```

RTS Instrumentation Architecture



RTS Instrumentation Software Stack

The RTS instrumentation software processes sensor data and implements trip logic.

- **Evaluates sensor conditions**:
  - Thresholds, rates of change, limit violations
- **Implements redundant voting logic**:
  - Determines validity across multiple inputs
  - Avoids false positives due to single-sensor faults
- **Supports channel independence**:
  - Each processing path is isolated for fault tolerance
- **Coordinates actuation decisions**:
  - Triggers output logic when trip conditions are confirmed

This logic was developed in formally analyzable forms (Cryptol, C) and verified using tools like SAW and Frama-C.

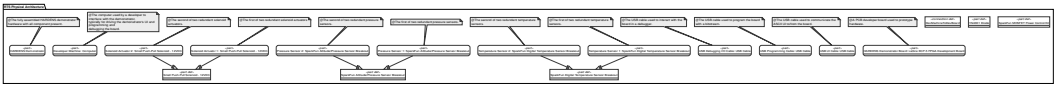
RTS Instrumentation, Actuation, and Voting Hardware Stack

This hardware stack implements the physical control logic behind sensing and actuation.

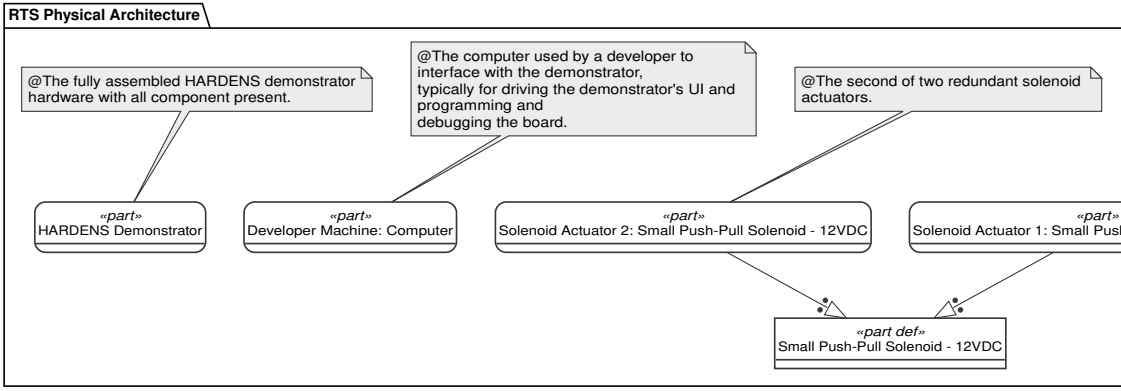
- **Sensor interfaces**:
  - Handle analog input conditioning and digitization
- **Voting logic circuits**:
  - Perform majority voting (e.g., 2-out-of-3)
  - Ensure robust decisions in the presence of sensor faults
- **Actuation logic**:
  - Drives trip signals to external safety systems
  - Enforces correct and timely shutdown behavior

Implemented in **SystemVerilog** and **Bluespec**, the logic was verified against the software models and supports real-time, fault-tolerant control.

SysML Model



SysML Model


RTS Physical Architecture
@The fully assembled HARDENS demonstrator hardware with all component present.
@The computer used by a developer to interface with the demonstrator, typically for driving the demonstrator's UI and programming and debugging the board.
@The second of two redundant solenoid actuators.
«part»
HARDENS Demonstrator
«part»
Developer Machine: Computer
«part»
Solenoid Actuator 2: Small Push-Pull Solenoid - 12VDC
«part»
Solenoid Actuator 1: Small Push
«part def»
Small Push-Pull Solenoid - 12VDC

Specifications

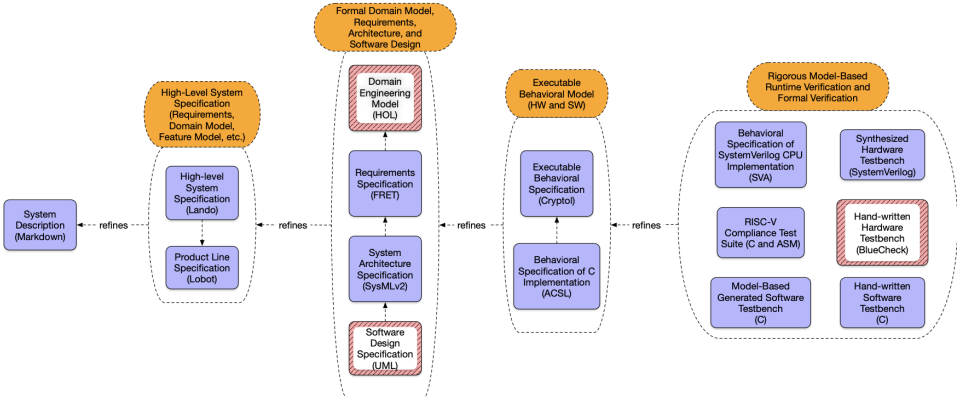
Specifications define **how the RTS system will meet its requirements** — describing intended behavior, structure, and responses.

In the HARDENS project, specifications were developed using:

- **Architecture-informed logic** to define how sensor inputs are processed, compared, and acted upon
- Precise rules for **redundant voting**, **fault masking**, and **actuator triggering**
- Conditions for **normal operation**, **error handling**, and **edge-case behaviors**
- Modeled behavior that accounts for timing, sequencing, and state transitions

These specifications were written in a form suitable for **formal verification**, and directly informed the development of **verifiable implementations** and **testable simulations**.

RTS Specifications



Verification and Correctness

Verification was a core focus of the RTS demonstrator:

- **Cryptol** was used as a formal verification tool:
  - Modeled sensor logic, redundancy, and actuation behavior
  - Theorems were proved against FRET specifications
  - Properties verified using **SAW** (Software Analysis Workbench)
- **Frama-C + ACSL** verified C implementations against behavior and safety contracts
- Hardware logic (SystemVerilog, Bluespec) was aligned and checked

This multi-layer verification ensured correctness across software and hardware.

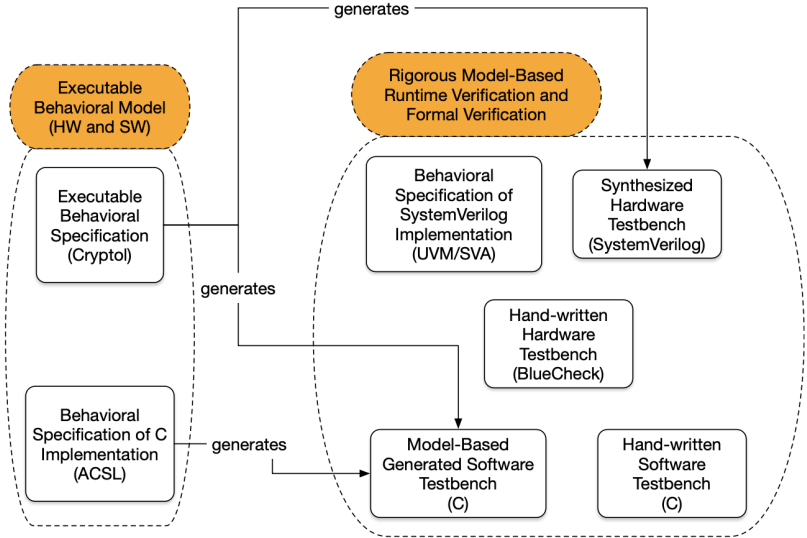
Behavior

System behavior was expressed and simulated through:

- Temporal and logical properties in **FRET**
- Executable functional models in **Cryptol**
- Control logic verified in C and SystemVerilog

Behavioral properties were validated against expectations from the original requirements, ensuring correct responses to all relevant inputs.

RTS Behavior Artifacts



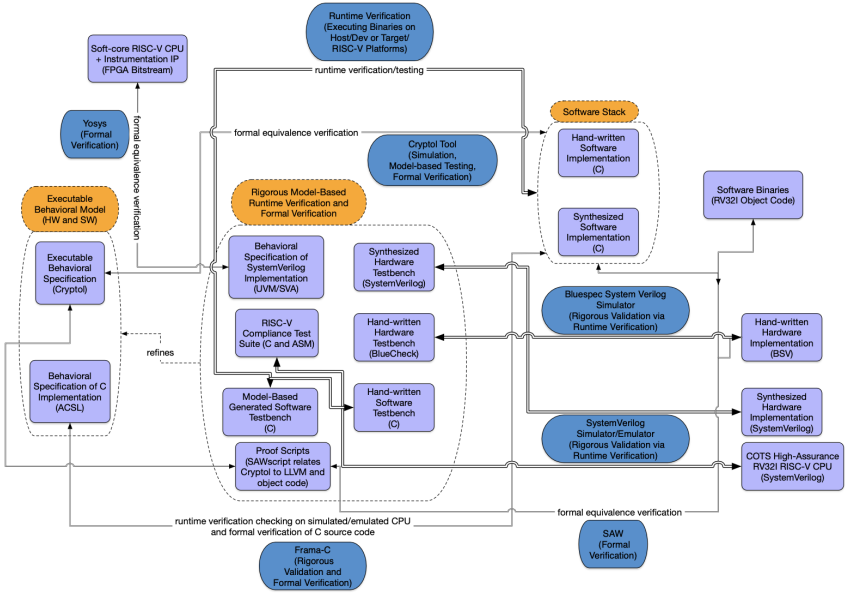
Validation and Testing

Validation strategies included:

- **Simulation of Cryptol models** to explore behavior
- **Test benches** generated from specifications
- **Runtime monitors** aligned with formal properties
- Hardware validation using assertion-based test environments

These efforts ensured that observed behavior aligned with the modeled expectations and safety goals.

RTS Validation Artifacts



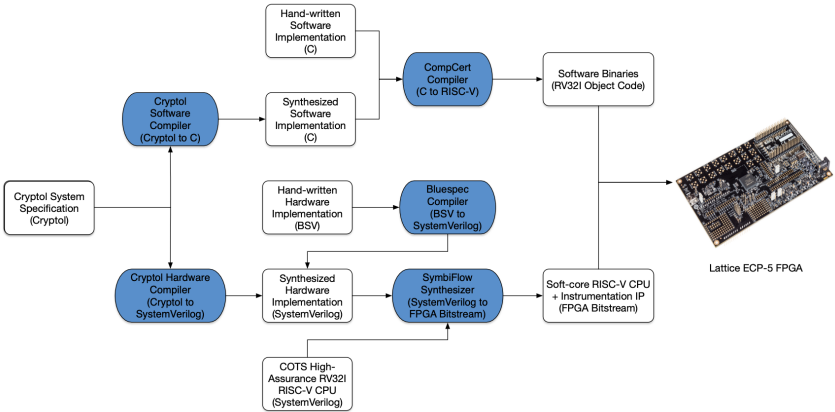
Implementation

Final implementation artifacts were grounded in the verified specifications:

- **Software**:
  - C code verified using Frama-C and SAW
  - Aligned with verified Cryptol specifications
- **Hardware**:
  - HDL implementations cross-checked with functional models
  - Assertion coverage supported traceability

All implementation artifacts were packaged with evidence of correctness, traceability to requirements, and structured documentation for review.

RTS Implementation Artifacts



Summary: The RTS Engineering Stack (Revised)

Layer	Approach Used in HARDENS RTS
Requirements	Natural language refined with Lando and FRET
Architecture	Modeled component breakdown and control flow
Specifications	Defined in FRET as formal, analyzable contracts
Verification/Correctness	Cryptol + SAW, Frama-C, assertion checking
Behavior	Expressed in FRET and executable in Cryptol
Validation & Testing	Simulated models, test benches, monitors
Implementation	Verified C and HDL with evidence

This workflow enabled end-to-end assurance through formal structure and layered verification.