galois

HARDENS: Explaining Rigorous Digital Engineering by Example

Joe Kiniry, Galois (kiniry@galois.com)

April 2022

This work is supported by the U.S. Nuclear Regulatory Commission (NRC), Office of Nuclear Regulatory Research, under contract/order number 31310021C0014.

All material is considered in development and not a finalized product.

Any opinions, findings, conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the NRC.

galois

Correctness · Cryptography · Security · AI and ML

Advancing computer science R&D
Creating trustworthiness in critical systems

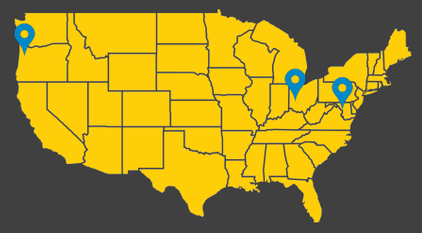
Clients

DOE DARPA NASA

DHS AMAZON NIST

Offices

Portland, OR
Dayton, OH
Arlington, VA



History

Founded in 1999
100+ employees

A different kind of company

No managers

We have no fixed hierarchy of rigid positions and titles, and no traditional managers.

Radical transparency

Everything is transparent by default: company financials, decision making, open meetings, and even salaries.

Choose your work

Research engineers choose the projects they work on, and move freely between projects depending on personal interests and career goals.

Ownership

Employees own the majority of the company together, making important decisions as a group and partaking in the financial success of the company.

More info at lifeatgalois.com

Outline

- Background on Rigorous Digital Engineering
  - Historical Perspective
  - Summary of Rigorous Digital Engineering
- The HARDENS Case Study
- Reflections and Next Steps

The RDE Research Program

- **Rigorous Digital Engineering** (**RDE**)
  - **Rigorous** = use *applied formal methods* to reason about *models, implementations, and evidence*
  - **Digital** = use digital, mechanized, computational, denotational and executable models of components and systems to create *digital twins* (executable digital representations of components and systems) and *digital threads* (relations between digital and physical artifacts with known, evidence-based fidelity)
  - **Engineering** = process, methodologies, tools, and technologies supporting *all* forms of engineering (particularly domain, requirements, software, firmware, hardware, safety, systems, and security engineering)

Historical Perspective on RDE

- RDE combines rigor, models, and engineering
- My 1980s and 1990s focused on software engineering, pure & applied mathematics, and computational science (simulation, DSLs, and user experience), and cybersecurity (as a broad spectrum “whitehat”)
  - focused on numerical analysis, multi-resolution finite element methods for computational fluid dynamics on supercomputers (particularly hypersonic fluid flows)
    - computational tools for simulating, analyzing, and visualizing scientific experiments (particularly particle beam accelerators
- My 1990s focused on formal methods, hardware design, and software, software and hardware specification languages and formal verification, and systems and modeling languages
- My 2000s focused on verification of real world systems, programming languages, formal methods for security
- My 2010s brought all of these areas back together for U.S.G. and commercial customers, and then the U.S.G. branded Digital Engineering

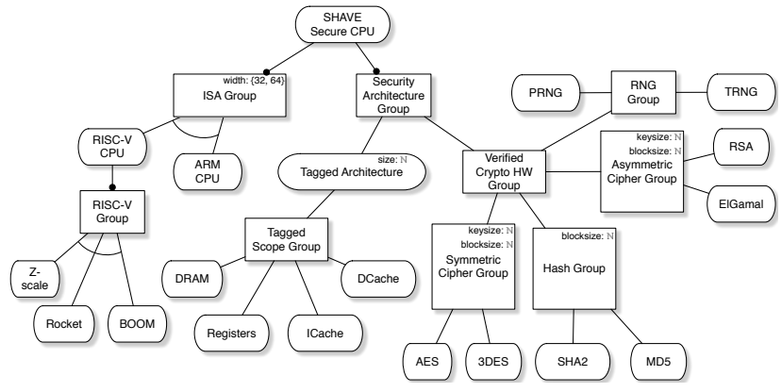
Models and Reasoning at Galois

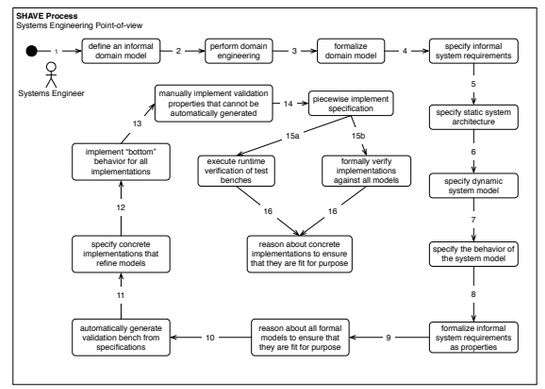
- models are primarily those that support RDE
  - digital, mechanized, computational, denotational and executable models of components & systems
- generally, models must have well-understood and maintainable relationships to other models and implementations
- models are either written by hand, generated from other artifacts (semi-formal or formal), or lifted/extracted from implementations (software, binaries, & hardware designs)
- reasoning about models and implementations is accomplished using one of dozens of formal reasoning techniques
  - interactive specification and proofs in a Logical Framework
  - automated reasoning with SAT/SMT
  - type systems, symbolic evaluation, and logic-based reasoning with various Hoare, separation, and domain-specific logics are popular
  - model checking and abstract interpretation, less so

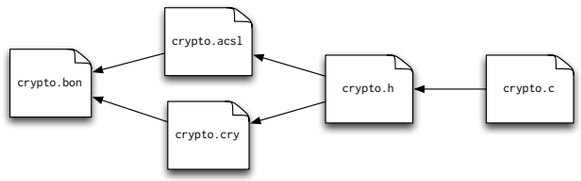
The RDE “Big Picture”

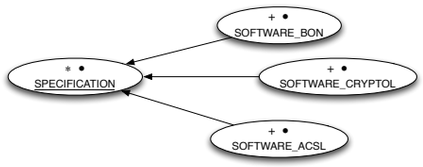
Rigorous Digital Engineering…

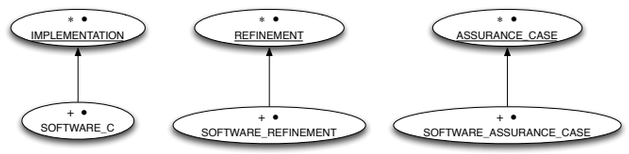
- Rigorous Digital Engineering (RDE) is all about…
  - the use of (preferably executable) models (with preferably known fidelity) to
  - rigorously, authentically describe things
  - at various levels of abstraction
  - and the models relate to each other
  - in well-understood ways
  - and the models refine to bits or atoms
  - and thus all of this connects to software, hardware, and systems engineering
  - and we use the models to provide assurance of various kinds for the product line/product/platform/system











…with Applied Formal Methods

- applied formal methods is about the practical application of formal methods to all stages of a system's life cycle
  - process, methodology, design, development, assurance, maintenance, evolution
- hold no bias in choice of formal method, tool, or technology—just choose the right tool for the job
- often focuses on find key places where small changes to the lifecycle have large impact
- and nearly always hides formalism from the typical user a la *Secret Ninja Formal Methods*

Engineering Today

- most systems are described today with English and source code; at best, descriptions include
    - unit or subsystem tests describing a tiny fraction of correct behavior with code
    - ad hoc assertions and logging messages
    - fairly precise developer documentation on source code and in a developer's guide
    - understanding and fixing flaws in a system is accomplished by using logging, print statements, and debuggers
- despite all of the excitement about model-driven engineering over the past twenty-odd years, very few models exist for the vast majority of systems
    - where have models succeeded? data models for DBs and networking and UI/UX models for GUI development

RDE at Galois/F&F in a Nutshell

- we have been defining, teaching about, and using RDE (or some ancestor thereof) since the mid-90s
  - most junior developers have been freshmen computer science students; most senior developers have been software and hardware engineers with decades of real world experience
- our approach is grounded in past R&D starting largely in the 60s in formal methods and software engineering
  - simply aggressively evolve ideas in practical ways over the decades as we care about new languages, platforms, scales, domains, reasoning tools, and concepts
- we know that the approach works because we have many hundreds of case studies demonstrating its impact and outcomes
- in some sense, we do what IT consultants do in the real world (write requirements, draw architectural sketches in Powerpoint, play with UML diagrams, have acceptance criteria, etc.), except the artifacts we create, use, and reuse have meaning and are not just pictures

Past, Current, and Future Customers

- in the past, we and the community, have mostly used RDE on many safety- and mission-critical systems
  - finance, elections, biomedical, avionics, aeronautics, cyber-physical, ASIC design, etc.
- but over the past twenty years, my teams have applied RDE to “normal” systems development, demonstrating over and over that it saves time and money and increases quality
- Model-Based Engineering remains of interest to our customers & Digital Engineering is the new hotness
- the DoD is now very focused on understanding what MBE is, what DE should be, particularly to save time and money while increasing quality

Risks

- engineers and organizations are very comfortable and are resistant to change
  - the challenges are rarely technical, they are mostly sociological, psychological, political
- RDE-shaped things often are adopted only when
  - leadership mandates (top-down)
  - society forces it (statutes, policy, certification, financial or human life risk mitigation, or lawsuits)
  - companies realize competitive benefit ($$)
- to demonstrate ROI one must measure and reflect
- to have RDE “stick”, one must deliver useful technology and practices that are culturally adopted

Impact

- adopting RDE ideas typically…
  - decreases effort, and thus cost
  - shortens POPs
  - changes focus and shifts thinking “left” and “up”
  - significantly decreases the volume and changes the nature of bugs found during development
  - increases assurance and changes its nature
  - helps certification, risk analysis, and transition

The Technologies of RDE

The technology stacks supported thus far by the RDE methodology include:

- many different kinds of programming languages (procedural, object-oriented, functional, hardware, logic, and mixed-model, such as C, C++, C#, Rust, Haskell, Java, Scala, Kotlin, Eiffel, Chisel, Bluespec SystemVerilog, System Verilog, VHDL),
- specification and modeling languages (such as F*, ACSL, JML, CodeContracts, Alloy, Z, VDM, Event-B, RAISE),
- architecture specification tools and languages (such as Cameo, Rhapsody, MagicDraw, OSATE, Visual Paradigm and UML, AADL, and SysML, resp.),
- integrated development environments (such as Eclipse, Visual Studio, VS Code, IDEA, Eagle, KiCAD, mbeddr, Rodin, OSATE, Crescendo, etc.),
- formal modeling and reasoning tools (such as Alloy, PVS, Coq, Isabelle, UPPAAL, CZT, Overture, Rodin, Frama-C, SAW, Ivy, TLA Toolbox, FDR4, NuSMV, BLAST, and SPIN),
- operating systems (RTOSs, UNIX variants, seL4, etc.), and
- spans systems, hardware (ASIC and FPGA-based), firmware, and software

Case Studies: Galois

- the BESSPIN top-level project
- the SHAVE (Software/Hardware Assurance Verified End-to-end) project (high-assurance hardware/firmware/software)
- BESSPIN-Voting-System-Demonstrator-2019: source code and design documentation (both software and hardware) for the smart ballot box and accompanying software that was used in the BESSPIN DEF CON 2019 voting demonstrator (see https://securehardware.org/)
- the DARPA FETT Bug Bounty Contest Voter Registration System BESSPIN-Voter-Registration
- the SSITH Cyberphysical Demonstrator

Case Studies: Free & Fair

- STAR-Vote – an end-to-end verifiable voting systems prototype
- OpenRLA – our first prototype Risk-Limiting Audit system
- Qubie – a compact privacy-preserving polling place monitoring system
- Tabulator – tallies digital Cast Vote Records, specified in an open JSON-based format, into an election result
- ElectionGuard (Specs) – an E2E-V voting cryptographic SDK for Microsoft
- ColoradoRLA – a distributed concurrent Risk-Limiting Audit system built for the State of Colorado
- Formal-RCV – a formalization of Rank Choice Voting in Coq
- DIVS – a rigorously engineered tabulator for the Danish voting scheme
- Votail – a formally verified tally system for Ireland's method of Proportional Representation by Single Transferable Vote (PR-STV)
- Logging – a formally verified elections logging system
- DVL – a digital/electronic pollbook system

Hard Problems of Past Focus 1

1. How to give models semantics that are usable and useful up and down the traceability refinement chain that is RDE?
2. How to define and use traceability with semantics?
3. How to guarantee reversibility of traceability via refinement?
4. How to remove ambiguities in interpreting English into models and properties?
5. How to use semi-formal domain models and domain engineering as a means by which to automatically perform interpretation between English specifications and concrete, unambiguous models and properties about implementations?
6. How to ensure that graphical/visual representations of systems and their properties have unambiguous semantics and are completely semantically consistent with textual representations and can be updated (either way) automatically as one writes or edits specifications?
7. How to specify and reason about not just models, not just code, not just model-code relationships, and not just systems, but systems of systems that include concurrency, distribution, and cryptography?
8. How to weave process, methods, and tools into an agile methodology that is supported by IDEs?
9. How to define, create, and augment popular IDEs (such as Eclipse, Visual Studio, Visual Studio Code, IDEA, Emacs, etc.) that embody the methodology and its pedagogy to help engineers continuously improve their knowledge and capabilities?

Hard Problems of Past Focus 2

10. How can applied formal methods be used for model validation, rigorous systems validation and verification, and certification?
11. How to move beyond specifying and reasoning about singular products, but instead focus on the specification of, reasoning about, and creation of certified product lines?
12. How to guarantee predictability and determinism in the development and deployment environment using container technologies and deterministic package managers, especially for complex embedded systems engineering with bespoke hardware, a la BESSPIN Infrastructure?
13. How to guarantee consistency in rigorous systems engineering across the whole of systems of systems (hardware/firmware/software) and throughout a product line's lifecycle?
14. How to perform tradeoff analysis based upon verifiable evidence and the conceptual, team, programmatic, contractual, and legal constraints of a particular product line's environment? What dimensions of measurable metrics can/should be involved in such analysis? Those that we have studied and used so far include: performance, latency, throughput, power, energy, resource use (memory and other finite resources limited by a given platform's hardware or OS), area (wrt ASICs), cost, comprehensibility/understandability, rigor of assurance case, and cost and effort of certification.

RDE Course Under Development

1. RDE Overview
2. The RDE Process
3. The RDE Methodology
4. Domain Engineering
5. Requirements Engineering
6. Product Line Engineering
7. Architecture
8. Models
9. Refinement
10. Design
11. Development
12. Validation
13. Verification
14. Evolution and Maintenance

HARDENS

- **HARDENS** (*High Assurance Rigorous Digital Engineering for Nuclear Safety*) is a R&D project run by Galois for the *Nuclear Regulatory Commission* (NRC)
- the purpose of HARDENS is to demonstrate and educate about cutting-edge, high-assurance model-driven engineering
  - our focus is on nationally critical infrastructure, and thus safety-critical embedded systems
- within HARDENS, Galois has designed and built a demonstration Reactor Trip System (RTS) that is representative of a Digital Instrumentation & Control (DI&C) system for a Nuclear Power Plant (NPP)
  - the RTS is fault-tolerant and high-assurance
  - the RTS has a physical manifestation (an FPGA board plus sensors/actuators) and a set of digital twins

The Reactor Trip System (RTS)

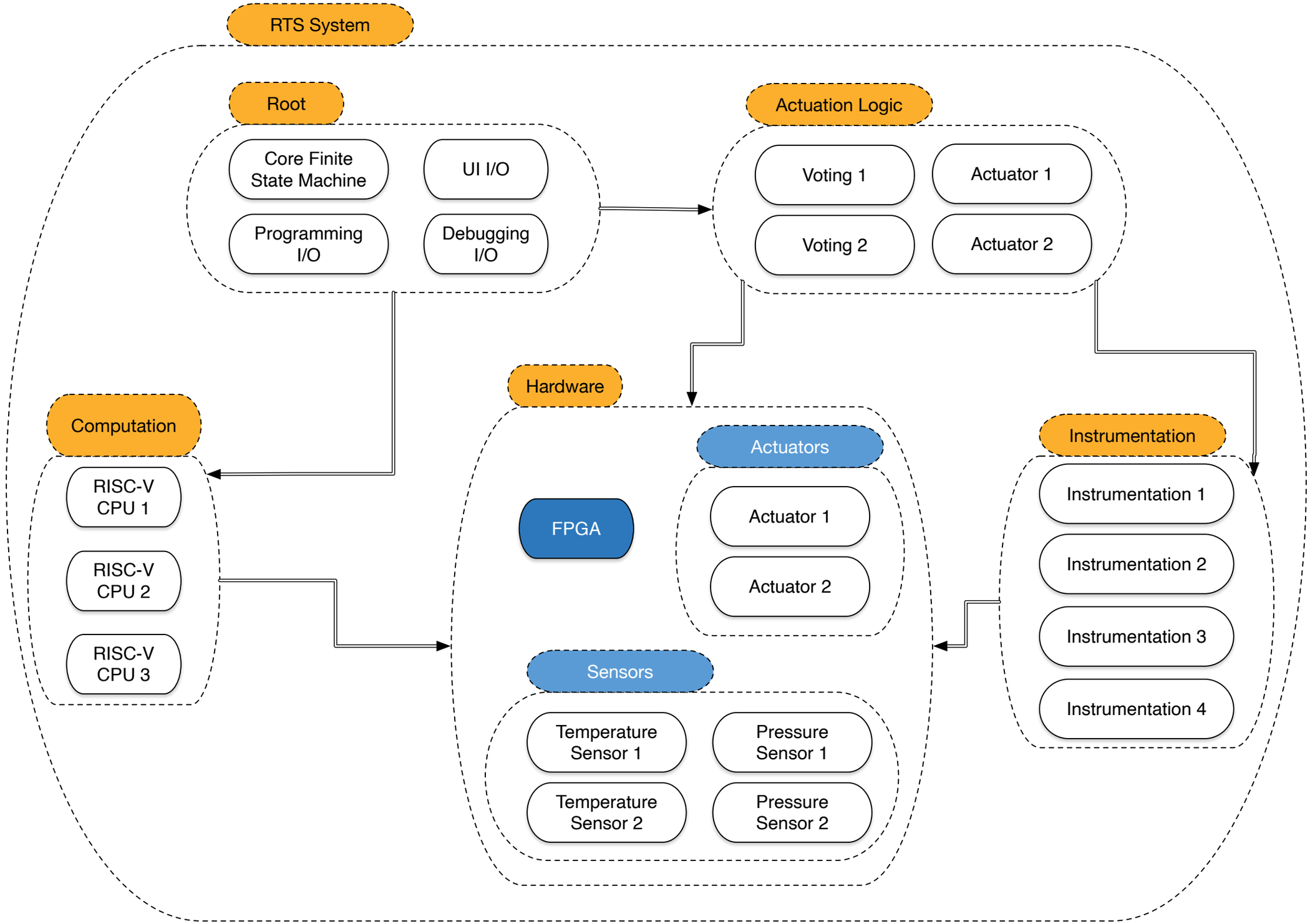
- a seemingly simple control system that measures temperature and pressure of a reaction vessel and actuates solenoids to open and close valves
- representative of a class of systems that are deemed nationally critical infrastructure by the DHS
- a fault-tolerant system with *heterogenous redundancy*
- must fulfill a set of assurance “characteristics” found in the IEEE Standard 603-2018 “IEEE Standard Criteria for Safety Systems for Nuclear Power Generating Stations”

An Overview of the RTS System

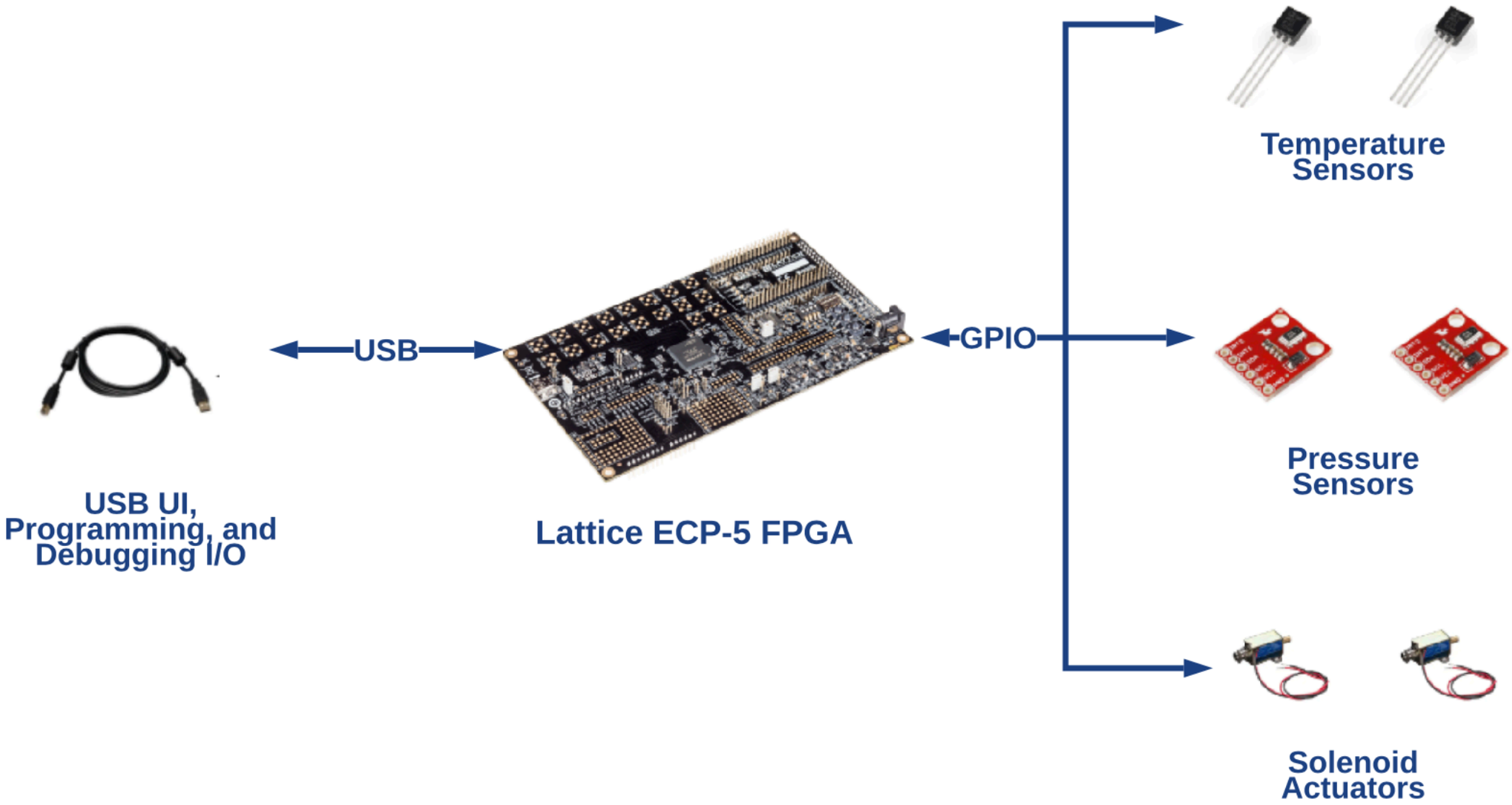
Digital Instrumentation & Control

- 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.

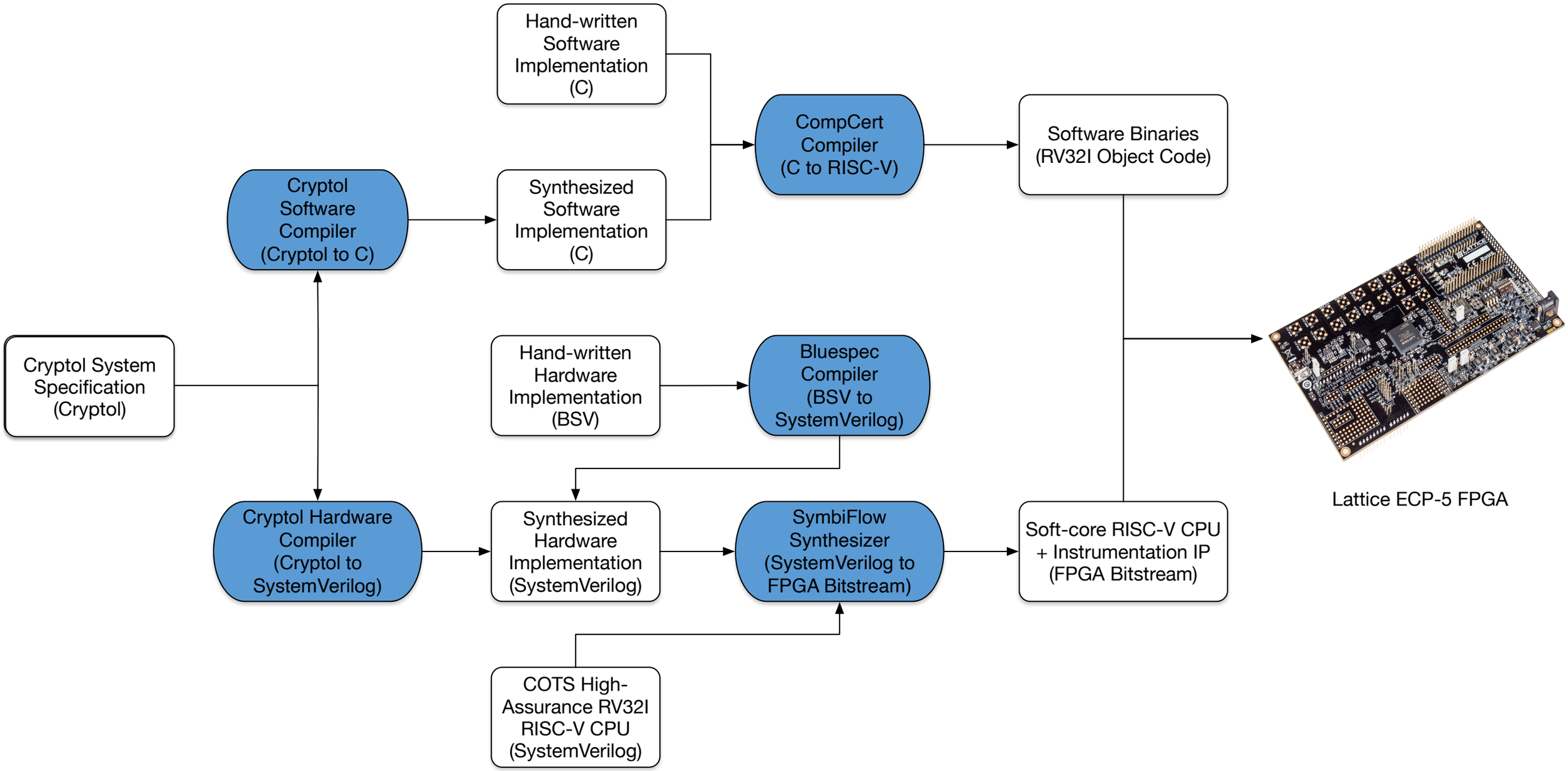
Proposal System Architecture



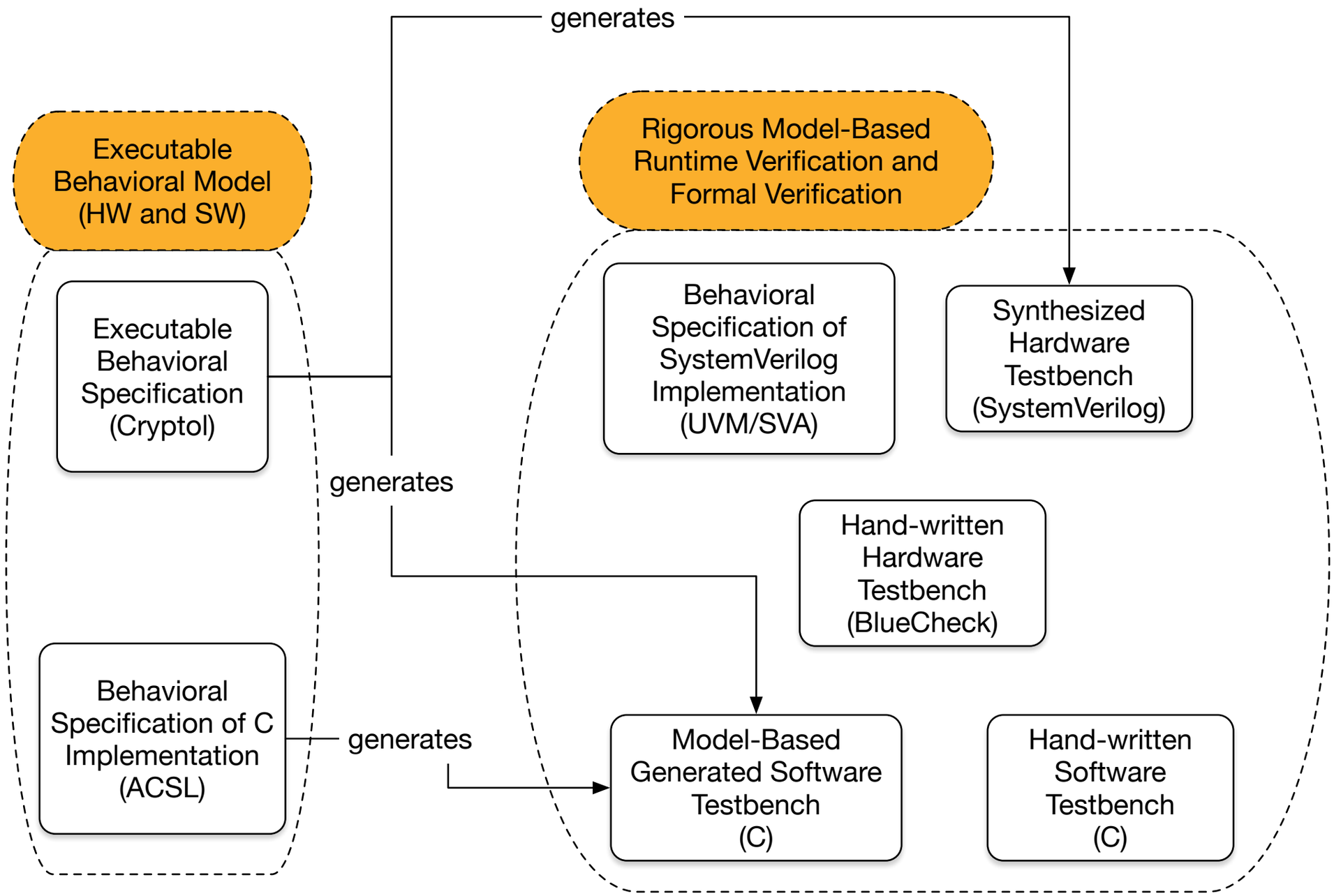
The RTS Hardware Artifacts



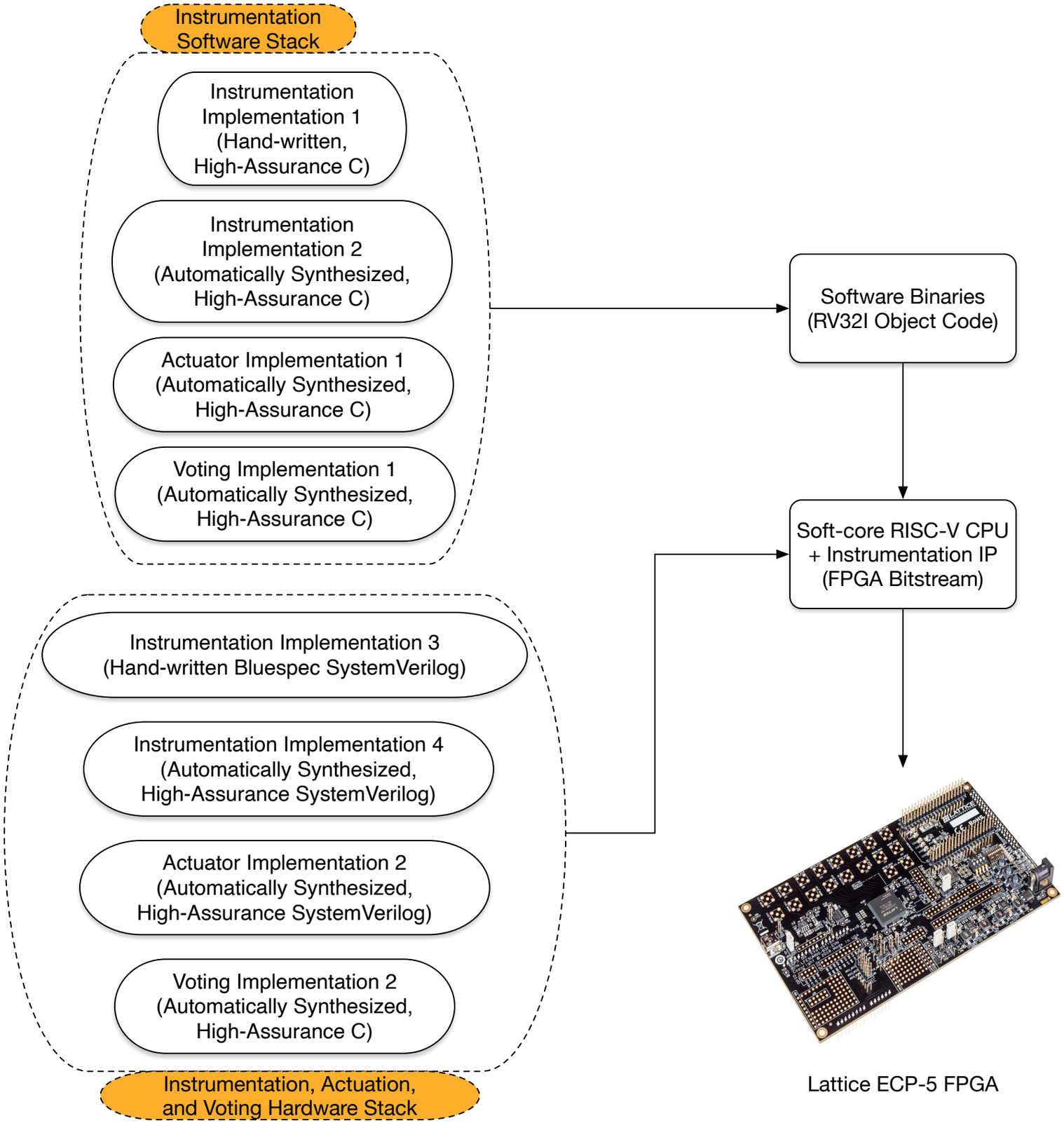
Implementation Artifacts and Relations



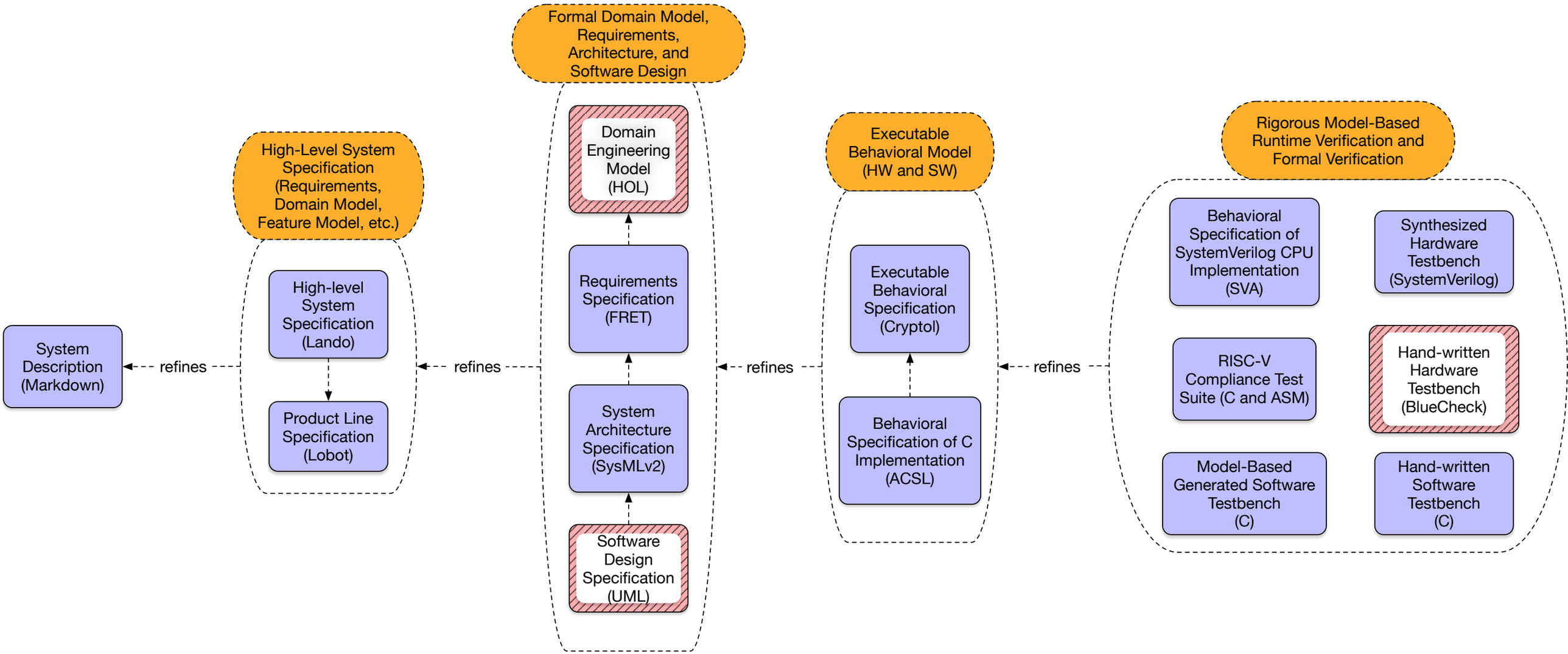
Generation of Assurance Artifacts



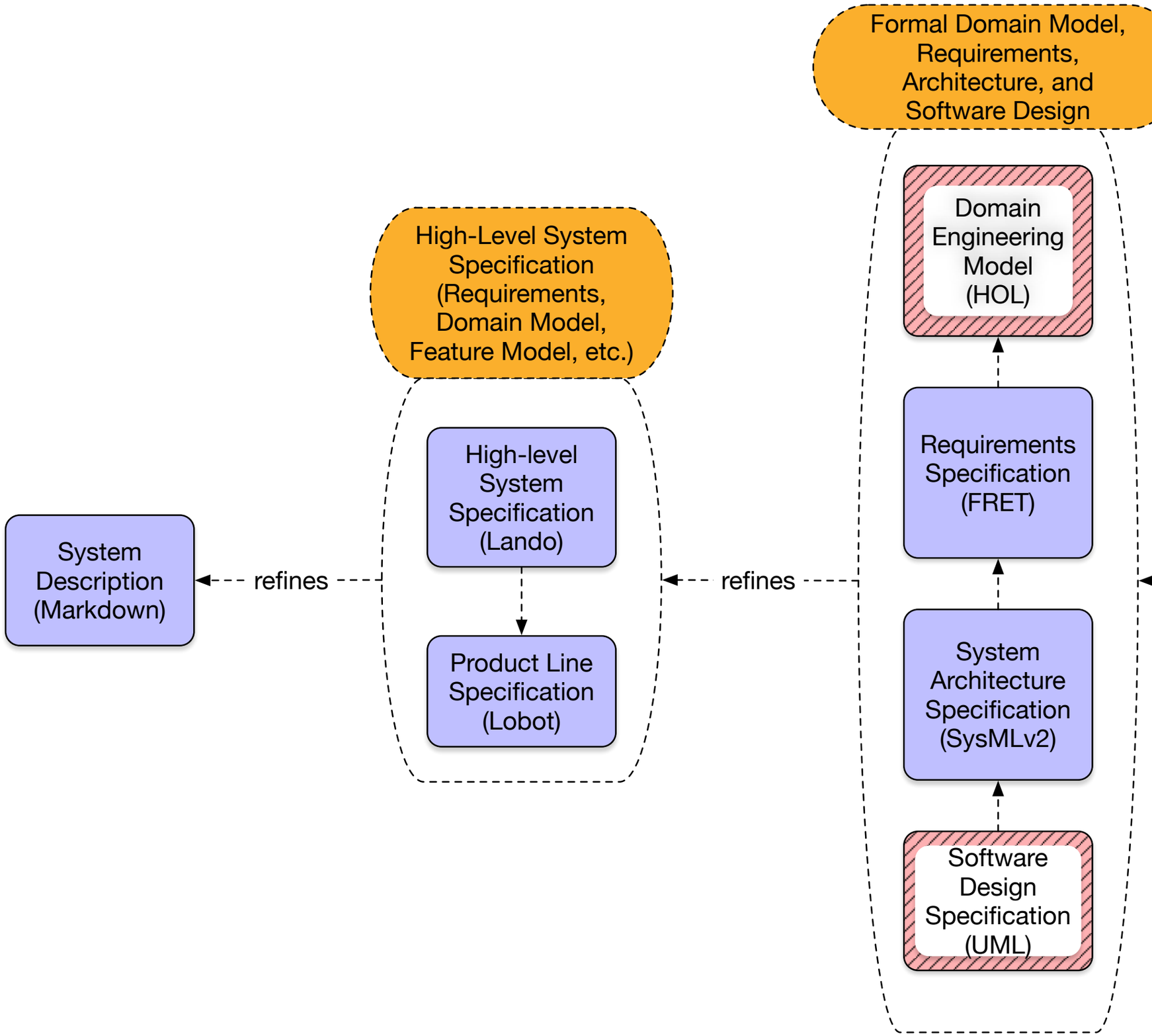
RTS Instrumentation Architecture



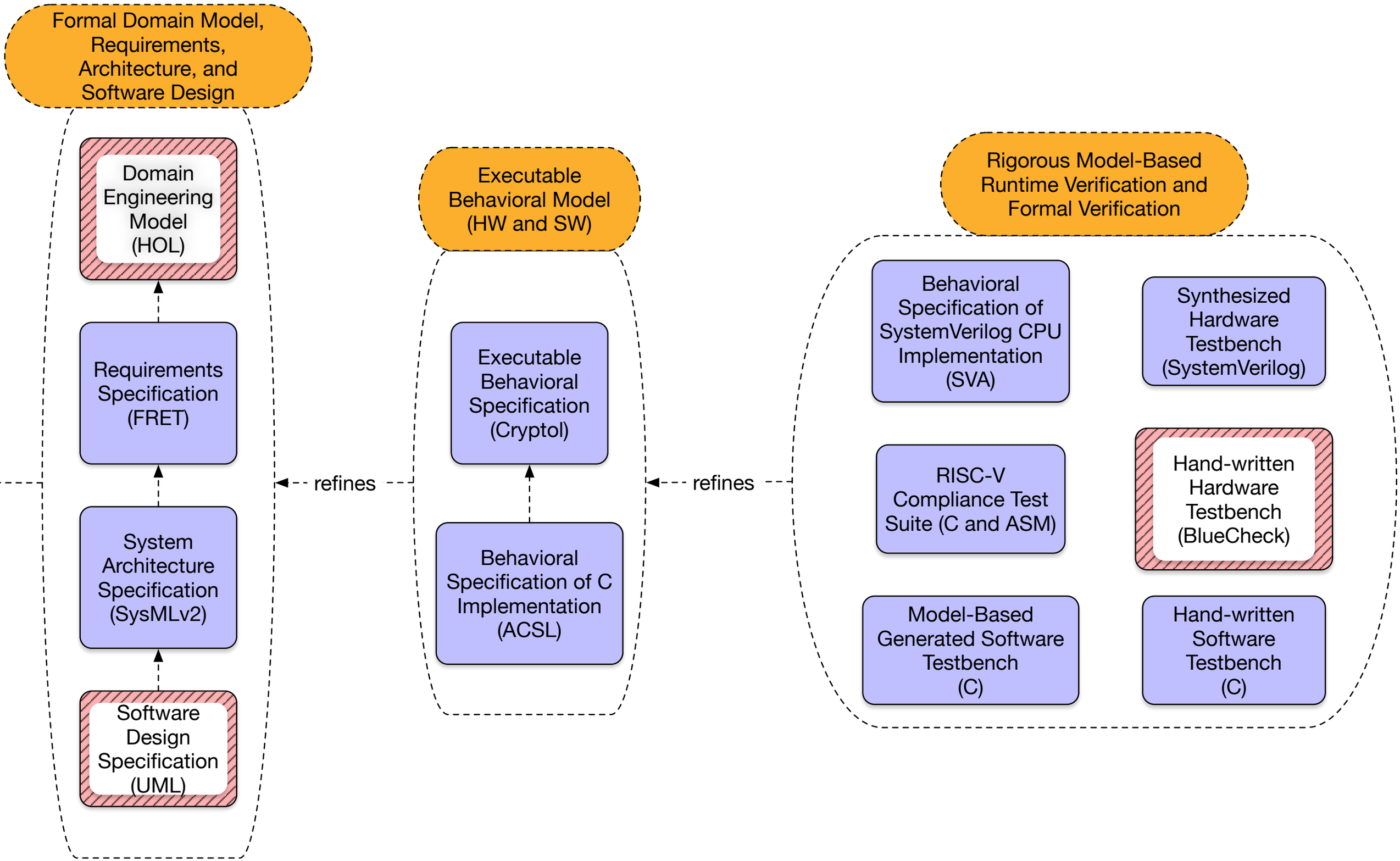
RTS Specification Artifacts



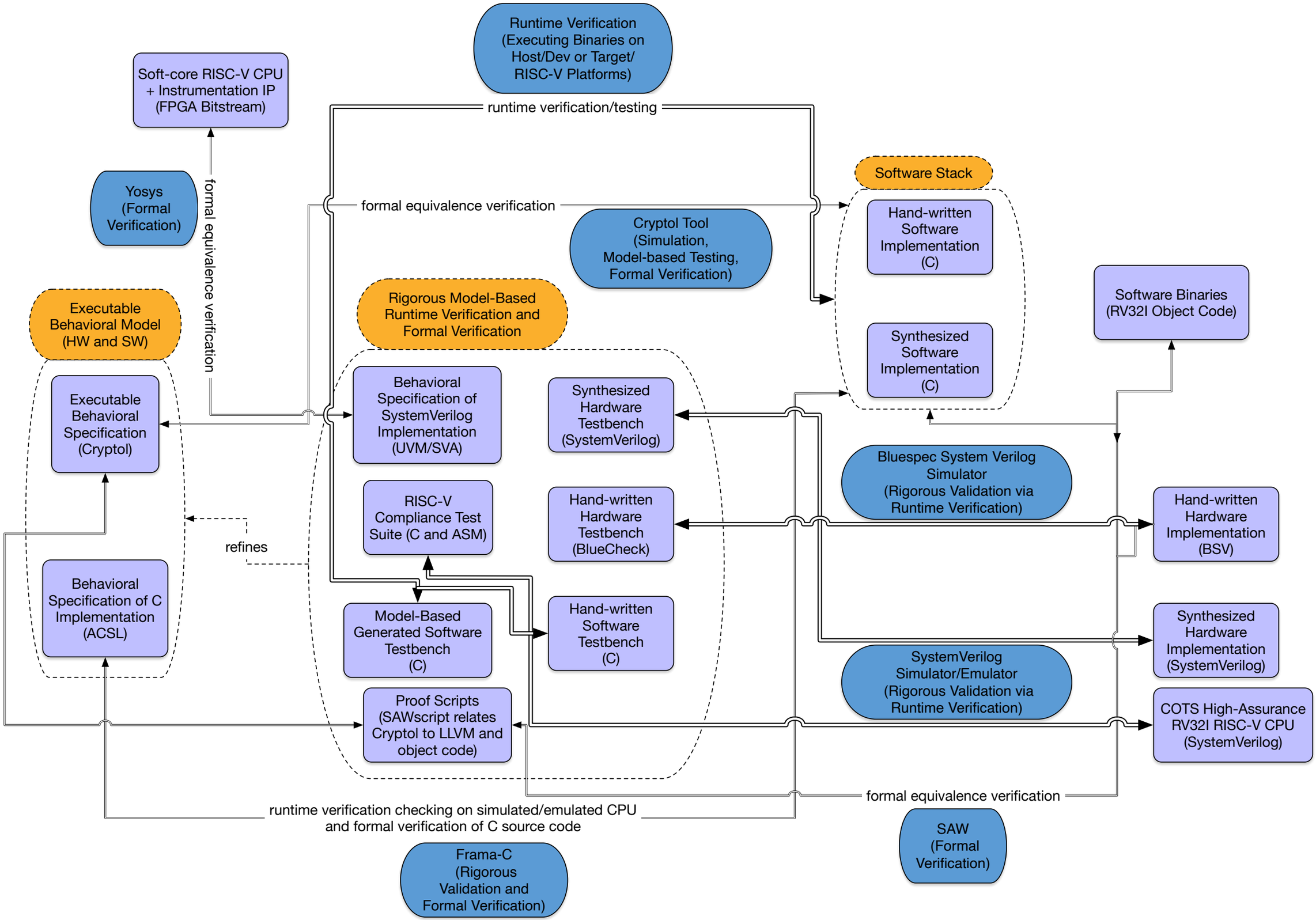
RTS Specification Artifacts



RTS Specification Artifacts



RTS Validation/Testing and Verification Artifacts



IEEE Standard 603-2018
Characteristics to be Demonstrated

- **[CCC]** completeness and consistency of requirements
- **[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)
- **[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)

IEEE Standard 603-2018
Characteristics to be Demonstrated

- **[Actuation Independence]** independence among the two trains of actuation logic (inability for the behavior of one train to interfere or adversely affect the performance of another)
- **[Actuation Correctness]** completion of actuation whenever coincidence logic is satisfied or manual actuation is initiated
- **[Self-Test/Trip Independence]** 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

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 Examples

subsystem Proposal Acronyms (Acronyms)

A list of words formed by combining the initial letters of a multipart name.

// Source: Frama-C website

component ISO ANSI C Specification Language (ACSL)

The ANSI/ISO C Specification Language (ACSL) is a behavioral specification language for C programs.

// Source: https://csrc.nist.gov/glossary/term/Application_Programming_Interface

component Application Programming Interface (API)

A system access point or library function that has a well-defined syntax and is accessible from application programs or user code to provide well-defined functionality.

component Application-Specific Integrated Circuit (ASIC)

Custom-designed and/or custom-manufactured integrated circuits.

Domain Engineering Examples

```
// Events are (seemingly-atomic, from the point of view of an external
// observer) interactions/state-transitions of the system.  The full
// set of specified events characterizes every potential externally
// visible state change that the system can perform.

// External input actions are those that are triggered by external input on UI.
events Demonstrator External Input Actions

Manually Actuate Device

The user manually actuates a device.


Select Operating Mode

The user puts an instrumentation division in or takes a division out of
'maintenance' mode.


Perform Setpoint Adjustment

The user adjusts the setpoint for a particular channel in a particular division
in maintenance mode.
```

Domain Engineering Examples

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

Domain Engineering Examples

scenarios Tool Scenarios

Verify Software
Formally verify that software or firmware programs fulfill their specifications.

Verify Hardware
Formally verify that a hardware design fulfills its specifications.

Formal Equivalence Checking
Formally verify that programs written in different languages (even across the hardware/software boundary) are equivalent.

Symbolic Testing
Improve the assurance of software using symbolic testing.

Backend Solver Libraries
Provide libraries for symbolic formula representation and solver interaction.

System Requirements

- system requirements are atomic semi-formal properties that a system must or should hold
- requirements engineering specifies predicates about an architecture, both of which are expressed using the concepts specified in the domain engineering model
- semi-formal requirements and specified using Lando & SysML, and formal requirements are specified using NASA's FRET tool in HARDENS

System Requirements Example

```
// All requirements that the RTS system must fulfill, as driven by the
// IEEE 603-2018 standards and the NRC RFP.

requirements HARDENS Project High-level Requirements
// The high-level requirements for the project stipulated by the NRC RFP.

NRC Understanding
Provide to the NRC expert technical services in order to develop a
better understanding of how Model-Based Systems Engineering (MBSE)
methods and tools can support regulatory reviews of adequate design
and design assurance.

Identify Regulatory Gaps
Identify any barriers or gaps associated with MBSE in a regulatory
review of Digital Instrumentation and Control Systems for existing
Nuclear Power Plants.
```

System Requirements Example

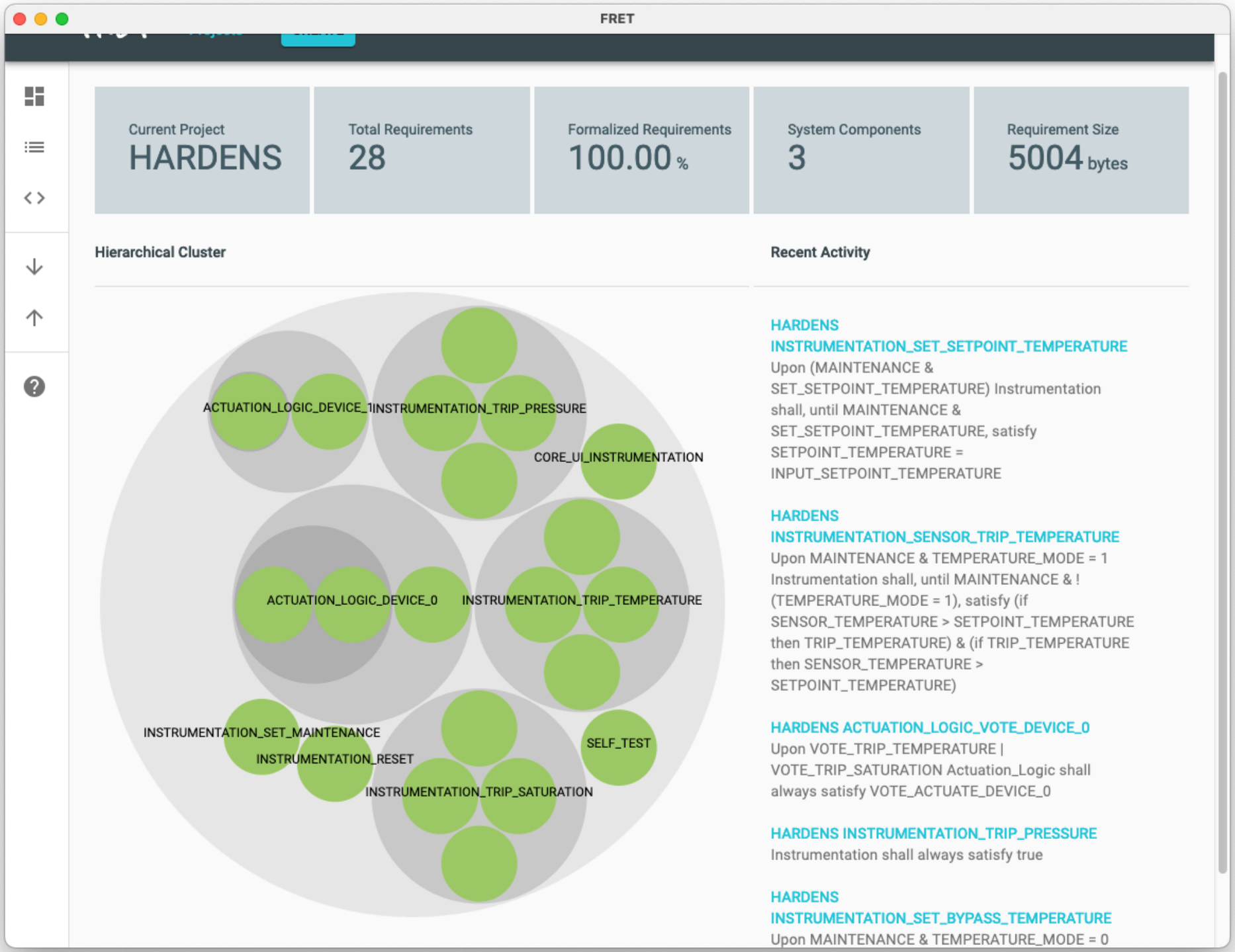
```
requirements NRC Characteristics
// The requirements driven by the IEEE 603-2018 standard for NPP I&C
// systems.

// Both formal and rigorous consistency checks of the requirements
// will be accomplished by using false theorem checks and proofs in
// the Cryptol model and in software and hardware source code;
Requirements Consistency
Requirements must be shown to be consistent.

// A rigorous completeness validation of the requirements will be
// accomplished by demonstrating traceability from the project
// specification (including the RFP text describing the reactor trip
// system) to the formal models of the system and its properties.
Requirements Colloquial Completeness
The system must be shown to fulfill all requirements.
```

Formal Requirements Engineering

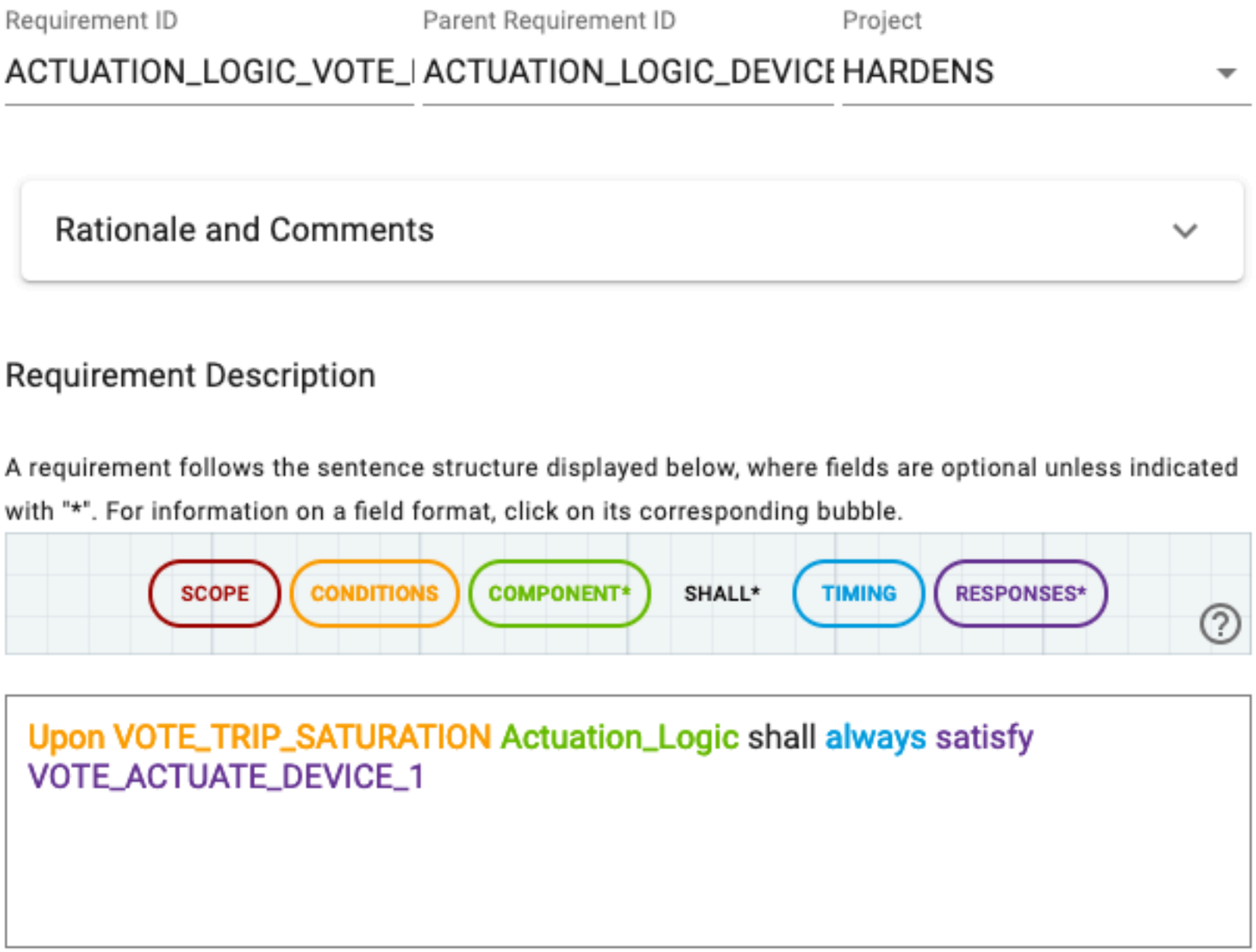
- FRET is a tool for writing, understanding, formalizing, and analyzing requirements.
- Users write requirements in an intuitive, restricted natural language, called FRETISH, with precise, unambiguous meaning.
- For a FRETISH requirement, FRET:
  1. produces natural language and diagrammatic explanations 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.

(*) This entire summary is straight from "*Formal Requirements Elicitation with FRET*" by Giannakopoulou, et al. REFSQ-2020.

HARDENS FRET Dashboard



The FRET Editor

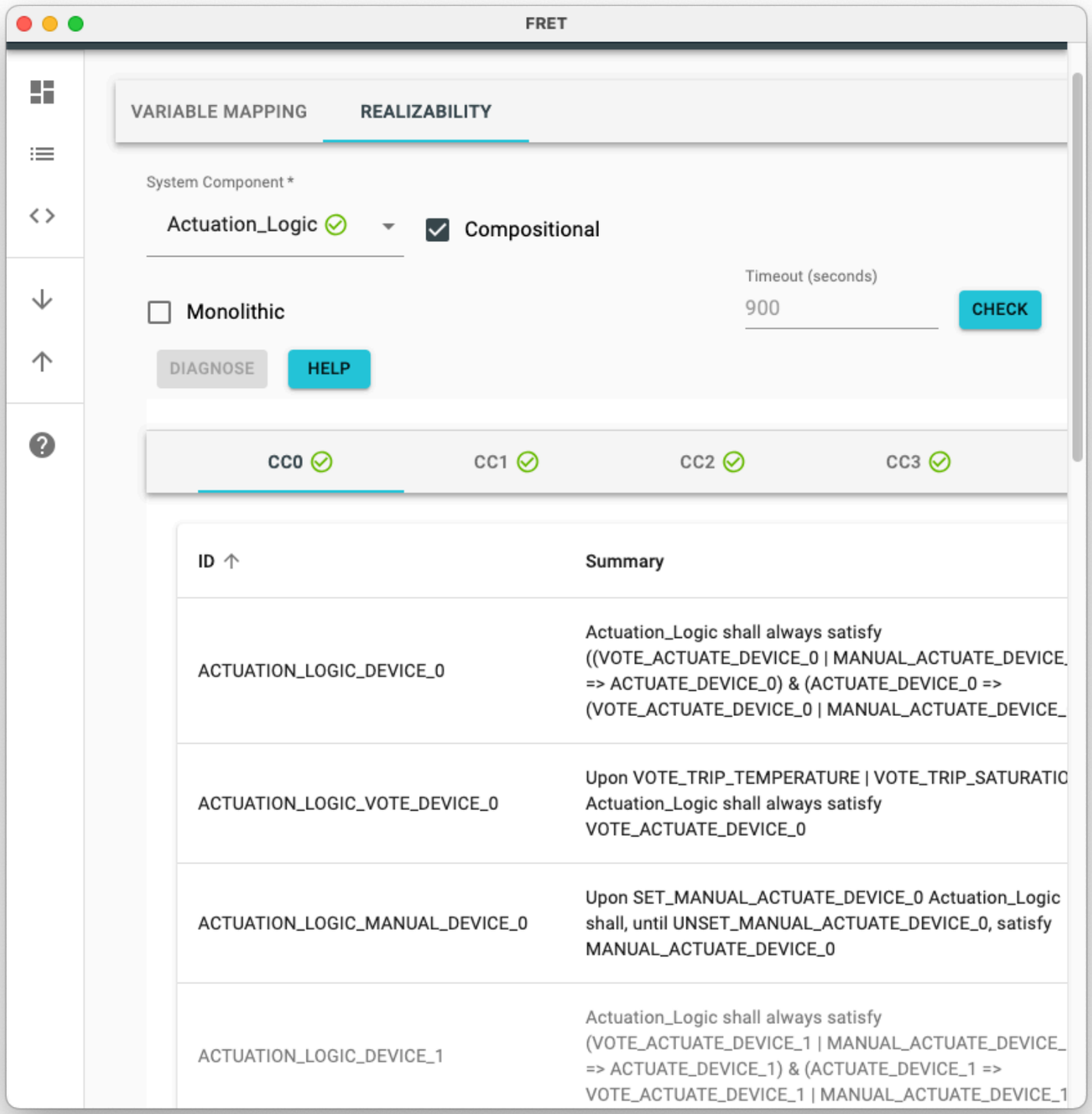


- a web-based, rich, dynamic-feedback editor for individual requirements
- requirements have IDs and are organized in a hierarchy
- provides grammar information and examples during editing
- provides English and diagrammatic explanations to clarify subtle semantic issues

FRET Semantics

- each requirement is formalized in a future time LTL and past time LTL proposition
- LTL propositions can be explored interactively in a simulator that renders as a linear trace
- FRET variables can be associated with refined target model elements such as signals and components

FRET Realizability Checking



- *realizability* (aka *relative consistency* or *implementability*) checks whether or not a specification can be implemented
- FRET generates Lustre contracts that can be checked for realizability with the JKind k-induction and fixpoint generation engine

Systems Engineering Models

- our systems engineering models are written in a variety of modeling languages, historical & modern
  - earlier projects have used a wide variety of languages BON, EBON, UML, Fusion, OBJ, etc.
- the HARDENS systems models are written in Lando, SysMLv2, and soon, some AADL
- these models capture all aspects of the system that are “appropriately” modeled at the system level
  - domain model, requirements, stakeholders, static and dynamic component/subsystem/system models, product line engineering/feature model and its variants, data and physical architecture

System Engineering Examples

```
// Scenarios are sequences of events.  Scenarios document normal and
// abnormal traces of system execution.

// Test scenarios are scenarios that validate a system conforms to its
// requirements through runtime verification (testing).  Each scenario
// is refined to a (possibly parametrized) runtime verification
// property.  If a testbench is complete, then every path of a
// system's state machine should be covered by the its set of scenarios.

scenarios Self-Test Scenarios

Normal Self-Test Behavior 1a - Trip on Mock High Pressure Reading from that Pressure Sensor
The user selects 'maintenance' for an instrumentation division, the
division's pressure channel is set to 'normal' mode, the pressure
setpoint is set to a value v, the user simulates a pressure input to
that division exceeding v, the division generates a pressure trip.

Normal Self-Test Behavior 1b - Trip on Environmental High Pressure Reading from that Pressure
Sensor
The user selects 'maintenance' for an instrumentation division, the
division's pressure channel is set to 'normal' mode, the pressure
setpoint is set to a value v, the division reads a pressure sensor
value division exceeding v, the division generates a pressure trip.
```

System Engineering Examples

```
package id RTS 'Reactor Trip System' {
  private import 'Semantic Properties'::*;
  import 'Project Glossary'::*;
  import 'RTS Viewpoints and Views'::*;
  import 'RTS Architecture'::*;


  package id Architecture 'RTS Architecture';
  alias Arch for Architecture;
  package id Hardware 'RTS Hardware Artifacts';
  alias HW for Hardware;
  package id Artifacts 'RTS Implementation Artifacts';
  package id Requirements 'RTS Requirements';
  package id Properties 'RTS Properties';
  alias Props for Properties;
  package id Characteristics 'IEEE Std 603-2018 Characteristics';
  comment TopLevelPackages about Architecture, Hardware, Properties, Characteristics
  /* These are the core top-level subsystems characterizing HARDEN work. */
}
```

- the top-level architectural breakdown of the RTS system in SysMLv2

```
package id Glossary 'Project Glossary' {
  // @design Eliminate all redundancy with concepts in KerML or SysML domain
  // libraries.
  private import ScalarValues::*;
  private import KerML::*;

  part def BlueCheck;
  /** A formal, state-based specification language that focuses on the
      specification of the interfaces of discrete modules in a system, and
      often times includes model-based specification constructs to improve
      usability and expressivity. */
  abstract item id BISL 'Behavioral Interface Specification Language';
  abstract part def Computer;
abstract part def Coq;
abstract part def Cryptol;
abstract item def DevSecOps;
abstract item def id DIANC 'Digital Instrumentation and Control Systems';
/** The NASA Formal Requirements Elicitation Tool is used to make writing,
    understanding, and debugging formal requirements natural and
    intuitive. */
part def id FRET 'Formal Requirements Elicitation Tool';
/** An Instruction Set Architecture, or ISA for short, is the set of
    instructions that a given kind of CPU can understand. Example ISAs
    include x86, x64, MIPS, RISC, RISC-V, AVR, etc. */
attribute def id ISA 'Instruction Set Architecture';
```

- the domain engineering model is expressed in SysMLv2 using parts and items

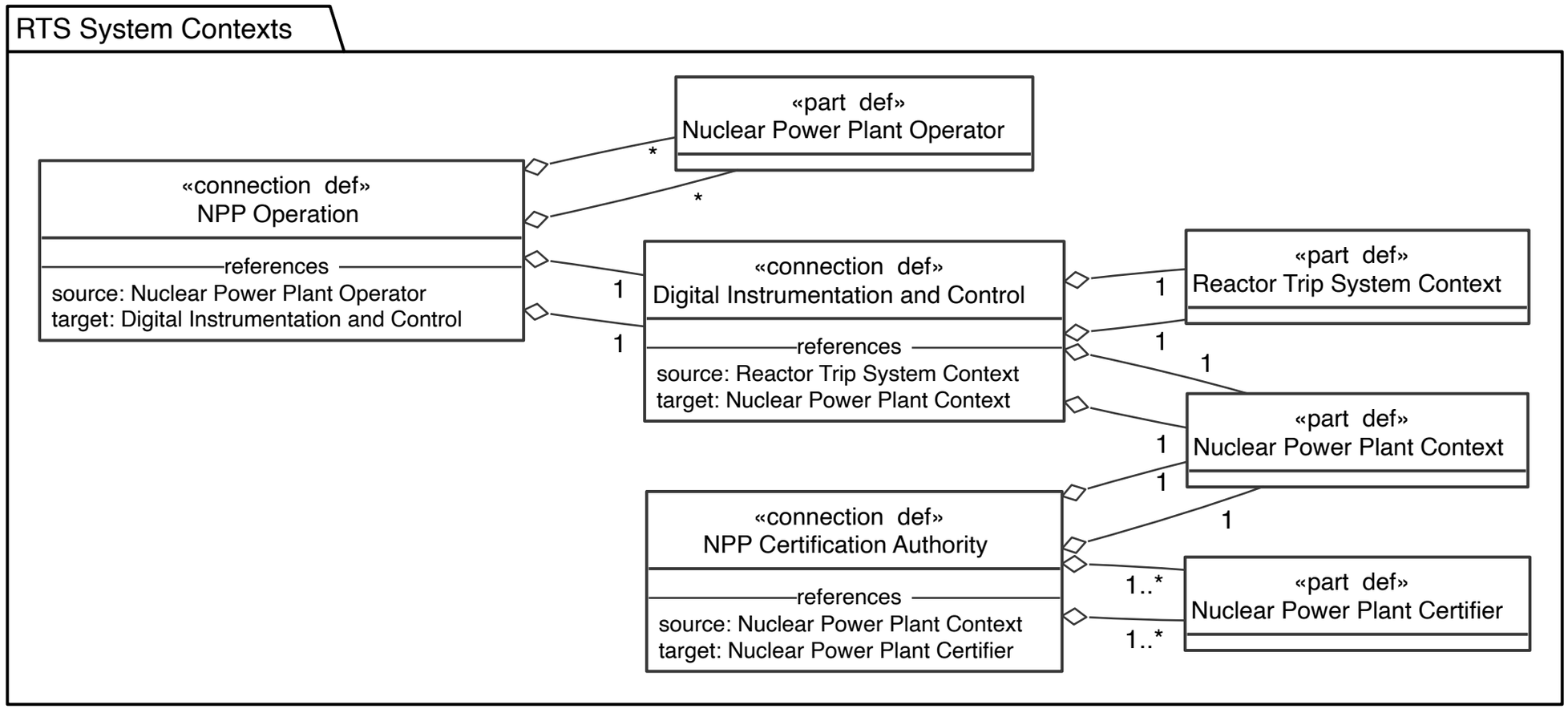
```
/**
 * The physical hardware components that are a part of the HARDENS RTS
 * demonstrator.
 */
package 'RTS Hardware Artifacts' {
  private import 'Project Glossary'::*;
  //import Architecture::RTS_System_Arch::Hardware::*;
  private import ScalarValues::*;

  part def 'SERDES Test SMA Connector' :> Connector;
  part def 'Parallel Config Header' :> Header;
  part def 'Versa Expansion Connector' :> Connector;
  part def 'SPI Flag Configuration Memory' :> Memory;
  part def 'CFG Switch' :> Switch;
part def 'Input Switch' :> Switch;
part def 'Output LED' :> LED;
part def 'Input Push Button' :> Button;
part def '12 V DC Power Input' :> Power;
part def 'GPIO Headers' :> Header, GPIO;
part def 'PMOD/GPIO Header' :> Header, PMOD, GPIO;
part def 'Microphone Board/GPIO Header' :> Header;
part def 'ECP5-5G Device' :> FPGA;
// @todo Ensure that JTAG is, in fact, USB.
part def 'JTAG Interface' :> JTAG, USB;
part def 'Mini USB Programming' :> USB;
part def id DevBoard 'Lattice ECP-5 FPGA Development Board' :> PCB {
  part J9_J26 : 'SERDES Test SMA Connector'[16] subsets components;
```

- the SysML model includes both the platform architecture and the RTS's physical architecture
- e.g., the FPGA board and its parts are specified at left

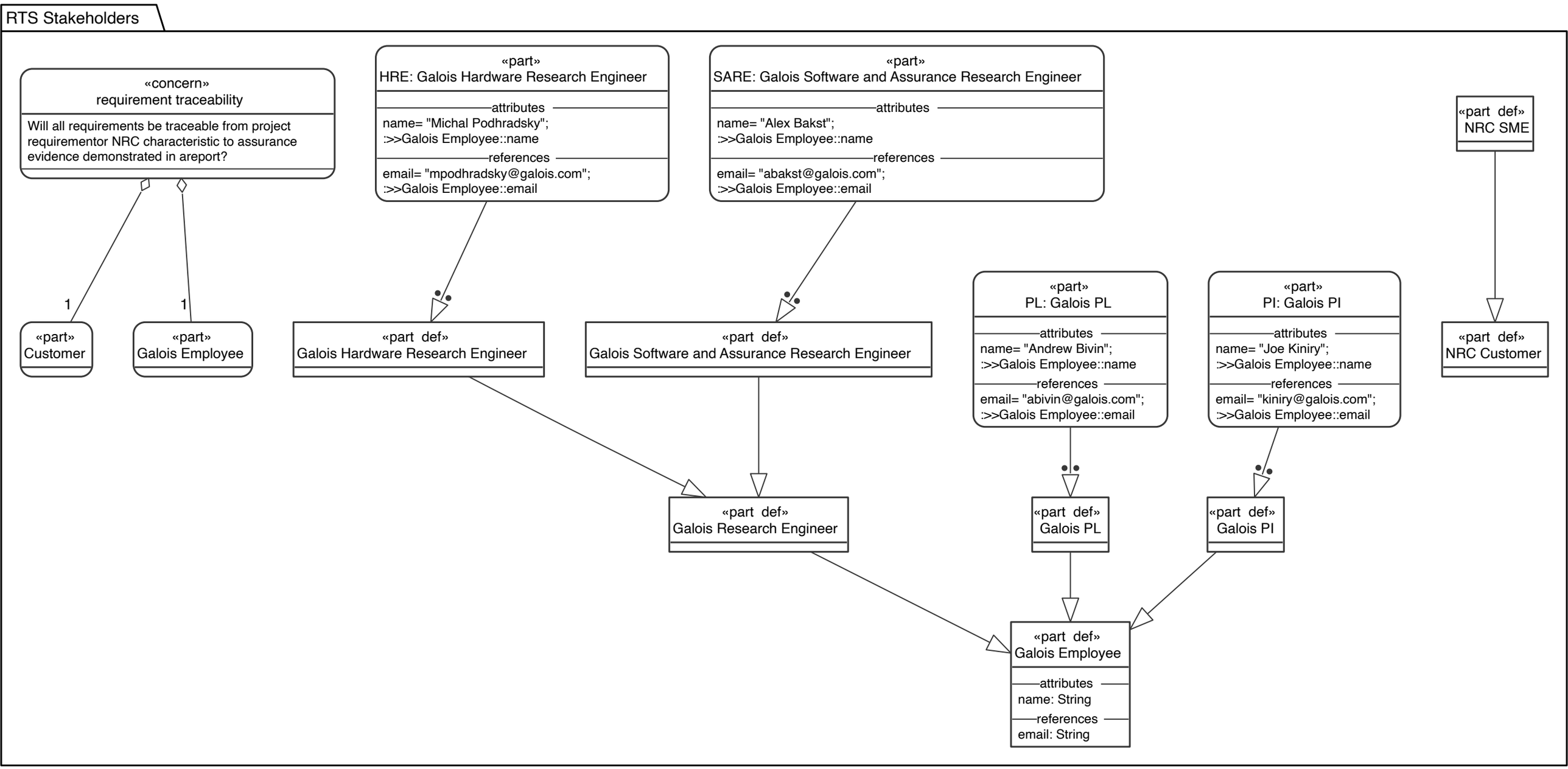
```
-- title: Reactor Trip System high-assurance demonstrator.
-- project: High Assurance Rigorous Digital Engineering for Nuclear Safety (HARDENS)
-- copyright (C) 2021 Galois
-- author: Joe Kiniry <kiniry@galois.com>

-- Our development platforms for running the RTS demonstrator in a
-- fully virtualized (Twin) mode.  If we choose to target a real RV32,
-- then we will be running on the bare metal.

type virtualized_platform_runtime =
  { Posix, RV32_bare_metal, None }

-- The developer boards we have to choose from.  We are using the
-- ECP-5 5G 85F variant of the Lattice Semiconductor dev board, and if
-- we choose to put the demonstrator on a real RV32, we will likely
-- use the Vega board.

type dev_board =
  { Virtual, LFE5UM5G_85F_EVN, RV32M1_VEGA, None }

-- The ECP-5 FPGA comes in several flavors.  We are using the 5G
-- variant for this project.  Other variants should be able to use the
-- exact same build chain.

type fpga =
  { ECP5, ECP5_5G }
```

- the SysML model includes both the platform architecture and the RTS's physical architecture
- e.g., the FPGA board and its parts are specified at left

RTS System Contexts



RTS Stakeholders



Executable Denotational Model

- semantic formal models that we write are either denotational or operational
  - denotational formal models describe structure and meaning, but are typically not executable
  - operational formal models are executable and focus on behavior, rather than structure
  - both kinds of models are often used to generate or reason about other artifacts, such as models, implementations, and assurance artifacts
- HARDENS model is both denotational (it describes structure and meaning) and executable (it describes behavior and can be executed & tested)

Cryptol

- Cryptol is a functional, strongly-typed, Domain Specific Language for specifying the structure and behavior of bit-level algorithms
- Cryptol was originally created for the NSA to specify and reason about cryptographic algorithms
- the Cryptol tool permits the type checking, execution, (quick check-style) runtime validation, and formal verification of Cryptol models
- other tools permit the compilation of Cryptol models to assurance artifacts (software and hardware runtime verification test benches and formal verification benches) and implementations (in high-performance, formally verified C, LLVM, JVM, and SystemVerilog)

Cryptol Examples

```
// HARDENS Reactor Trip System (RTS) Actuator Unit
// A formal model of RTS Actuator behavior written in the Cryptol DSL.
// @author Alex Bakst <abakst@galois.com>
// @created November, 2021
// @refines HARDENS.sysml
// @refines RTS.lando
// @refines RTS_Requirements.json

module RTS::Actuator where

type Actuation = Bit
type Mode      = Bit

/** @requirements ACTUATION_LOGIC_MANUAL_DEVICE_{0,1} satisfied by definition */
type Actuator =
  { input: Actuation
  , manualActuatorInput: Actuation }

SetInput: Actuation -> Actuator -> Actuator
SetInput on actuator = {actuator | input = on }

SetManual: Actuation -> Actuator -> Actuator
SetManual on actuator = {actuator | manualActuatorInput = on}

ActuateActuator : [2]Actuation -> Actuation
ActuateActuator inputs = (inputs @ (0:[1])) || (inputs @ (1:[1]))
```

```
type InstrumentationUnit =
  { setpoints: [NChannels][32]
  , reading: [NChannels][32]
  , mode: [NChannels]Mode
  , sensor_trip: [NChannels]
  , output_trip: [NChannels][8]
  , maintenance: Bit
  }

Initial: InstrumentationUnit
Initial =
  { setpoints   = zero
  , reading     = zero
  , mode        = zero
  , sensor_trip = zero
  , output_trip = repeat zero
  , maintenance = ~zero
  }

Step: Input -> Command -> InstrumentationUnit -> InstrumentationUnit
// @refines sensor input ports
Handle_Input: Input -> InstrumentationUnit -> InstrumentationUnit
// @refines mode, tripmode, setpoint input port attributes
Handle_Command: Command -> InstrumentationUnit -> InstrumentationUnit
Get_Reading: InstrumentationUnit -> [NChannels][32]
In_Maintenance: InstrumentationUnit -> Bit
Set_Maintenance: Bit -> InstrumentationUnit -> InstrumentationUnit
Set_Mode: Channel -> Mode -> InstrumentationUnit ->
InstrumentationUnit
In_Mode: Channel -> Mode -> InstrumentationUnit -> Bit
Get_Setpoint: Channel -> InstrumentationUnit -> [32]
Set_Setpoint: Channel -> [32] -> InstrumentationUnit ->
InstrumentationUnit
Get_Tripped: InstrumentationUnit -> [NChannels][8]
Is_Tripped: Channel -> InstrumentationUnit -> Bit
Is_Ch_Tripped : Mode -> Bit -> Bit
```

```
Step_Trip_Signals:
  InstrumentationUnit ->
  InstrumentationUnit
Saturation : [32] -> [32] -> [32]
PressureTable : [32] -> [32]
Generate_Sensor_Trips : [NChannels][32] ->
  [NChannels][32] -> [NChannels]
Trip: [NChannels][32] -> [NChannels][32] -> Channel -> Bit

private
  /** @requirements
        INSTRUMENTATION_RESET
  */
  property instrumentation_reset =
    In_Maintenance Initial
    /\ In_Mode P Bypass Initial
    /\ In_Mode T Bypass Initial
    /\ In_Mode S Bypass Initial

  /** @requirements
        INSTRUMENTATION_TRIP_PRESSURE
  */
  property instrumentation_trip_pressure (inp: Input)
    (instr: InstrumentationUnit) =
      In_Mode P Manual instr
    \/ (In_Mode P Operate instr /\ inp @ P > Get_Setpoint P instr')
    /*
------------------------------------------------------------------ */
==
    (Is_Tripped P instr')
      where instr' = Handle_Input inp instr >>>> Step_Trip_Signals
```

Behavioral Interface Specification

- a Behavioral Interface Specification Language (BISL) is a model-based, abstract state-based language for specifying the behavior of things
- BISLs exist for software, hardware, and systems
  - e.g., ACSL, JML, Larch, and the contract languages for Eiffel, .NET, SPARK-Ada, etc.
- the de facto BISL for the C programming language is the ANSI C Specification Language (ACSL)
- ACSL permits one to specify and reason about formal models of low-level C code using datatype invariants, pre- and postconditions (contracts) for functions, and much more

Frama-C

- the Frama-C tool suite permits one to type check, compile, analyze, and reason about ACSL-annotated C programs
  - the tools suite includes dozens of plugins to support different kinds of analysis
  - Frama-C can animate model-based specifications, compile annotations to runtime assertion checks and test benches, and formally reason about C implementations
  - Frama-C's formal verification tools use automated solvers (SAT and SMT) and logical frameworks (Coq) as their backends

Example ACSL Specifications

```
/*@requires \valid(&trips[0.. NINSTR -1]);
  @assigns \nothing;
  @ensures (\result != 0) <==> Coincidence_2_4(trips);
*/
uint8_t Coincidence_2_4(uint8_t trips[4]);

/*@requires \valid(&trips[0.. NTRIP - 1][0.. NINSTR - 1]);
  @requires \valid(trips + (0.. NTRIP-1));
  @assigns \nothing;
  @ensures (\result != 0) <==>
  Actuate_D0(&trips[T][0], &trips[P][0], &trips[S][0], old != 0);
*/
uint8_t Actuate_D0(uint8_t trips[3][4], uint8_t old);

/*@requires \valid(&trips[0.. NTRIP-1][0.. NINSTR-1]);
  @requires \valid(trips + (0.. NTRIP-1));
  @assigns \nothing;
  @ensures (\result != 0) <==>
  Actuate_D1(&trips[T][0], &trips[P][0], &trips[S][0], old != 0);
*/
uint8_t Actuate_D1(uint8_t trips[3][4], uint8_t old);

struct actuation_logic {
    uint8_t vote_actuate[NDEV];
    uint8_t manual_actuate[NDEV];
};
```

```
extern struct actuation_logic actuation_logic[2];

/* The main logic of the actuation unit */

/*@requires \valid(state);
  @requires logic_no <= 1;
  @assigns state->manual_actuate[0.. NDEV-1];
  @assigns state->vote_actuate[0.. NDEV-1];
  @assigns core.test.actuation_old_vote;
  @assigns core.test.test_actuation_unit_done[logic_no];
*/
int actuation_unit_step(uint8_t logic_no,
                        struct actuation_logic *state);
```

Hardware Design and Verification

- CPU is a 32 bit RISC-V “NERV” CPU developed by YosysHQ
  - single issue, user mode, shallow pipeline, designed for clarity, elegance, and verifiability, not performance
- our SoC is either a 1 core or 3 core SoC with exactly the architecture driven by the requirements
  - consequently, the architecture has a novel shape
- I/O units and core critical computation are hand-written and/or automatically synthesized
- assurance case includes both formal and runtime verification
  - includes digital twins of both CPU and SoC
  - run every hand-written scenario refined to property-based test, and every parametrized test generated from model-based specification, on every digital twin and the hardware
  - formal assurance that NERV CPU exactly implements RV32 instruction set—no more, no less

Reasoning about Models and Implementations

Kinds of Reasoning

- one can reason about a thing (a model, a software implementation in source code, a binary, a hardware design at various abstraction levels, etc.) in many different ways
  - model check states
  - attempt to satisfy a preposition
  - interactively prove theorems
- our tools automatically generate theorems about artifacts by statically translating the artifacts into their semantic representations,
- embed that translation in a background theory of the universe of discourse (using, in part, the domain engineering model),
- and then state and try to prove theorems about the relationship between artifacts

Equivalence and Refinement

- The main relationships that we specify and reason about in our RDE are equivalence and refinement.
  - *Equivalence* means that two artifacts are “equal” under some formal equivalence relation defined in terms of the underlying semantics used to describe the artifacts.
    - Cryptol spec ≈ C program
    - C program ≈ Java program
    - LLVM bitcode ≈ Verilog spec
  - Equivalence facilities reasoning about correctness across models, languages, optimizations, evolution.
- *Refinement* means that one artifacts “refines” another under some formal refinement relation defined in terms of the underlying semantics used to describe the artifacts.
  - C ⫤ ACSL ⫤ Cryptol ⫤ FRET ⫤ SysMLv2 ⫤ Lando ⫤ Markdown
  - SystemVerilog ⫤ SVA ⫤ Cryptol ⫤ FRET ⫤ SysMLv2 ⫤ etc.

Traceability as Refinement

- traceability in our formal method is effected through refinement and is *implicit* and *discoverable*
- when defined properly, refinement relations enable…
  - refinement checking
    - “Is A a refinement of B, and if so, how, and if not, why?”
  - model lifting
    - “What is a formal model of this artifact?”
  - model or code generation
    - “What is an artifact that refines this artifact?”
- explicit annotations for traceability are only used to connect development artifacts to process
  - e.g., annotation of a design decision on a specification that links to a GitLab issue that provide evidence and provenance and connects to a specific signed pull request and release tags

Assurance Case

- for high-assurance systems created with rigorous digital engineering, we must demonstrate that…
  - the specification which describes the system is the right specification (it is consistent, realizable, and has all of the properties we demand—aka spec validation)
  - all specifications relate to each other as specified to build the compositional argument of the system's properties (equivalent specs are equivalent, specs that are refinements are refinements, etc.)
  - the implemented system conforms to its specifications (it is shaped like, and operates as, promised—no more, no less—in precisely characterized environments)
- for HARDENS, in order to build this compositional assurance case we use the Lando type checker, the SysMLv2 type checker, FRET, Cryptol, SAW, Frama-C, and Yosys

Dynamic Assurance Case

- dynamic assurance classically means “run hand-written tests on the real platform”
- in the DevSecOps universe, it typically means “run hand-written and some automated tests on a test platform that is meant to duplicate the real one”
- in the MBE universe, it means “run tests that are automatically generated from models on…”
- in the Digital Engineering universe, it means “run tests on Digital Twins and the real platform…”
- in the RDE universe, it means “run rigorously created (>>99% automatically generated and a bit of hand-written) parametrized property-based tests derived from theorems about the architecture and assurance goals on all digital twins and the real test and deployment platforms”

Dynamic Assurance for HARDENS

- all Markdown requirements refine to SysMLv2 requirements refine to Cryptol properties (first-order theorems about the behavioral model) refine to property-based tests about the hardware and software
- refinement from formal Cryptol properties results in…
  - theorems stated in ACSL about the software (at the unit, subsystem, and system level, & software device drivers),
  - theorems stated using Bluecheck, SVA, and UVM about the hardware (CPU & SoC & hardware device drivers), and
  - a software testbench is automatically generated from the Cryptol model by translating every theorem into a software property that is runtime or formally verified

Dynamic Assurance for HARDENS

- all tests are run on…
  - the executable behavioral Cryptol model,
  - a POSIX emulation of the RTS hardware,
  - a hardware simulator of the CPU to verify that the CPU conformed to the RISC-V ISA spec and the (single-core and multi-core) SoC to verify SoC-level properties about devices, I/O, concurrency, and
  - the deployment hardware (the Lattice FPGA)
- tests are run with RTS self-test enabled and disabled
- the entire and test system is specified with a feature model and we aim to runtime verify all realizable models (e.g., identical results across three C compilers targeting three ISAs)

Model Validation

- to ensure that our formal models are valid…
  - we use FRET to formally analyze the system requirements for consistency, completeness, and realizability
    - FRET's realizability checker, which verifies that for a set of input values satisfying the requirements, a set of output values exists that satisfy the requirements
    - dually, the tool identifies inputs for which no output is feasible, which helps narrow down subsets of requirements that are inconsistent.
  - we use Cryptol to demonstrate (either test automatic property-based testing or, in the main, formally prove) that all theorems about the Cryptol behavioral model hold
  - we use Frama-C to demonstrate that all theorems about the software model are consistent by attempting to prove “bottom”
  - we use Yosys to demonstrate through formal verification that all theorems about the hardware hold by attempting to prove “bottom”—the implementation that crashes—satisfies the spec

Hardware Formal Verification

- our hardware has three kinds of components…
  - hardware device drivers for I/O
  - a 32-bit RISC-V CPU (NERV CPU from YosysHQ)
  - a single core and three core NERV-based SoC
- thus we must verify…
  - each device driver conforms to its formal model
  - the CPU exactly conforms to a specific flavor of RISC-V
  - the SOC behaves exactly as we specify

Hardware Formal Verification

- yosys is an open source tool suite that includes tools to synthesize bitstreams to some FGPAs and a formal verification tools that can perform equivalence checking & property-based verification
  - thus we use yosys to re-run the existing ISA formal verification on the CPU (as a kind of formal regression analysis), and
  - we runtime verify all SoC properties on the synthesized SoC (in simulation and on the FPGA)
- we use our correct-by-construction tools to automatically synthesize hardware implementations of some of the architecture from the Cryptol model

Software Formal Verification

- to formally verify the RTS software…
  - we use our correct-by-construction tools to generate some software components directly from the Cryptol model
    - then we use the SAW tool to further guarantee that the generated components are, in fact, correct through formal verification
  - we use formal static verification of the C source code
    - formally prove functional correctness relative to the Cryptol model, and
    - guarantee the absence of any runtime errors (e.g., due to array out-of-bounds errors and null pointers)

Digital Engineering Artifacts

- our RDE model includes system models that capture everything from domain engineering model and requirements (Lando and SysMLv2) to low-level architectural structure and properties (SysMLv2, Cryptol, ACSL, and SVA)
- we must demonstrate that our several digital twins are correct and are refinements of each other
- we must understand and quantify their fidelity, assurance level, and performance

Digital Twins

- the RTS system includes several digital twins
  - the executable behavioral Cryptol model,
  - a POSIX emulation of the RTS hardware,
  - a hardware simulator of the RISC-V CPU, and
  - the (single-core and multi-core) SoC
- all system properties must be/are valid for all twins
- there are several other twins we would like to add
  - a machine emulator model (QEMU) of the platform
  - a Bluespec event-based simulator of the SoC
  - a SystemC-based platform emulator such as Imperas's
  - an alternative software-based hardware simulator such as Metrics's cloud/LLVM-based simulator

Digital Threads

- a digital thread is a chain of properties that run through refinements of digital twins and the deployment platform
- *traceability via refinement* guarantees that digital threads are *sound* and *complete*
  - from every relevant domain-specific concept in the domain engineering model to…
  - every system characteristic and requirement to…
    - every structural element and formal property of every model and digital twin…
    - to every structural element and formal property of the deployment hardware
- threads are *sound* when their elements (theorems/properties) are shown to hold for every twin and target in the refinement chain
- threads are *complete* when every abstract element is shown to hold for every twin and target in the refinement chain

Reflections on SysMLv2

- our SysMLv2 model does not yet include…
  - the RTS feature model specified as a variability model
  - all formal model properties specified as constraints
  - a complete set of occurrences, time slices, and snapshots for all normal and exceptional behaviors
  - a characterization of the full assurance case of our product line
- the syntax is adequate but sometimes offensive to programming language experts
- the Eclipse-based pilot implementation is adequate
- language evolution has been manageable
- error-reporting needs improvement
- PlantUML-based rendering is not acceptable for moderately-complex models; we starting to work with Tom Sawyer Software to mitigate
- specifying formal properties is more awkward than necessary

Next Steps

- there are enormous opportunities in the design and implementation of SysMLv2 to facilitate RDE
- we need a formal, denotational, executable model of SysMLv2 to realize SysMLv2 digital twins directly and to define formal refinement relations to other formal models
  - we have a draft of a partial HOL model in PVS
- we need a means by which to validate and verify SysMLv2 implementations are conformant to its formal semantics
- we need an IDE that facilitates automatic refinement checking and refinement-based model lifting & generation
  - we need to define a refinement between SysMLv2 and AADL
- we need high-quality, interactive graphical model editing, refactoring, and rendering
  - preferably defined via an equivalence relation between the textual and graphical semantics, and then extracted to an implementation

More Information

- Galois https://galois.com/ & https://lifeatgalois.com/
- NRC https://www.nrc.gov/
- Cryptol https://cryptol.net/
- Lando https://github.com/GaloisInc/BESSPIN-Lando
- SAW https://saw.galois.com/
- Frama-C https://frama-c.com/
- ACSL https://frama-c.com/html/acsl.html
- FRET https://ti.arc.nasa.gov/tech/rse/research/fret/
- Yosys https://www.yosyshq.com/
- Coq https://coq.inria.fr/
- PVS https://pvs.csl.sri.com/
- Galois is hiring in this R&D area! https://galois.com/careers/