|galois|

High Assurance Rigorous Digital Engineering for Nuclear Safety (HARDENS)

Joe Kiniry, Galois (kiniry@galois.com)

October 2022

Draft version as of October 2022. Unreviewed by NRC staff.

galois

Correctness · Cryptography · Security · AI and ML

Advancing computer science R&D
Creating trustworthiness in critical systems

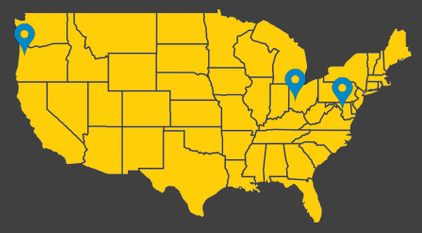
Clients

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

Overview of the Project

The NRC RFP

Summary of the NRC RFP

- The NRC RFP asked the following questions, all focused on safety-critical systems design.
- How might Model-Based Engineering…
  - …address software and systems complexity?
  - …address productivity challenges of complex distributed embedded systems?
  - …methods and tools support regulatory review?
  - …enable an alternative review process?
- What kinds of barriers or gaps are there in the use of MBE for regulatory review?
- Demonstrate modern MBE for a complex system.
- Begin to help NRC staff start to prepare for reviewing non-document-based evidence as a part of safety evaluation.

Plain Language Goals

- What is Model-Based Engineering (MBE)?
- How might it impact the design, construction, and assurance of nationally critical infrastructure?
- How to differentiate between hype and utility?
- How might the NRC evaluate a complex digital control system built with MBE?
- How might MBE solve some semi-recently unsolvable problems in safety-critical engineering?

NRC Project Info

- performer: Galois
- period of performance: circa 1 year
- active work on demonstrator: 4 months
- budget: circa $150K
- deliverables
  - demonstrator that runs in simulation and on an FPGA board
  - this presentation and Q&A
  - final report summarizing the project and making recommendations about the use of MBE

The HARDENS Project

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

HARDENS Goals

- explain modern Model-Based Engineering (MBE)
- frame MBE in the context of other high-assurance engineering disciplines (software, hardware, etc.)
- review the state-of-the-art in MBE tools, platforms, languages, methodologies, etc.
- build a concrete demonstrator of a high-assurance DI&C system using advanced MBE technologies available that are sponsored and used by the DoD
  - create the system quickly, and at low cost
  - use only open source software & open hardware

HARDENS Status

- final talk about project complete
- final report is nearing completion
- specification, software and simulated hardware implementation, and verification of high-assurance demonstrator is complete
- demonstrator running on the FPGA board not (yet) complete; what remains:
  - fitting the 3 core SoC onto this small board
  - finishing device drivers for sensors and actuators
  - final platform validation and demonstration

Model-Based Engineering in a Nutshell

Model-Based Engineering

- Model-Based Engineering (MBE) is fundamentally just *engineering* that uses *models*
  - *Engineering* is any kind of engineering: software, firmware, hardware, systems, safety, security…
  - *Models* are conceptual abstractions of physical or digital phenomena
- marketing information about products and services often talks about MBSE, which is ambiguous.
  - Model-Based *System* Engineering or
    Model-Based *Software* Engineering
- MBE *augments* other forms of engineering mentioned above, is *mandatory* for verification of high-assurance systems, and *complements* digital engineering with digital twins

The Basic Facts of MBE

- MBE is not a panacea, a silver bullet, or a joke
- its use on a project says very little, other than the project team uses moderately advanced tools

- MBE often means, at least, that…
  - some abstractions are written down that describe a system under development
  - repetitive, boilerplate code that programmers are prone to get wrong is automatically generated
  - the environment in which a system operates and evolves is taken into account

Engineering in the 2020s

- engineering today, especially for embedded systems, looks a lot like it did in the 90s
- programmers like to program, and docs are often out-of-date
- abstractions taught in CS, ME, and CE programs are rarely used in creating systems
- the C programming language and antique ISAs (e.g., x86 and ARM) are in widespread use
- hardware design languages (HDLs) standardized twenty years ago (e.g., SystemVerilog) are still not widely adopted, and hardware engineers are loath to learn new HDLs (Chisel, BSV, SystemC, many others)
- rigorous specifications (even assertions) are unheard of, especially anything formal
- hardware engineers still use hardware print statements, debuggers, and laborious testing to check correctness

Novel Improvements

- hardware is faster and cheaper than ever
- new languages that are great for safe embedded software engineering are breaking through (cf Rust)
- a new open, unencumbered Instruction Set Architecture (ISA) called RISC-V has catalyzed an explosion of novel hardware design for the masses
- open hardware has become respected
- reprogrammable hardware (e.g., FPGAs) is widely available, well-understood, and adopted
- Domain Specific Languages (DSLs) and Domain Specific Architectures (DSAs) have proliferated and increased productivity and quality

Typical Set of MBE Technologies

- **modeling languages**
  - UML: Unified Modeling Language
  - AADL: Architecture Analysis & Design Language
  - SysML: System Modeling Language
- **modeling environments**
  - IBM Rational Software, Dassault Systems Cameo, Eclipse Software Foundation's Eclipse, NI LabVIEW, Vitech's GENESYS, MathWorks System Composer, MATLAB, and Simulink, Sparx Systems Enterprise Architecture, OpenMBEE, Ansys ModelCenter and SCADE, BigLever onePLE

Typical Set of MBE Technologies

- **programming languages** such as…
  - various subsets of C (MISRA, verifiable)
  - Rust, Safety-critical Java, SPARK
- **reasoning tools** available from…
  - AbsInt, Galois, Microsoft, GrammaTech, Perforce, Parasoft, MathWorks, SonarSource, Ansys, Cadence, Synopsys, Siemens
- **specification languages** such as…
  - MATLAB, Simulink, SCADE, Statecharts, Java Modeling Language, CodeContracts, Cryptol, SystemVerilog Assertions
- **operating systems** available from…
  - Green Hills Software, Wind River, seL4 Foundation, etc.

MBE Technologies That Really Work

- data modeling and database generation, evolution, and management (Ruby on Rails and other tools)
- user interface design and behavior (e.g., Apple's Xcode, numerous tools/frameworks for Web 2.0)
- lexer and parser generators for language design (e.g., ANTLR, yacc/bison, flex/lex, XText, MPS)
- algorithm and protocol specification and reasoning systems (e.g., Cryptol, F*, Ivy, ProVerif, CryptoVerif, FDR)
- Design-by-Contract and assurance of software (SPARK, JML, CodeContracts, Dafny, Kotlin)
- Design-by-Contract and assurance of hardware (SVA, BlueCheck, OneSpin, Jasper, Formality, etc.)

MBE for High-assurance Engineering

- **very** few MBE technologies with a focus on mission-/safety-critical systems have been commercialized
  - design environments from Dassault, CMU SEI, Vitech, Sparx, IBM, Ansys, Siemens, Visual Paradigm, PTC, etc.
  - reasoning tools from AbsInt, Galois, GrammaTech, Ansys, Runtime Verification, etc.
- numerous MBE technologies exist that are applicable to high-assurance engineering, but have no corporation solely supporting their use (mostly supported by ICs and SMEs)
  - open source development and modeling environments (OpenMBEE, Eclipse, VS Code, etc.), model checking tools (CBMC, SPIN, UPPAAL, FDR4, SPIN, TLA+, Alloy), hardware verification tools (QED, Yosys, etc.)
- numerous MBE technologies have been funded by the DoD, IC, NASA
  - Cryptol, SAW, Crux, AADL, CASE, ARCOS, CoPilot, AdvoCate, etc.

Gap Analysis

Market Gaps

- there are insufficient customers in high-assurance engineering to support a rich ecosystem
- software engineers expect tools to be free
- few students graduate with knowledge of MBE
- the good is the enemy of the best
- easy-to-use trumps quality every day
- compliance and culpability are not forcing functions for new technologies
- mission-/safety-critical industries are risk adverse

Research Gaps

- usable semantics that refine between levels
- usable and useful traceability with semantics
- reversibility of traceability
- ambiguities in interpreting English into models
- connecting semi-formal and formal models
- consistency between text and graphical models
- specifying and reasoning about systems of systems
- supporting MBE in an agile methodology
- augmenting IDEs with MBE for continuous learning
- integrating rigor without loosing usability and scale
- supporting product lines throughout the lifecycle
- predictable and deterministic environments and products
- consistency of all models across the whole lifecycle
- trade space analysis and measurable, useful metrics

Practice Gaps

- writing models is rarely as immediately rewarding as programming, since most are not executable
- modeling tools feel more like documentation than programming, and programmers like to program
- there are rarely “print” statements in models
- some modeling languages are only graphical, and most programmers prefer textual languages
- trillions of lines of legacy code are not described with any existing models
- extracting models from code is much harder than generating code from models

MBE at Galois = RDE

The RDE Research Program at Galois

- **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)

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

Recommendations for MBE at the NRC

Recommendations for MBE at the NRC

- Tool Dependencies
- Certification Review
- Key Relations
  - Traceability
  - Representational and Descriptive
    - Structural and Operational
    - Data
    - Action
- Validating Models
- Validating Model-Implementation Correspondences
- Validating Claims

Tool Dependencies Recommendations

- **Tool Metadata**: For each artifact kind and file type in a system under review, ensure that the tool(s) necessary to validate that artifact are precisely documented (name, vendor, version, operating system platform).
- **Tool Availability**: For each artifact kind and file type in a system under review, ensure that the tool(s) necessary to validate that artifact is made available to the reviewer(s).
- **Evaluation Platform**: For each system under review, the system development team should provide a snapshot of a stable, coherent, documented evaluation platform to reviewers, such as a Virtual Machine image (e.g., VDI, VHD, VMDK file types), a Docker image, or a Nix configuration and cache.

Certification Review Recommendations

- **End-to-End**: Review at least one most abstract to most concrete chain.
- **Regularity**: Review at least one instance of each artifact kind in detail, and check for regularity across adjacent model elements of the same kind.
- **Workflow**: The recommended workflow for reviewing model-based assets moves through the four key facets of any system specification: structure, then data, then behavior, and then properties.
- **Limit Risk**: Review remaining artifacts (of all kinds, model, code, documentation, etc.) using a *risk-limiting audit* approach.

Traceability Recommendations

Traceability relations are those that link artifacts, often in a *bi-directional* fashion. Traceable relations should be labeled to indicate the purpose of the reference.

- **Traceability Completeness**: At every layer of model refinement, every assumption, goal, and requirement that frames a model-based specification at the abstract end of the refinement must be traceable to a corresponding assumption/rely/precondition, goal, or property in at least one refined artifact.
- **Bi-directional Traceability**: Every model, code, or assurance artifact *A* that is in a traceability relation to another artifact *B* should (i) permit a trace from *A* to *B* and from *B* to *A*, and (ii) the nature of both traces must be clear.
- **Traceability Reachability**: Every model, code, or assurance artifact must be reachable by a trace relation to at least one, and preferably only one, other artifact.
- **Traceability Implicitness**: Implicit traceability is superior to explicit traceability.

Representation Recommendations

Representational relations connect model abstractions to physical, digital, or computational manifestations.

- **Representation Realizability**: Every model must be realizable within an implementation; that is, once the model is refined to an implementation, an implementation must be able to be exist which fulfills all of the properties of the model. A model that is unrealizable is a model that is unimplementable.
- **Representation Soundness**: Models must be sound; unsound models are neither useful nor realizable. Models and model-code relations should be checked for soundness by attempting to rigorously validate or formally verify “bottom” implementations.
- **Representation Completeness**: Every model should be *relatively complete*, where relative completeness is defined with regards to (i) the expressivity of the modeling language, (ii) the utility and capability of tools which can reason about the model, and (iii) the property coverage necessary based upon the refinement relationships in which the model participates.

Descriptive Recommendations

A **descriptive relation** is a relation that also connects an abstraction to a descriptive target, but no property correspondence is meant to hold between the model and the object it describes.

- **Descriptive Accuracy**: Descriptions should be accurate, as judged by a reader who is familiar with its target.
- **Descriptive Completeness**: Descriptions should be complete with regards to the relative completeness of the artifacts to which they refine.

Structural and Operational Recommendations

Structural relations are *part-whole* relations of various kinds.

Operational relations are relations that describe possible unconstrained behaviors. One side of the relation often characterizes a set of possible abstract operations that can take place in a model, and the other side of the relation describes the concrete operations (such as function calls, message send/receives, UI events, etc.) that correspond to those abstract operations.

- **Structural Property Preservation**: All traceable refinements of models that have structure should preserve all structural properties.
- **Operation Realizability**: Operations must be realizable in a system.
- **Operational Elegance**: Operation design and implementation should be *elegant*. Elegance means: (1) there should be only a single operation that fulfills any system feature, (2) each operation should only implement a single feature, (3) simpler operational specifications are superior to more complex ones, (4) operations should either be pure queries (that change no system state) or simple commands that return no information beside success or failure, and (5) an operation that is both a query and a command (it changes state and then returns information) should be specified and implements as the composition of those two existing operations.
- **Operation-Action Coupling**: Every system operation should refine to a single system action.

Data Recommendations

Data relations describe models of data and their relationship with data in implementations.

- **Absent Invariants**: It is difficult to tell the difference between a component that has *no invariants* and one that has *absent invariants*. *Absent invariants* are invariants that hold for a model, but are not expressed by the development team due to accidental omission, oversight, or neglect.
- **Data Simplicity**: The *core data model* of a system should be modeled in the simplest possible way, and model or implementations complexities due to optimization, representation, platform, or infrastructure should be modeled as data refinements of the core data model.
- **Data Portability**: Data portability should be accounted for in a model using a platform product line specification, thereby factoring out platform-specific issues from core system specification matters.
- **Data Transparency**: Data models and their representations should contain no undocumented values, interpretations, structures, or relations. If the semantics of data fulfill this condition, it is *transparent* to any developer or reviewer.
- **Data Explainability**: The semantics of data should be *explainable* within a model refinement; every data invariant should be justified by a traceable refinement *up* to a more abstract model property, or *down* to a more concrete model or system property.

Action Recommendations

Action relations describe the decomposition of complex actions into simple events, and how those actions and events are realized in a system implementation. An *event* is a state change in a system that, from the point of view of the observer, is atomic. An action is set of a (linear or branching) sequences of events.

- **Positive Action Completeness**: Actions that define the behavior of a system under normal circumstances (no errors, exceptions, or faults) are *positive actions*. The model-based specification of a system should be *representationally complete* with regards to positive actions.
- **Negative Action Completeness**: Actions that define the behavior of a system under abnormal circumstances (errors, exceptions, or faults) are *negative actions*. The model-based specification of a system should be *representationally complete* with regards to negative actions.
- **Positive-Negative Consistency**: The composition of positive and negative actions must be consistent, and thus realizable.
- **Action-Implementation Consistency**: For every action *A*, *A*'s action-event structural relation (that is, each action is decomposed into a set of events) should be maintained by the model refinement or implementation of *A*.
- **Action-Event Completeness**: Every event must be used in at least one action, and realized by a single model refinement or implementation feature.

Relation Recommendations

- **Relation Typing**: For each relation identified or discovered in a system created with MBSE, if the relation is not explicitly typed according to the above classification scheme, deduce its type and check if it has the corresponding recommended properties described above. If it does not, ask the developers to justify its partial properties.

Validating Models Recommendations

- **Model Purpose**: Every model should have a *purpose*—a reason for it to exist.
- **Model Utility**: Every model must have a *utility*: a means by which to use the model toward providing assurance for a system.
- **Model Context**: Every model must have an explicit *context*: a frame of reference in which the model's assumptions/rely are explicit.
- **Model Relationships**: Models are often connected with one another through relations. A reviewer must understand and validate every model relationship, and every model relationship should have a purpose.
- **Model History**: A reviewer should be able to observe the history of every model artifact, as that history often tells a story about the design and modeling decisions made by developers during a model's evolution, validation, verification, and refinement.
- **Model Evolvability**: All models should be *evolvable* in a relation-preserving fashion.
- **Model Maintainability**: All models should be *maintainable* after system deployment.

Validating Model-Implementation Correspondences Recommendations

- **Model-Implementation Traceability**: The most concrete models that refine to implementations must be traceable.
- **Model-Implementation Rigor**: The strongest, preferred relationship between a model and an implementation is one that is *rigorous*. A *rigorous model-implementation relation* is one that is codified mechanically (programmatically, in a machine-readable fashion) and can be automatically (re-)checked computationally.
- **Model-Implementation Refinement**: The strongest, preferred model-implementation relation is one that is a *refinement* relation.
- **Model-Implementation Evolvability**: Model and their implementations should *co-evolve*, preferably in an automated fashion.
- **Model-Implementation Canonicity**: In every model-implementation relation, either the model or the implementation should be declared canonical, and used as the ground truth for the relation.

Validating Claims Recommendations (1)

- **Automated Claim Validation**: Automating claim validation is far superior to, and preferred to, manual claim validation.
- **Interactive Claim Validation**: While machine-assisted interactive claim validation can provide as much, or more, rigor, confidence, and assurance of a claim, interactive evidence often requires much more effort, time, cost, and attention than automatic validation.
- **Claim Multi-Validation**: Rigorously validating or formally verifying a claim with multiple tools or techniques is what we call *multi-validation*. In general, it is better to witness validation via multiple techniques, and multiple tools which rely upon distinct foundations, reasoning approaches, and different organizations.
- **Continuous Claim Validation**: Claims should be *continuously validated*, not checked all at once aperiodically with great effort. Preferably claims are (re-)checkable on every developer's development machine, on private or cloud-based compute servers, or in a continuous integration (CI) service (e.g., as a part of a DevSecOps infrastructure).

Validating Claims Recommendations (2)

- **Claim Validation Dependencies**: All dependencies of a claim and its validation should be included in an assurance package, be explicitly or implicitly traceable, and be validated as well, preferably starting from axiomatic foundations and moving up dependencies eventually to top-most theorems and properties.
- **Safety Claim Validation**: Safety claims should be, if at all possible, validated with formal verification, not (only) with runtime assertion/property checking.
- **Risk Mitigation Claim Validation**: Risk mitigation claims should, if at all possible, be validated using mechanized/computational techniques, and not just with subjective human judgment.
- **Hazard Mitigation Claim Validation**: Hazard analysis and mitigation claims should, if at all possible, be validated using mechanized/computational techniques, and not just with subjective human judgment. And when modeled computationally, they should be validated with formal verification, not (only) with runtime assertion/property checking.

Validating Claims Recommendations (3)

- **Security Mitigation Claim Validation**: Security analyses and mitigations should, if at all possible, be validated using mechanized/computational techniques, and not just with subjective human judgment. And when modeled computationally, they should be validated with formal verification, not (only) with runtime assertion/property checking.
- **Claim Fidelity**: The *fidelity* of every claim must be explicit if it is not *precise*. A *precise* claim is one that is a theorem/property of model or code for which all assumptions are explicit for for which a proof could be created/generated. An *imprecise* claim is a claim that is not precise. The *fidelity* of a claim is the context of a claim and its evidence—particularly its set of assumptions—including any necessary caveats, error bars, or other statistical analysis of related measures.

Final Key Recommendations

- **Understandable**: An MBSE-based assurance case should be *understandable* to any moderately experienced engineer.
- **Believable**: An MBSE-based assurance case should be *believable* to any moderately experienced engineer.
- **Coherent**: An MBSE-based assurance case should be *coherent* to any moderately experienced engineer.
- **Surety**: An MBSE-based assurance case should meet or surpass the *assurance threshold* set by a client (its *surety*), and the fact that an assurance case is not yet good enough should not be a surprise to any involved.

BREAK

MBE Modeling Platforms, Tools, and Languages

Dominant Tools

- only a small set of tools dominate the landscape

- Dassault Cameo product line / CATIA
- CMU SEI's AADL product line / OSATE2
- Ansys Medini, SCADE Suite, Digital Safety Manager, Twin Builder, ModelCenter, Systems Tool Kit
- various IDE ecosystems (Eclipse, Microsoft Visual Studio and VS Code, JetBrains IDEA, CLion, and MPS, etc.)
- Cadence, Synopsys, and Siemens product lines
- IBM Rational product line
- honorable mentions: PTC, Sparx, Visual Paradigm

Domain Engineering

- domain engineering products either focus wholly on product line engineering feature models or on defining and managing ontologies
  - BigLever's onePLE
  - there are dozens of ontology management tools, but none have stood out in the marketplace

Requirements Engineering

- several tools dominate the landscape of writing and managing requirements—some primitive (Microsoft Excel), some heavyweight (DOORS)
- no commercial tool formalizes semi-formal requirements into formal properties
- the main tools that mechanize translations come from NASA (FRET) and Collins Aerospace (SpeAR)

Data Engineering

- many MBE frameworks and tools exist for specifying data's shape, storage, ownership, etc.
- entity-relation models, or class/datatype models, are used to generate in-memory datatypes, database schemas and operations, etc.
- none of them connect out of the box with popular systems engineering tools or software MBE tools
- formally verifying high-assurance datatypes is straightforward and can be used to assure persistence, networking, and event subsystems

UI/UX Engineering

- some of the earliest MBE tools focused on UI design and implementation
- drag-and-drop UI builders go as far back as NeXTStep circa 1989
- modern tools permit the modeling of desktop applications, mobile applications, and web applications using declarative domain specific languages and interactive visual IDEs (a la Xcode)

Systems Engineering

- only two systems modeling languages matter today
  - AADL: Architecture Analysis & Design Language
  - SysML: System Modeling Language

- AADL is largely used only for embedded systems
  - there is a moderate sized set (circa 12–15) of open/disclosed source tools that reason about AADL models and generate code from models
  - the assurance case possible using AADL tooling is strong, and has seen use in the DoD and NASA

Systems Engineering

- only two systems modeling languages matter today
  - AADL: Architecture Analysis & Design Language
  - SysML: System Modeling Language

- SysML is largely used to model complex systems
  - few tools exist to reason about SysML models
  - no tools generate code from SysML models

- SysML version 2 is under development now, and represents a significant evolution of SysML away from its original UML-based foundations

Reflections on SysMLv2

- the syntax of SysMLv2 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

Software Engineering

- MBE in software engineering focuses on
  - data modeling (for easy datatype definitions, persistence mechanisms, data transmission, database creation and management, etc.),
  - distributed systems design (packet definitions, automatic endpoint creation), and
  - user interface design (declarative specifications of UIs and their behavior)
- MBE for high-assurance software engineering focus on…
  - behavioral interface specification languages (BISLs) and their tools (ACSL, JML, CodeContracts, SPARK, Kotlin, new experiments in Rust)
  - frameworks and tools for creating Domain Specific Languages (EDSLs, MPS, XText, Visual Studio, Kotlin, Scala, etc.)

Firmware Engineering

- little to no tools exist for MBE for general firmware

- creating **new** firmware is just treated as creating low-level, bare-metal software
  - e.g., device drivers and bare-metal (OS-free) systems

- reverse engineering **existing** firmware (to perform reasoning, security analyses, retrofitting, etc.) sometime requires lifting models from binaries (e.g., tools funded by DARPA available from Galois, Ghidra from the NSA)
- in general, existing firmware has vanishing low transparency and trust, so is suspect in any high-assurance system

Hardware Engineering

- MBE in hardware mainly comes in the form of hand-writing models in Hardware Design Languages (HDLs) like VHDL and Verilog
- SystemVerilog, while standardized 20 years ago, has still not seen widespread adoptions
- SystemC, a subset of C that can be compiled to software or synthesized to hardware, has mainly seen use in the writing of tests and emulators
- MBE tools that permit, e.g., the specification and reasoning about simple models like Finite State Machines have largely been rejected by industry

Safety Engineering

- several commercial and open source tools exist that permit the specification of, and reasoning about, safety engineering models
  - Leveson's (MIT) and NASA/JPL's research and tools are largely looked upon as the state-of-the-art
  - e.g., SFTA, SFMEA, etc. see the *NASA Software Engineering Handbook* and the *Systems Safety Handbook* (and many more artifacts) for more details
- no MBE tools available today integrate hazard analysis with high-assurance MBE
- mainly safety engineering is manual, leverages RMF-based approaches to qualified and quantified risk analysis, and is ad hoc

Security Engineering

- there is a lot of snake oil in security engineering on the market today
- very little is grounded in foundations that deeply connect security models to code and its properties
- the DARPA I2O HACMS program focuses on the MBE-based design, implementation, and formal verification of embedded systems
- the DARPA I2O CASE program has focused for several years on defining new security-centric DSLs for annotating AADL systems engineering models
- several DARPA MTO programs (e.g., CRASH, SSITH ,and AISS) programs have focused on hardware security engineering (mitigating IP theft or software-based attacks)

Best Exemplars in the USG

- NASA/JPL
  - the best systems+safety engineering in the world
- NSA
  - the best software+security engineering
- DARPA
  - driving R&D in advanced MBE, assurance, etc.
- Sandia National Labs
- MIT Lincoln Labs
- DoD Primes (such as LMCO, Raytheon, GD, Boeing, Northrop Grumman, BAE, Rolls-Royce)
  - programs rarely demonstrate advanced MBE

Best Exemplars in Industry

- Amazon
- AMD
- Apple
- ARM
- Facebook
- Google
- Intel
- Microsoft
- NVIDIA
- SiFive

Discussion Topics:

Challenge Problems in Safety-Critical Systems

Challenge Questions (1)

1 What independently verifiable evidence would suffice for eliminating the need for “design diversity” yielding at least the same level of assurance (as obtained with diverse designs) that the safety function of a safety system such as RPS (reactor protection system) will be performed when needed.

2 What logically correct reasoning could demonstrate that the evidence supports the conclusion (i.e., the safety function will be performed when needed) with the residual uncertainty lower than or comparable to the uncertainty associated with a design incorporating “design diversity”.

3 What independently verifiable evidence would suffice to demonstrate that the same deficiencies are not present in the diverse designs OR, if present, would not degrade the performance of the safety function.

4 What logically correct reasoning could demonstrate that the evidence supports the conclusion: the safety function will be performed when needed and no deficiency common to the diverse designs will prevent this performance.

5 What independently verifiable evidence would satisfy the claim that interactions across redundant divisions will not degrade the performance of the safety function.

6 What logically correct reasoning could demonstrate that interactions across redundant divisions will not degrade the safety function.

7 What independently verifiable evidence would satisfy the claim that safety constraints were identified correctly, completely, and without any inconsistencies.

8 What logically correct reasoning could demonstrate that the evidence satisfies the claim.

Challenge Questions (2)

1 What independently verifiable evidence would indicate that there is a significant potential for degrading the safety function of systems such as RPS due to a combined effect of the uncertainties from V&V of development phase work products.

2 What logically correct reasoning could be used to determine the combined effect of the uncertainties from V&V of development phase work products.

3 What independently verifiable evidence is needed to justify non-compliance with over-constraints (or prescriptive content) in standards (e.g., IEEE standards).

4 What logically correct reasoning (or argument structure) could be used to justify such non-compliance with a standard.

5 What independently verifiable evidence would suffice to claim that the residual uncertainties will not degrade the performance of the safety function.

6 What logically correct reasoning (or argument structure) could be used to justify the claim.

7 What structure of the safety assurance plan would suffice to align expectations between a licensee and the regulator (when using an outcome-oriented approach rather than a compliance-oriented approach).

8 What independently verifiable evidence would suffice to claim that the practitioner has the competence to identify all significant hazards (i.e., conditions which could degrade the performance of the safety function of RPS) which are rooted in engineering deficiencies, especially hazards caused by interactions.

9 What independently verifiable evidence would suffice to claim that the people performing the hazard analysis possess the right skills to produce correct and complete results.

BREAK

The Reactor Trip System Demonstrator

An Overview of the RTS

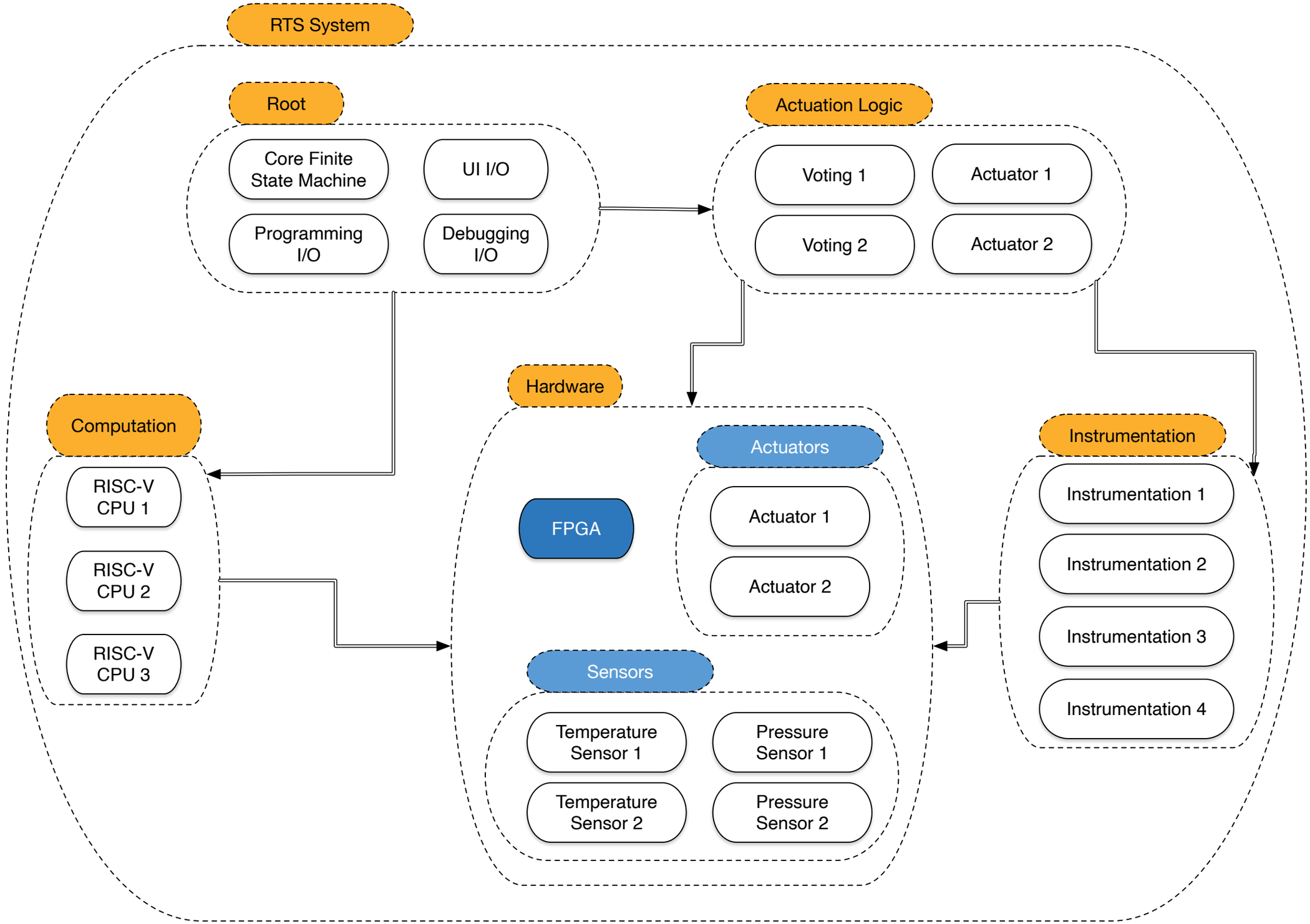
The Reactor Trip System (RTS)

- 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”

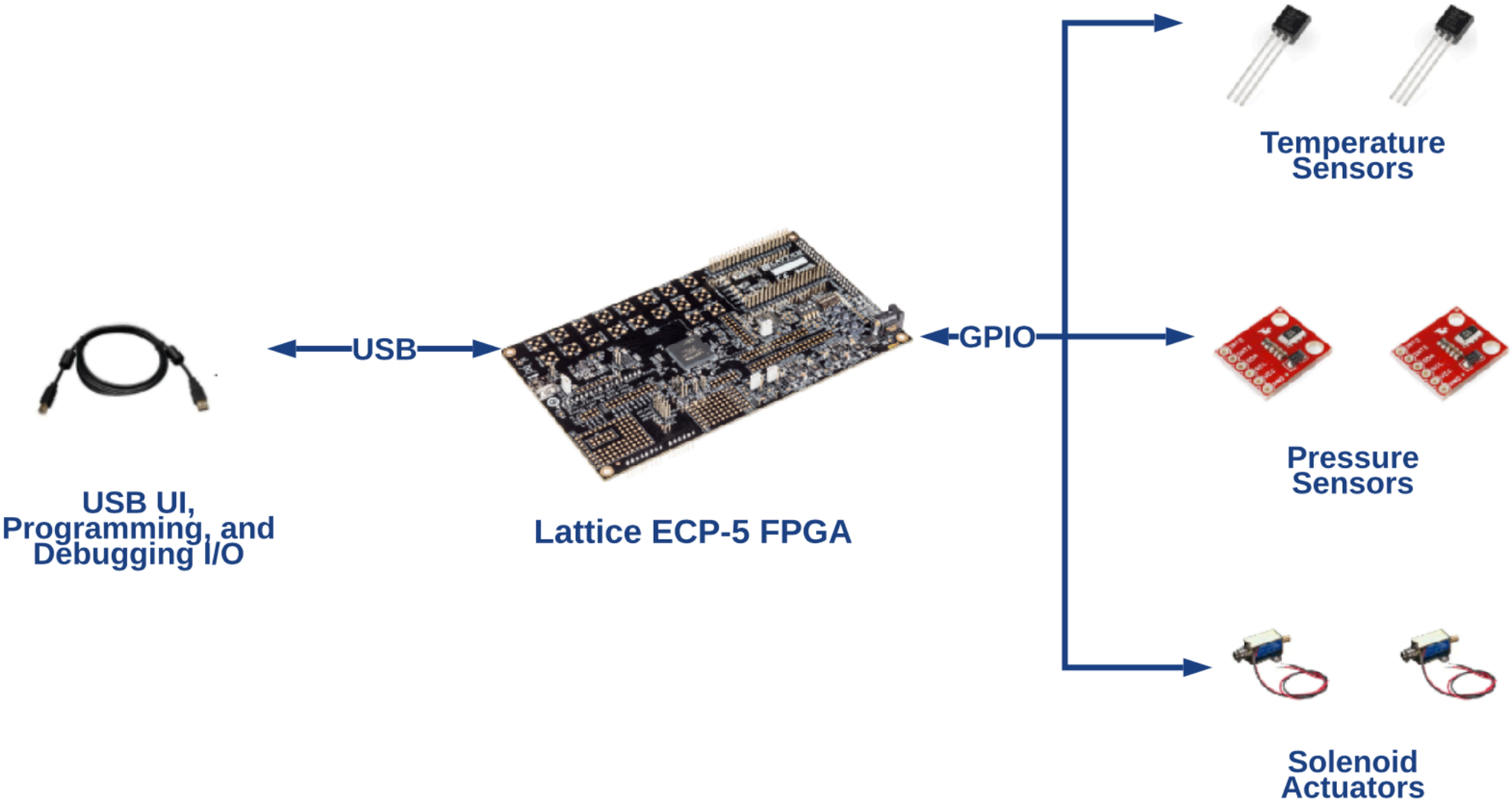
Digital Instrumentation & Control

- 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.

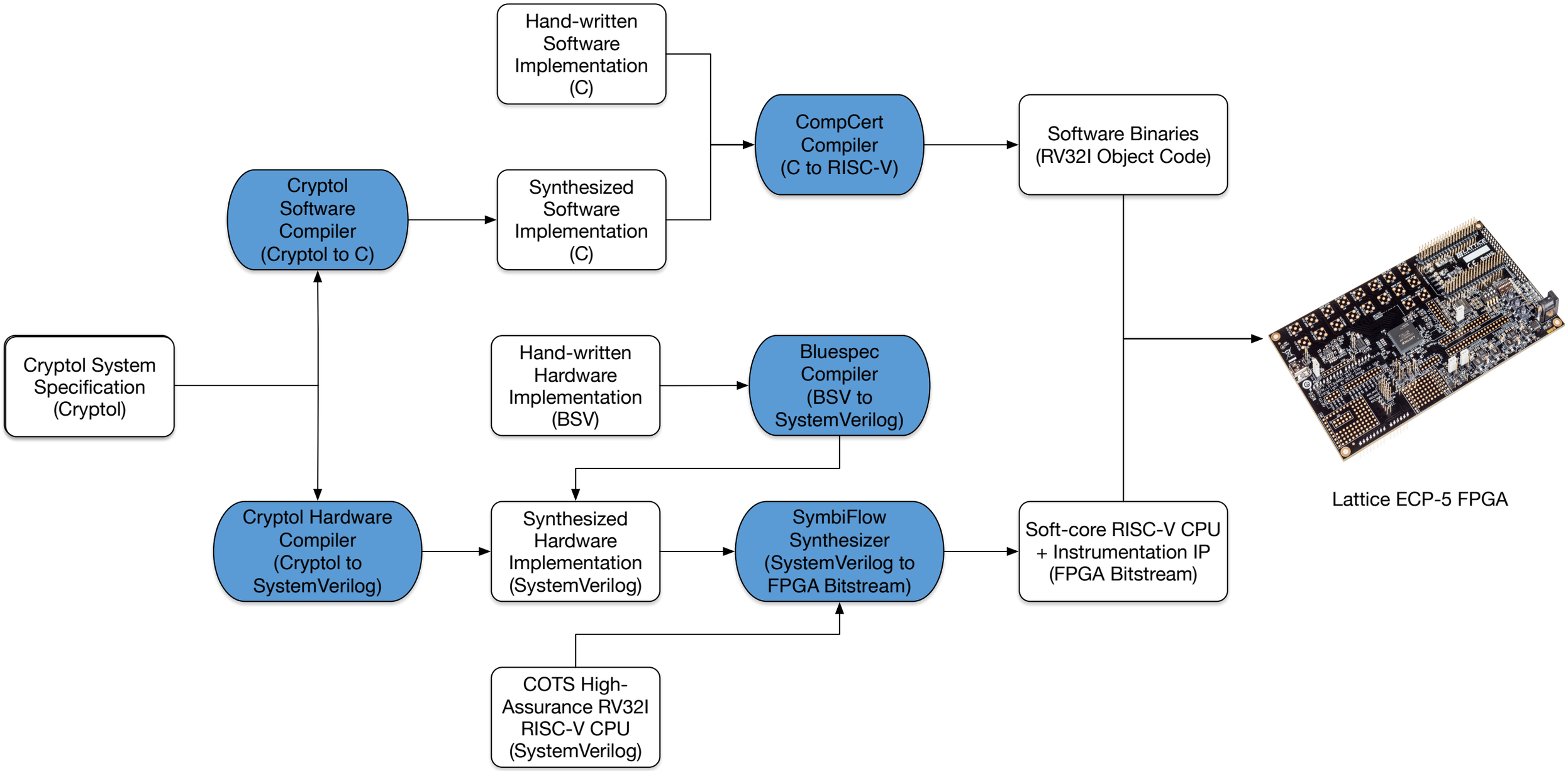
Proposal System Architecture



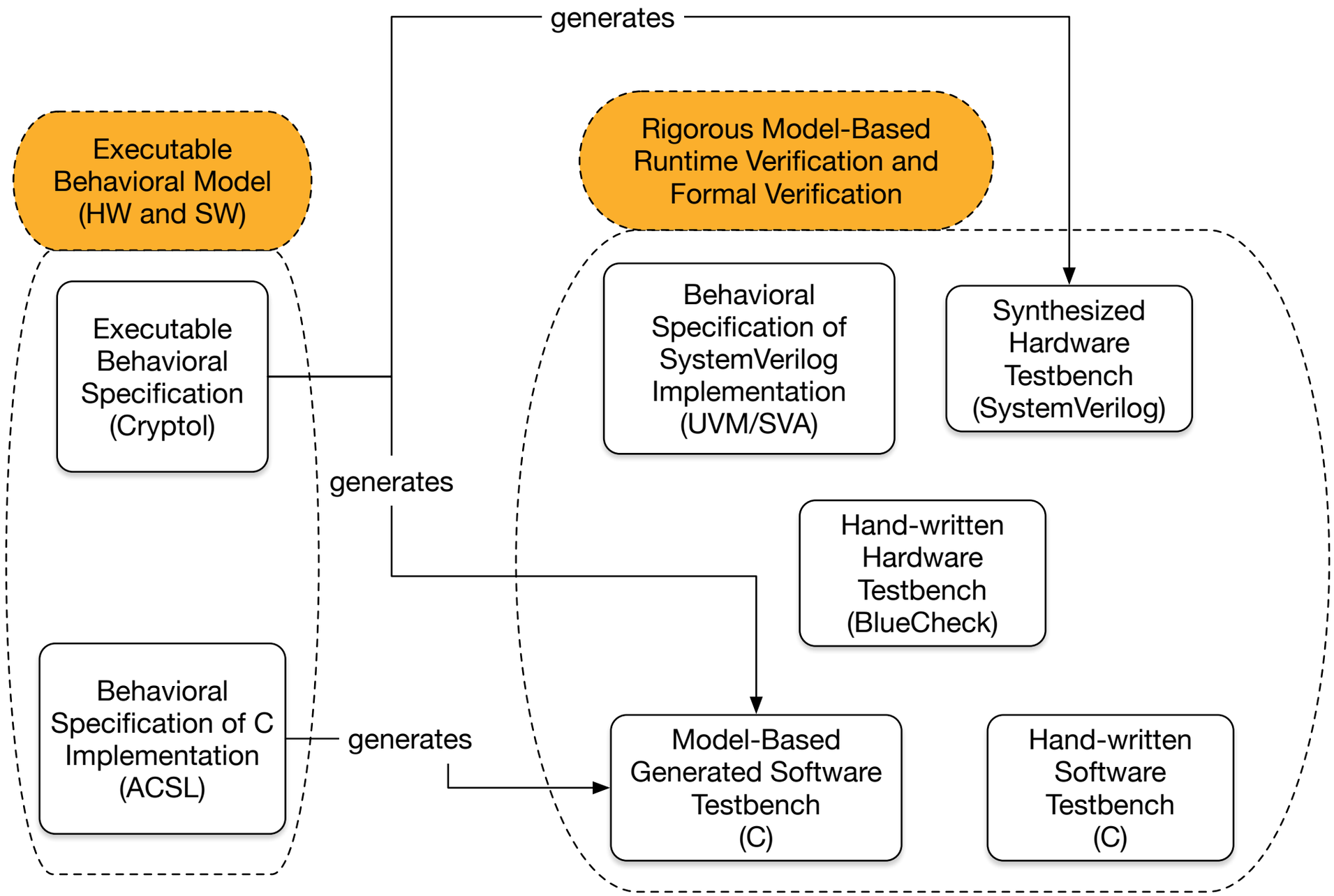
The RTS Hardware Artifacts



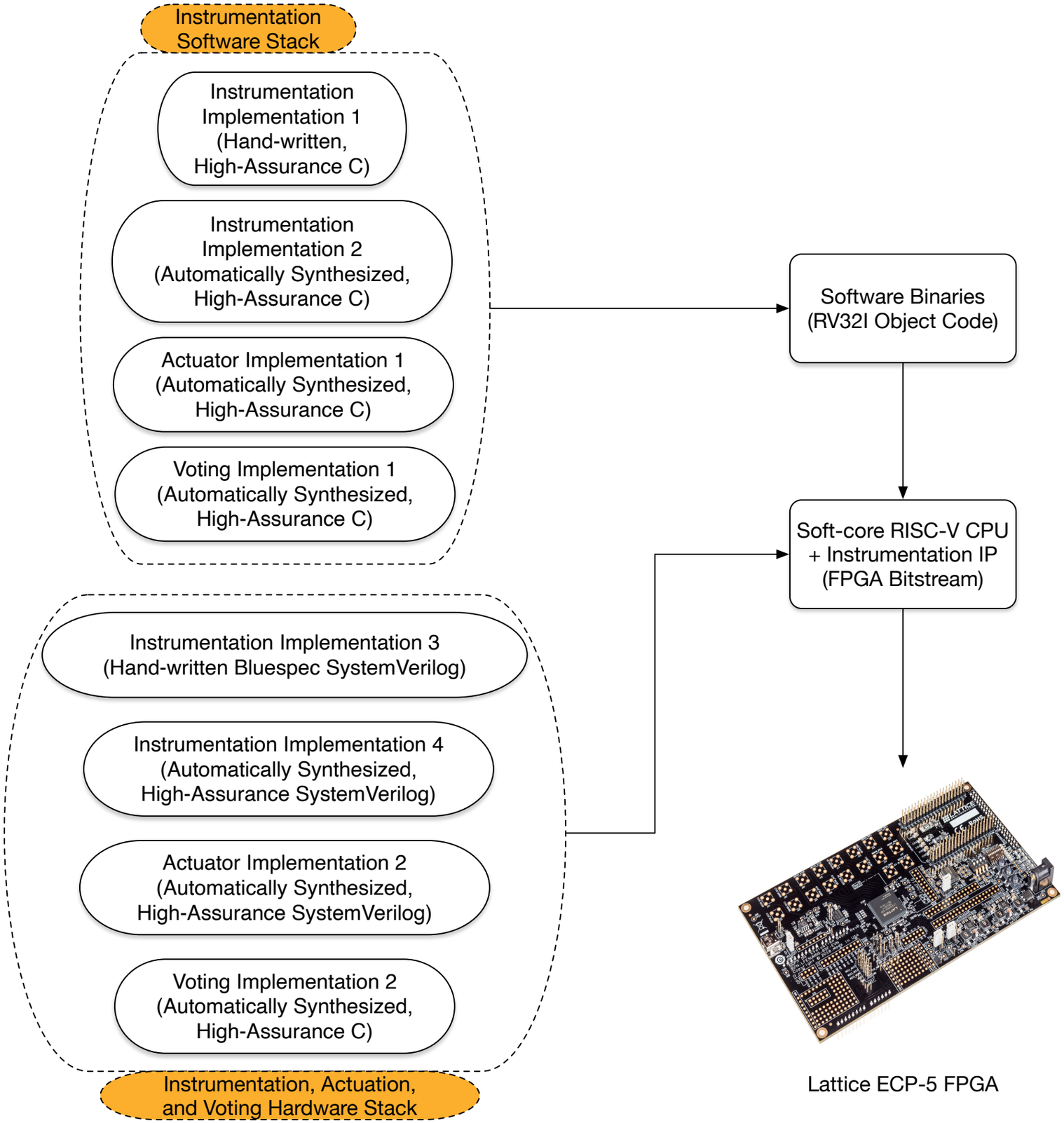
Implementation Artifacts and Relations



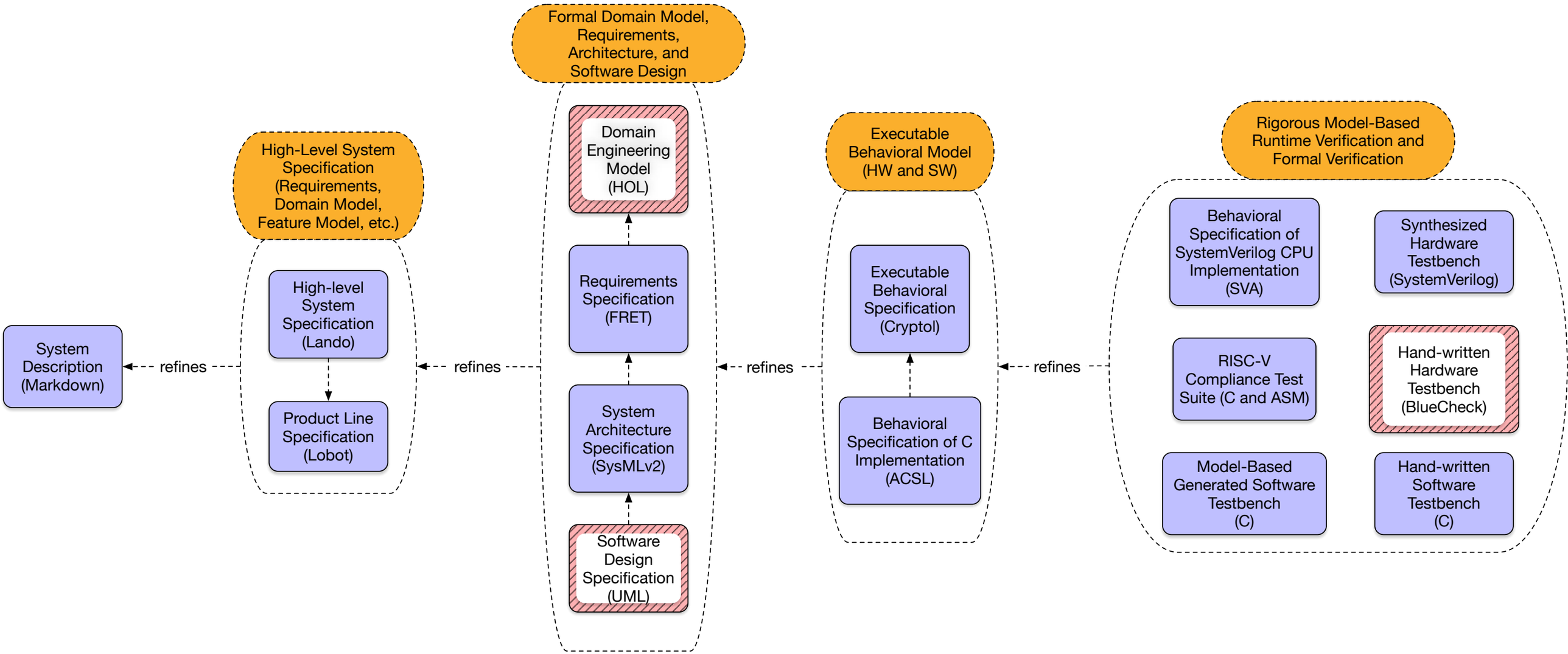
Generation of Assurance Artifacts



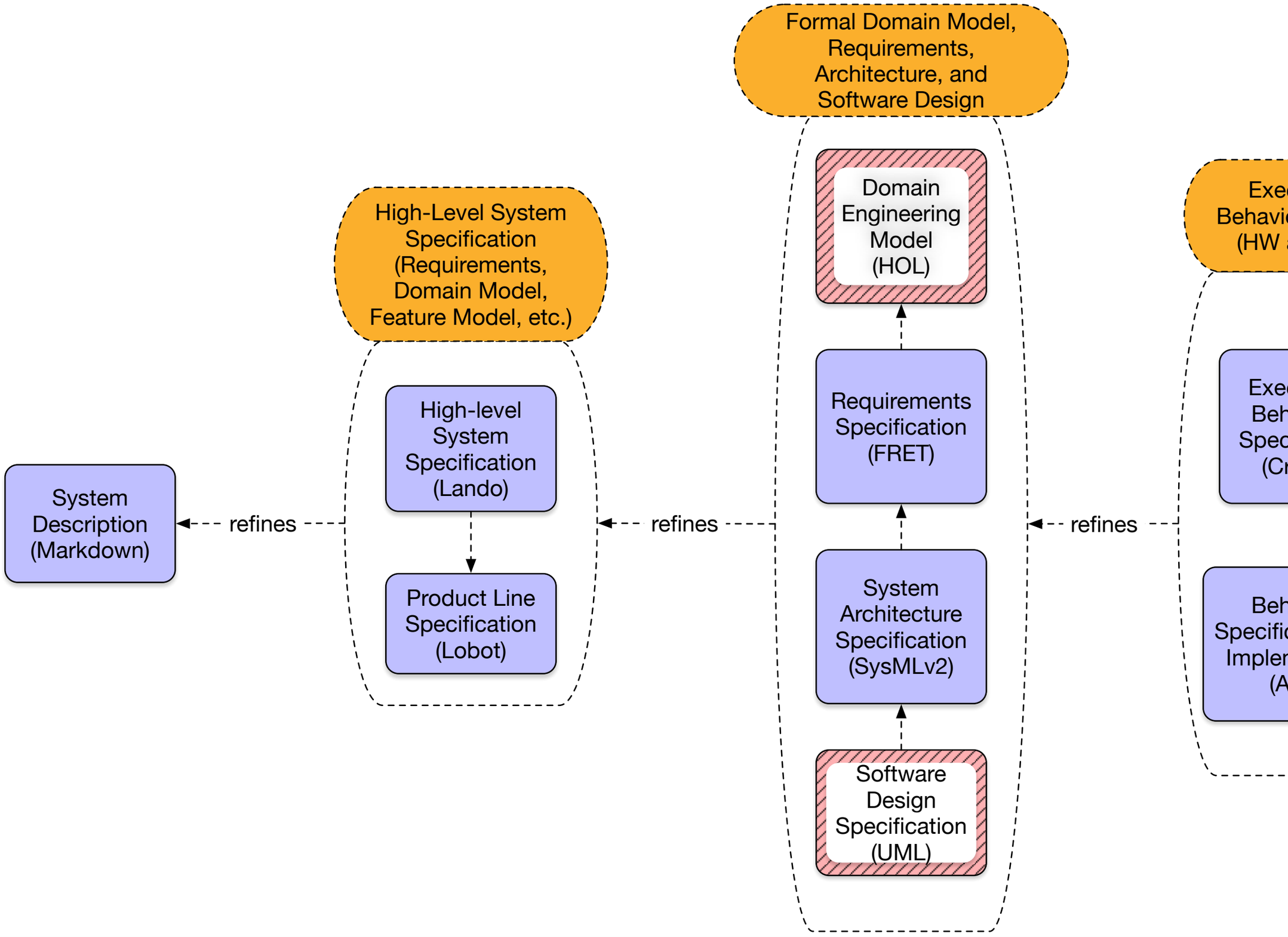
RTS Instrumentation Architecture



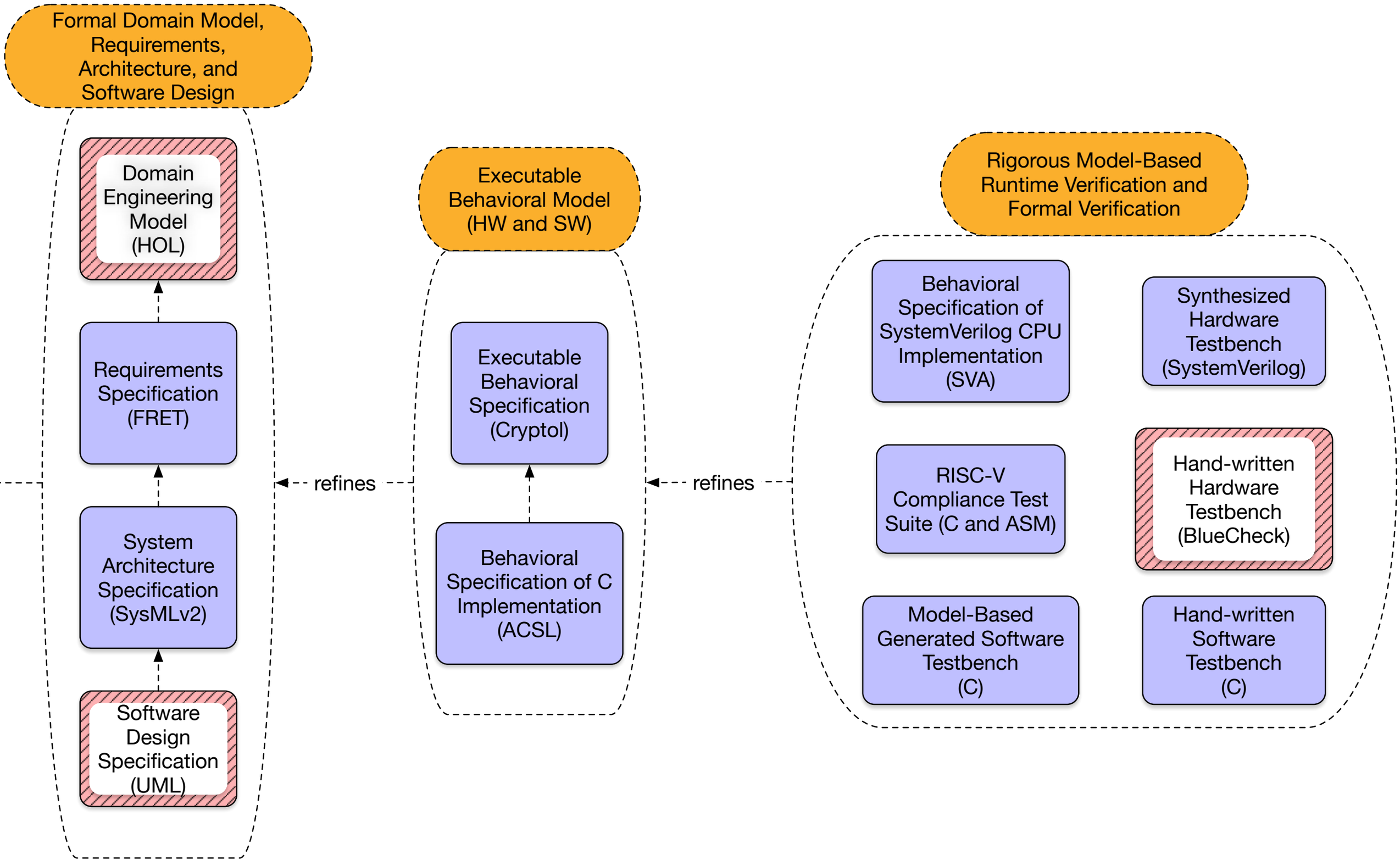
RTS Specification Artifacts



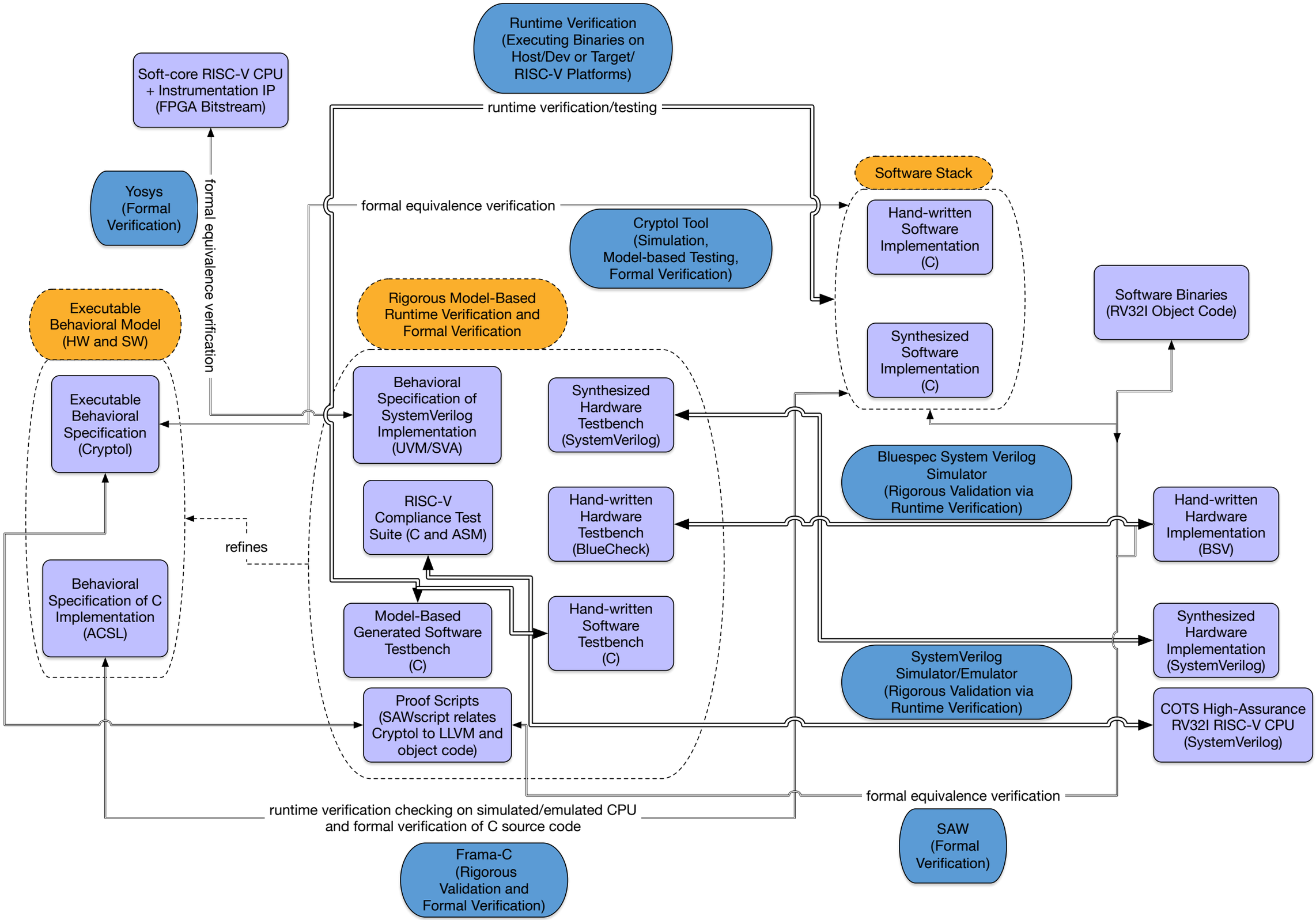
RTS Specification Artifacts



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

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

- 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

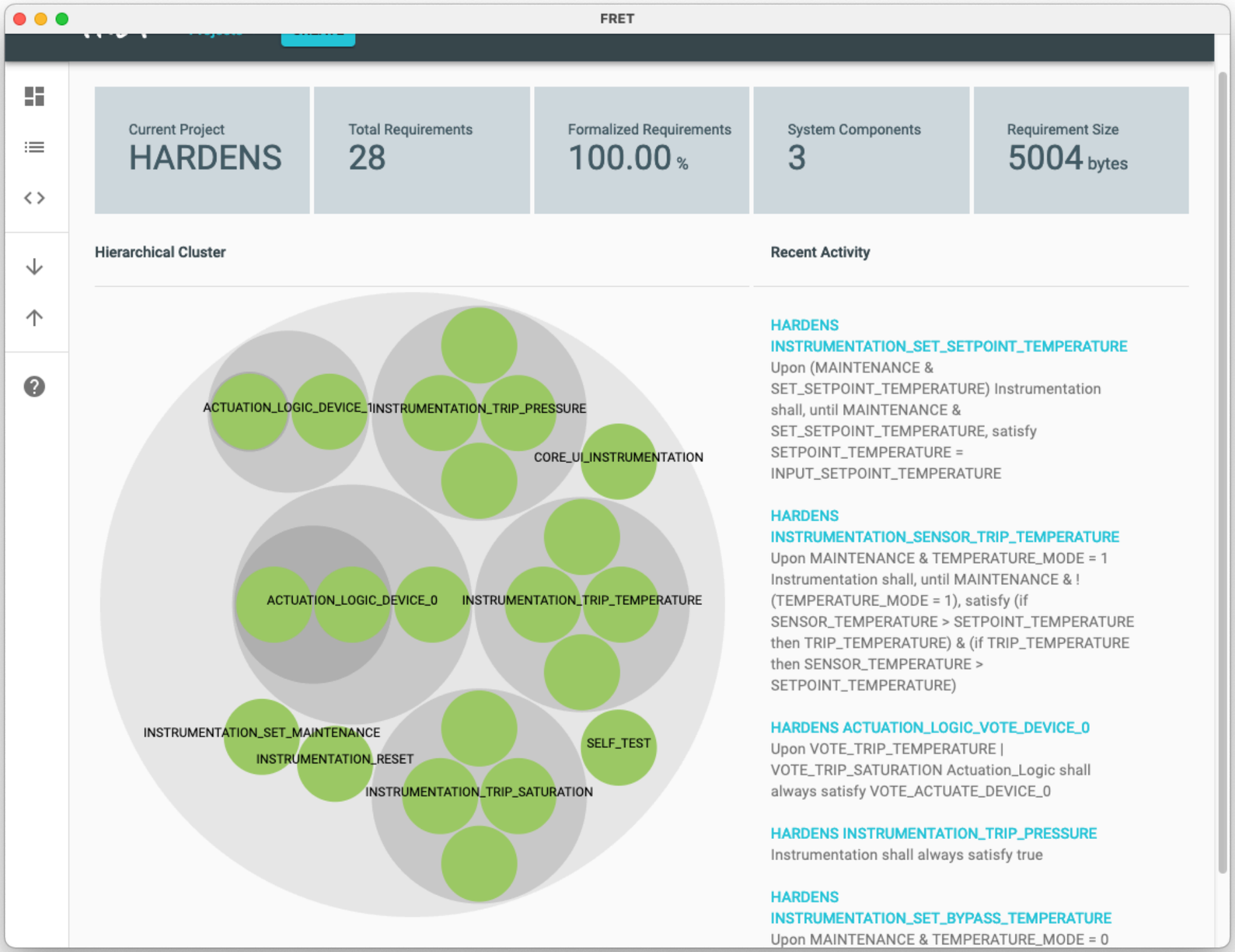
```
requirements NRC Characteristics
// The requirements driven by the IEEE 603-2018 standard for NPP I&C
// systems.

// 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

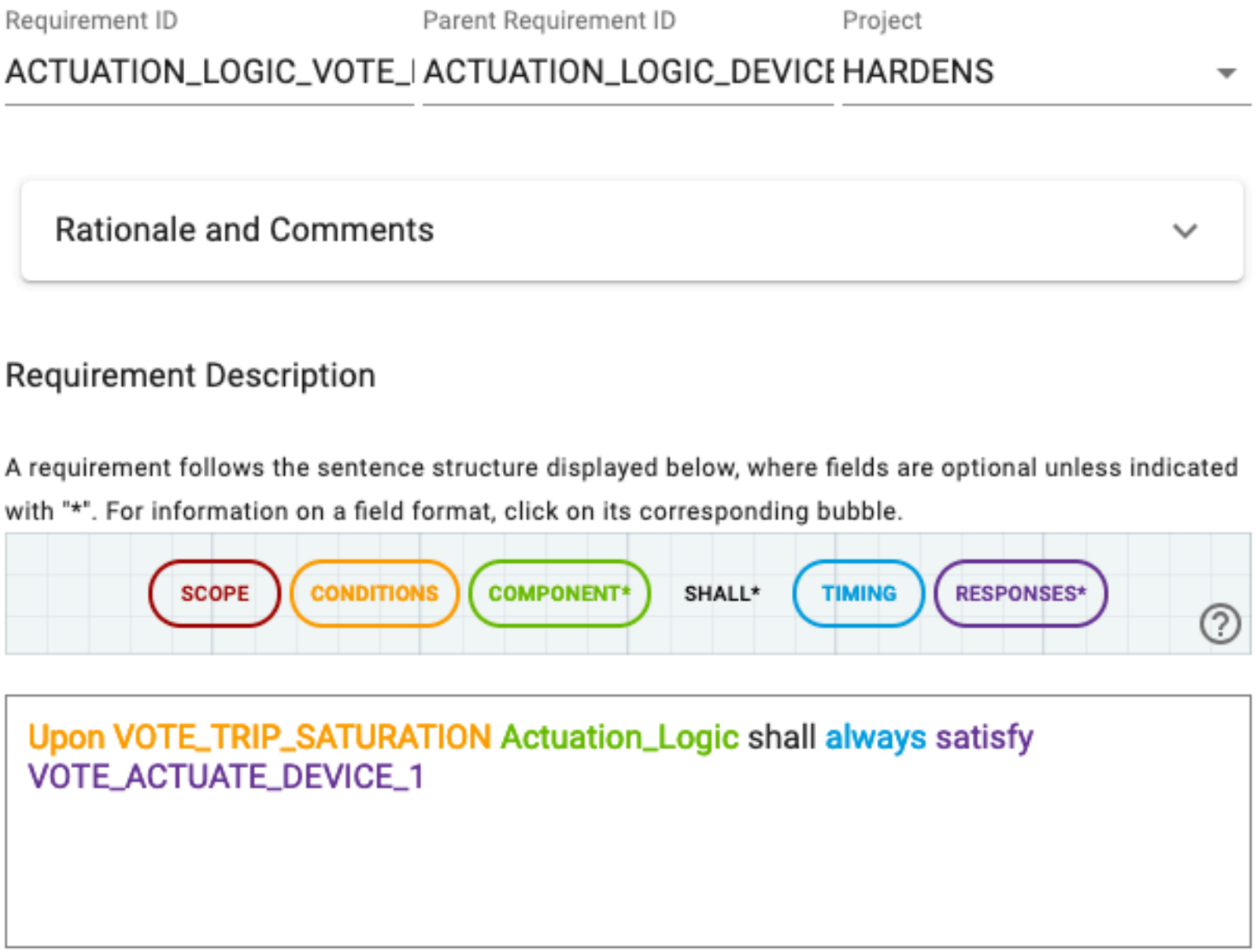
- FRET is a tool for writing, understanding, formalizing, and analyzing requirements.
- Users write requirements in an intuitive, restricted natural language, called FRETISH, with precise, unambiguous meaning.
- For a FRETISH requirement, FRET:
  1. produces natural language and diagrammatic explanations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

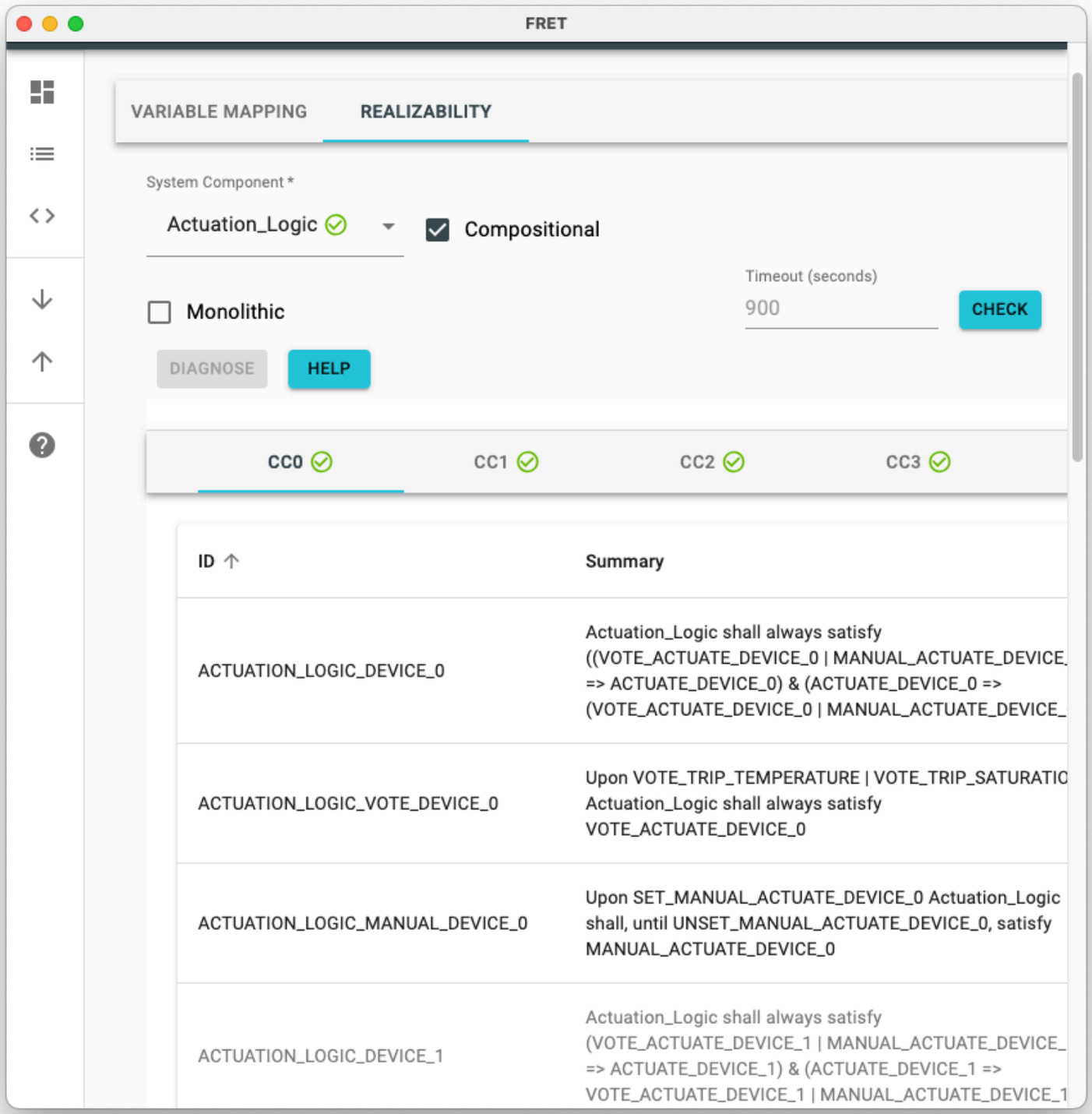


- a web-based, rich, dynamic-feedback editor for individual requirements
- requirements have IDs and are organized in a hierarchy
- provides grammar information and examples during editing
- provides English and diagrammatic explanations to clarify subtle semantic issues

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

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. */
}
```

```
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 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;
```

```
-- 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 }
```

Executable, Denotational, and Behavioral Models

Kinds of Models

- Executable Models
- Denotational Models
- Behavioral Models

Executable and Denotational Models

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 Models

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);
```

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 Cases

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 parametrized software property that is executable

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”

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

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

Conclusion

HARDENS as an Experiment

- HARDENS has been a successful experiment in demonstrating RDE for safety-critical, mission-critical embedded systems engineering
  - we have finished far more engineering, with fewer resources, than many felt possible
    - the assurance case for the RTS is as strong as any we have ever witnessed (comparable to Common Criteria EAL 7+)
- we wish to finish the deployment and testing of the soft-core RISC-V SoC and device drives to an FPGA board, even on our own resources

RDE at Galois

- over half a dozen projects have been funded to further develop and use Rigorous Digital Engineering in practice
  - main focus areas at this time are ASIC design, space-based platforms, IoT platforms using post-quantum cryptography, DSLs, formal verification tools, and classified national security systems
- RDE continues to be used at some of Galois's daughter companies, such as Free & Fair and Niobium Microsystems
- we expect that more than one upcoming DARPA BAA will focus on developing or using RDE

Capabilities at Galois

- a course in Rigorous Digital Engineering is being written and will be available to the public
  - NRC staff are welcome to take the course
- the vast majority of our RDE tools are available as open source or disclosed source
  - only hardware-centric RDE tools have more restricted licensing
- Galois or a daughter company will provide support and training for RDE, our products, and products that are used in RDE (commercial, open source, research, etc.)

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/