HARDENS_Oct_2022

October 31, 2022

1 HARDENS

Principal Investigator: Joe Kiniry kiniry@galois.com

Project Lead: Andrew Bivin abivin@galois.com

Research Engineers: Alexander Bakst abakst@galois.com *and Michal Podhradsky* mpodhradsky@galois.com

Repository for the HARDENS project for the Nuclear Regulatory Commission.

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.

1.1 Project Overview

The goal of HARDENS is to provide to the NRC expert technical services in order to (1) develop a better understanding of how Model-Based Systems Engineering (MBSE) methods and tools can support regulatory reviews of adequate design and design assurance, and (2) identify any barriers or gaps associated with MBSE in a regulatory review of Digital Instrumentation and Control Systems for existing Nuclear Power Plants (NPPs).

In the HARDENS project Galois will demonstrate to the Nuclear Regulatory Commission (NRC) cutting- edge capabilities in the model-based design, validation, and verification of safety-critical, mission-critical, high-assurance systems. Our demonstrator includes high-assurance software and hardware, includes open source RISC-V Central Processing Units (CPUs), and lays the groundwork for a high-assurance reusable product for safety critical Digital Instrumentation and Control Systems systems in NPPs.

Details about the HARDENS project are found in our original proposal, which was written in response to the original NRC RFP.

This document summarizes the current state of affairs of the project and demonstrator.

1.2 Executive Summary

In the **High Assurance Rigorous Digital Engineering for Nuclear Safety** (HARDENS) project, Galois has developed a high-assurance, safety-critical demonstration system for the Nuclear Regulatory Commission using Rigorous Digital Engineering (RDE). The system in question is a Digital Instrumentation and Control (DI&C) system for Nuclear Power Plants (NPPs), and is called the Reactor Trip System (RTS).

RDE is the combination of *Model-based Engineering*, *Digital Engineering*, and *Applied Formal Methods*. The engineering focus of RDE is broad, as we have used it to perform *software, firmware, hardware, systems, domain, requirements, product line, safety, and security engineering* of *high-assurance, secure-by-design systems*. The HARDENS project includes nearly all of these kinds of engineering, but for security engineering at this time.

To our knowledge, this demonstrator is the most rigorously specified and assured system of its kind that includes formally assured software and hardware for a safety-critical system.

Model-based Engineering focuses on the use of semi-formal and formal models and their properties to describe aspects of a system independent of a particular implementation. Models are connected to each other through a variety of relations (refinement, containment, subtyping/subsumption/implication, traceability, etc.), models are either or both denotational or operational (and thus executable), and models are used to specify, examine, understand, and reason about a system well-prior to a line of code ever being written. Models are used for rigorous validation—through automatic model-based test bench generation and bisimulation—and formal verification—through automatic model-based verification bench generation.

The models used in HARDENS include, from most to least abstract: - a Lando high-level system specification model, which includes within it: + a domain engineering model, + a requirements engineering model, which includes: * derived certification requirements, * contractual requirements, * safety requirements, and * correctness requirements. + a product line (feature) model, + a static system model, + a dataflow model of the RDE methodology, + a system event model, + a system scenario model (including all normal and exceptional behaviors), + a hardware, software, and evidence Bill of Materials (BOMs), - a SysMLv2 system model, which includes within it, as refined directly from the Lando model: + a stakeholder model, + a domain engineering model, + a requirements engineering model, + property specifications for all correctness and safety properties derived from the formal requirements model, + a product line (feature/variant) model, + a static system model that includes both the software and hardware manifestations of the system, + a system action model, and + a validation and verification assurance case model. - a formal requirements model expressed in JPL's FRET tool, as refined from the Lando and SysMLv2 requirements models above, - a Cryptol model of the entire system, including all subsystems and components, including formal, executable digital twin models of the system's sensors, actuators, and compute infrastructure, and this Cryptol model includes a refinement of all formal requirements from FRET into Cryptol properties (theorems) about the Cryptol model itself, - a model of the semantics of the RISC-V instruction set, - a model-based specification of critical portions of the RTS's software stack expressed in ACSL, - an executable and synthesizable Bluespec System Verilog model of a family of RISC-V-based SoCs, and - a System Verilog executable and synthesizable model of a simple, in-order 32-bit RISC-V CPU (NERV, from YosysHQ).

Digital Engineering focuses on the use of *digital twins* of physical systems, subsystems, and their components. A digital twin is typically an executable model that has known and measurable

objective fidelity in relation to the models or systems that relate to the twin. For example, an executable Cryptol or SCADE model are a digital twins.

The HARDENS system includes several digital twins, including simulation and emulation of the system hardware (CPUs and SoCs), software implementation, and system model.

Applied formal methods are the sensible use of formal methods concepts, tools, and technologies to formally specifying and reasoning about systems and their properties. In the context of RDE and the HARDENS project, we use formal methods to achieve the following assurance: - critical components of the RTS are automatically synthesized from Cryptol model into both formally verifiable C implementations and formally verifiable System Verilog implementations, - automatically generated C code and hand-written implementations of the same models are used to fulfill safety-critical redundancy and fault-tolerance requirements, and all of those implementations are formally verified both against their model, as well as verified against each other as being equivalent, using Frama-C and Galois's SAW tool, - the RISC-V CPU is formally verified against the RISC-V ISA specification using the Yosys open source verification tool, - the RISC-V-based SoC is rigorously assured against the automatically generated end-to-end test bench, - the formal requirements specified in FRET are formally verified for consistency, completeness, and realizability using SAT and SMT solvers, - the refinement of these requirements into Cryptol properties are used as model validation theorems to rigorously check and formally verifying that the Cryptol model conforms to the requirements, - the Cryptol model is used to automatically generate a component-level and end-to-end test bench (in C) for the entire system, and that test bench is executed on all digital twins and (soon) the full hardware implementation as well, and - all models and assurance artifacts are traceable and sit in a semi-formal refinement hierarchy that spans semi-formal system specification written in precise natural language all of the way down for formally assured source code (in verifiable C), (a side-effect of the optional use of CompCert) binaries, and hardware designs (in System Verilog and Bluespec System Verilog).

[2]:
```
/** Semantic properties are annotation to model and system artifacts
    used to semantically markup those artifacts for documentation,
    traceability, and more. */
package 'Semantic Properties' {
  doc /* Semantic Properties are used to document arbitrary
         constructs in our specifications and implementations.
         @see https://www.kindsoftware.com/documents/whitepapers/code_standards/
 ↪properties.html
       */
  import ScalarValues::*;

  /* @todo kiniry The scope of every attribute needs to be tightened. */

  attribute def id SP 'Semantic Property';
  attribute def id SPD 'Semantic Property with Description' :>
    'Semantic Property' {
    attribute description: String;
  }
  attribute def Exception :> String;
```

```
package 'Meta-Information' {
  attribute def Author :> SP {
    author: String;
  }
  attribute def Lando :> SP {
    summary: String;
  }
  attribute def Bug :> SPD;
  attribute def Copyright :> SP {
    copyright: String;
  }
  attribute def Description :> SPD;
  attribute def History :> SPD;
  attribute def License :> SP {
    license: String;
  }
  attribute def Title :> SP {
    title: String;
  }
}

attribute def 'Author Description Scope Triple' :> SPD {
  import 'Meta-Information'::*;
  attribute author: Author;
  attribute scope: Boolean;
}
package 'Pending Work' {
  attribute def Idea :> 'Author Description Scope Triple' {
    classifier: String;
  }
  attribute def Review :> 'Author Description Scope Triple';
  attribute def Todo :> 'Author Description Scope Triple';
}

attribute def 'Rich Assertion' :> SPD {
  attribute label: String;
  attribute expression: Boolean;
  attribute exception : Exception;
}

attribute def 'Expression Description Pair' :> SPD {
  attribute expression: Boolean;
}

package 'Formal Specifications' {
  import Collections::*;
  enum def 'Modifies Frame' {
```

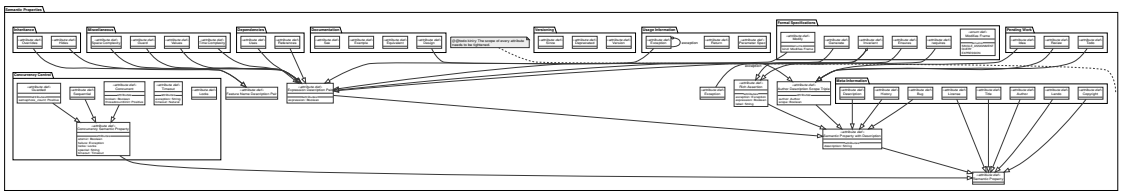
```
    SINGLE_ASSIGNMENT;
    QUERY;
    EXPRESSION;
  }
  attribute def Ensures :> 'Rich Assertion';
  attribute def Generate :> 'Expression Description Pair';
  attribute def Invariant :> 'Expression Description Pair' {
    exception: Exception;
  }
  attribute def Modify :> 'Expression Description Pair' {
    enum kind: 'Modifies Frame';
  }
  attribute def requires :> 'Rich Assertion';
}

package 'Concurrency Control' {
  import Collections::*;
  attribute def Locks :> Set;
  attribute def Timeout {
    attribute timeout: Natural;
    attribute exception: String;
  }
  attribute def 'Concurrency Semantic Property' :> 'Semantic Property' {
    attribute locks: Locks;
    attribute failure: Exception;
    attribute atomic: Boolean;
    attribute special: String;
    attribute timeout: Timeout;
  }
  attribute def Concurrent :> 'Concurrency Semantic Property' {
    attribute threadcountlimit: Positive;
    attribute broken: Boolean;
  }
  attribute def Sequential :> 'Concurrency Semantic Property';
  attribute def Guarded :> 'Concurrency Semantic Property' {
    attribute semaphore_count: Positive;
  }
}

package 'Usage Information' {
  attribute def 'Parameter Spec' {
    parameter_name: String;
    precondition: Boolean;
    description: String;
  }
  attribute def Return :> String;
  attribute def Exception :> 'Expression Description Pair' {
```

```
      exception: Exception;
    }
  }

  package Versioning {
    attribute def Version :> String;
    attribute def Deprecated :> String;
    attribute def Since :> String;
  }

  attribute def 'Feature Name Description Pair' {
    feature_name: String;
    description: String;
  }
  package Inheritance {
    attribute def Hides :> 'Feature Name Description Pair';
    attribute def Overrides :> 'Feature Name Description Pair';
  }

  package Documentation {
    attribute def Design :> 'Author Description Scope Triple';
    attribute def Equivalent :> String;
    attribute def Example :> String;
    attribute def See :> String;
  }

  package Dependencies {
    import Collections::*;
    attribute def References :> 'Expression Description Pair';
    // Note that we rename 'use' to 'uses' to avoid SysML keyword conflict.
    attribute def Uses :> 'Expression Description Pair';
  }

  package Miscellaneous {
    attribute def Guard :> 'Expression Description Pair';
    attribute def Values :> 'Expression Description Pair';
    attribute def 'Time Complexity' :> 'Expression Description Pair';
    attribute def 'Space Complexity' :> 'Expression Description Pair';
  }
}
```

[2]: Package Semantic Properties (bf69903f-8342-48ef-8869-0844f7b95e14)

[4]:
```
%viz "Semantic Properties"
```

[4]:



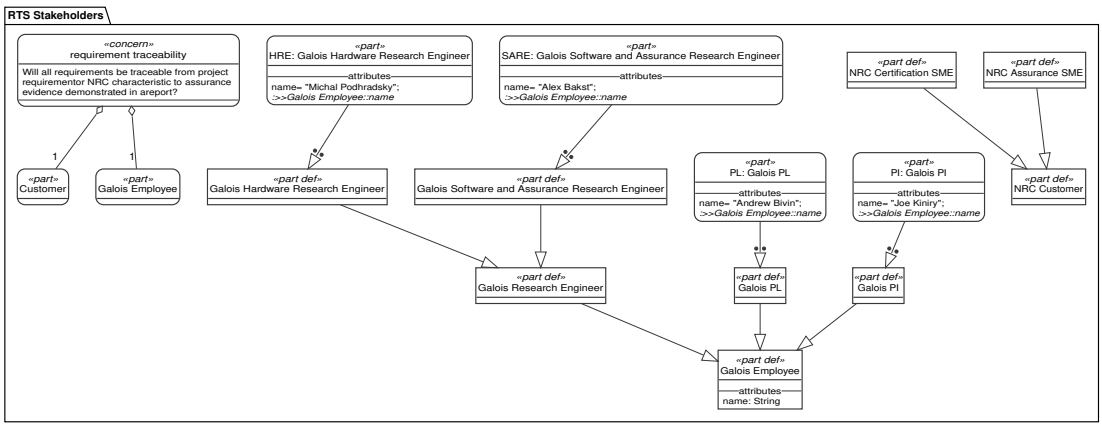
[5]:
```
package 'RTS Stakeholders'
{
  // NRC Stakeholders
  part def 'NRC Customer';
  part def 'NRC Assurance SME' :> 'NRC Customer';
  part def 'NRC Certification SME' :> 'NRC Customer';

  // Galois Stakeholders
  part def 'Galois Employee' {
    attribute
      name: ScalarValues::String;
      email: ScalarValues::String;
  }
  part def 'Galois PI' :> 'Galois Employee';
  part def 'Galois PL' :> 'Galois Employee';
  part def 'Galois Research Engineer' :> 'Galois Employee';
  part def 'Galois Software and Assurance Research Engineer'
    :> 'Galois Research Engineer';
  part def 'Galois Hardware Research Engineer' :> 'Galois Research Engineer';
  part PI : 'Galois PI' {
    attribute
      redefines name = "Joe Kiniry";
      redefines email = "kiniry@galois.com";
  }
  // @todo kiniry Add details for other performers.
  part PL : 'Galois PL' {
    attribute
      redefines name = "Andrew Bivin";
      redefines email = "abivin@galois.com";
  }
  part SARE : 'Galois Software and Assurance Research Engineer' {
    attribute
      redefines name = "Alex Bakst";
      redefines email = "abakst@galois.com";
  }
  part HRE : 'Galois Hardware Research Engineer' {
    attribute
      redefines name = "Michal Podhradsky";
      redefines email = "mpodhradsky@galois.com";
```

```
  }

  concern 'requirement traceability' {
    doc /* Will all requirements be traceable from project requirement
         * or NRC characteristic to assurance evidence demonstrated in a
         * report? */
    stakeholder 'Customer';
    stakeholder 'Galois Employee';
  }

  // @todo kiniry Express other concerns expressed in the RFP, from the NRC,
  // or from Galois or the Galois HARDENS team.
}
```

[5]: Package RTS Stakeholders (1f108c71-1615-4827-bac6-4eda11d3d7c4)

[6]:
```
%viz "RTS Stakeholders"
```

[6]:



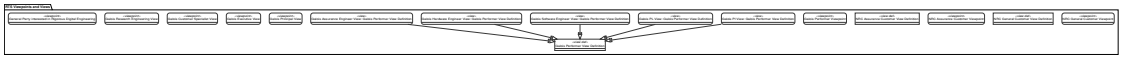
[7]:
```
import Views::*;

package 'RTS Viewpoints and Views' {
  import 'RTS Stakeholders'::*;

  // Viewpoints and view definitions specify the different points of view
  // that a system may be viewed from, and which parts of the system model
  // attend to each view.  We will specify viewpoints for a few kinds
  // of NRC actors, various kinds of HARDENS performers, various kinds of
  // other Galois employees who are interested in this project and its
  // outcomes, and other government and industry parties interested in RDE.

  viewpoint 'NRC General Customer Viewpoint' {}
```

```
  view def 'NRC General Customer View Definition' {}

  viewpoint 'NRC Assurance Customer Viewpoint' {}

  view def 'NRC Assurance Customer View Definition' {}

  viewpoint 'Galois Performer Viewpoint' {}

  view def 'Galois Performer View Definition' {}

  view 'Galois PI View' : 'Galois Performer View Definition' {}

  view 'Galois PL View' : 'Galois Performer View Definition' {}

  view 'Galois Software Engineer View' : 'Galois Performer View Definition' {}

  view 'Galois Hardware Engineer View' : 'Galois Performer View Definition' {}

  view 'Galois Assurance Engineer View' : 'Galois Performer View Definition' {}

  viewpoint 'Galois Principal View' {}

  viewpoint 'Galois Executive View' {}

  viewpoint 'Galois Customer Specialist View' {}

  viewpoint 'Galois Research Engineering View' {}

  viewpoint 'General Party interested in Rigorous Digital Engineering' {}
}
```

[7]: Package RTS Viewpoints and Views (da16fd87-4914-49a4-9e8e-2aa8a88dbfb5)

[8]:
```
%viz "RTS Viewpoints and Views"
```

[8]:



[10]:
```
/*
 # Reactor Trip System (RTS) High-assurance Demonstrator
 ## project: High Assurance Rigorous Digital Engineering for Nuclear Safety␣
 ↪(HARDENS)
 ### copyright (C) 2021-2022 Galois
```

```
 ### author: Joe Kiniry <kiniry@galois.com>
*/

// @see https://github.com/GaloisInc/HARDENS/issues/30

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

  // Original proposal glossary.
  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';
  /** A specification language integrated with support tools and an
      automated theorem prover, developed at the Computer Science Laboratory
      of SRI International.  PVS is based on a kernel consisting of an
      extension of Church's theory of types with dependent types, and is
      fundamentally a classical typed higher-order logic. */
  part def PVS;
  /** RISC-V (pronounced ``risk-five'') is an open standard instruction set
      architecture (ISA) based on established reduced instruction set
      computer (RISC) principles. Unlike most other ISA designs, the RISC-V
      ISA is provided under open source licenses that do not require fees to
      use. A number of companies are offering or have announced RISC-V
      hardware, open source operating systems with RISC-V support are
      available and the instruction set is supported in several popular
      software toolchains. */
  attribute def RISC_V_ISA :> ISA;
  /** A formal specification language that uses hierarchical finite state
```

```
    machines to specify system requirements. */
part def id RSML 'Requirements State Modeling Language';
/** The Boolean satisfiability problem (sometimes called propositional
    satisfiability problem and abbreviated SAT) is the problem of
    determining if there exists an interpretation that satisfies a given
    Boolean formula. */
abstract item def SAT;
/** The proof script language is used to specify the assumptions and proof
    goals of formal verifications to the SAW tool. */
part def SAWscript;
/** A CPU or SoC that is implemented in an HDL and synthesized to a
    bitstream and loaded onto an FPGA. */
abstract item def 'Soft Core' {
  // size: estimated number of gates
  // complexity: measured complexity metric
  // hdls: which HDLs are used in the design
}
/** A formally defined computer programming language based on the Ada
    programming language, intended for the development of high integrity
    software used in systems where predictable and highly reliable
    operation is essential. It facilitates the development of applications
    that demand safety, security, or business integrity. */
part def SPARK;
/** An integrated development environment for formally specifying and
    rigorously analyzing requirements. */
part def SpeAR;
/** VCC is a program verification tool that proves correctness of
    annotated concurrent C programs or finds problems in them. VCC extends
    C with design by contract features, like pre- and postcondition as
    well as type invariants. Annotated programs are translated to logical
    formulas using the Boogie tool, which passes them to an automated SMT
    solver Z3 to check their validity. */
part def id VCC 'Verifier for Concurrent C';
/** A software toolchain that includes static analyzers to check
    assertions about a C program; optimizing compilers to translate a C
    program to machine language; and operating systems and libraries to
    supply context for the C program. The Verified Software Toolchain
    project assures with machine-checked proofs that the assertions
    claimed at the top of the toolchain really hold in the
    machine-language program, running in the operating-system context. */
part def id VST 'Verified Software Toolchain';

// Mathematical modeling concepts in RDE.
abstract item def Refinement:> Relationship;
abstract item def Property:> BooleanExpression;
abstract item def 'Safety Property' :> Property;
abstract item def 'Correctness Property' :> Property;
```

```
abstract item def 'Security Property' :> Property;
abstract item def Model;
abstract item def 'Semi-Formal Model' :> Model;
abstract item def 'Formal Model' :> Model;
abstract item def Consistent:> Property;
abstract item def Complete:> Property;
abstract item def 'Consistent Model' :> Consistent, Model;
abstract item def 'Complete Model' :> Complete, Model;
abstract item def 'Consistent and Complete Model' :> 'Consistent Model',␣
↪'Complete Model';
abstract item def Denotational;
abstract item def Operational;
abstract item def Semantics;
/** A specification that has a precise, unambiguous, formal semantics
    grounded in real world formal foundations and systems engineering
    artifacts, such as source code and hardware designs. */
abstract item def Rigorous;
abstract item def Deterministic;
abstract item def 'Non-deterministic';
abstract part def id FM 'Formal Method';

// Systems modeling concepts in RDE.
// @design Probably in KerML or SysML domain libraries.
abstract item def Requirement;
abstract item def Scenario;
abstract item def Product;
abstract item def 'Product Line';
abstract item def Configure;
part def DOORS;
part def Clafer;
part def Lobot;
abstract item def id FSM 'Finite State Machine';
abstract item def id DFSM 'Deterministic Finite State Machine'
  :> FSM, Deterministic;
abstract item def id NFSM 'Non-deterministic Finite State Machine'
  :> FSM, 'Non-deterministic';
abstract item def id ASM 'Abstract State Machine';
abstract part def Design;
abstract part def Architecture;
abstract part def Specification;
abstract part def 'Architecture Specification' :> Specification;
abstract part def System;
abstract part def 'Distributed System' :> System;
abstract part def 'Concurrent System' :> System;
abstract part def Algorithm;
abstract part def Program;
```

```
// Concepts related to measurable abstractions of systems.
abstract item def Risk;
abstract item def Power;
abstract item def Resource;
abstract item def Reliability;

// Assurance concepts and techonlogies.
abstract item def id CDE 'Collaborative Development Environment';
abstract item def id CI 'Continuous Integration';
abstract item def id CV 'Continuous Verification';
abstract item def Analyzer;
abstract item def 'Static Analyzer' :> Analyzer;
abstract item def 'Dynamic Analyzer' :> Analyzer;
abstract part def Solver;
abstract part def id LF 'Logical Framework';
abstract item def 'High-Assurance';

// Concepts relevant to languages and protocols.
abstract part def Language;
abstract part def 'Specification Language' :> Language;
abstract part def Protocol;
abstract part def 'System Specification' :> Specification;
abstract item def 'Hand-written';
abstract item def 'Machine-generated';
abstract part def 'Source-level Specification Language'
  :> 'Specification Language';
abstract part def 'Model-based Specification Language'
  :> 'Specification Language';
abstract item def Cryptological;
abstract item def 'Cryptographic Protocol' :> Protocol, Cryptological;
abstract item def 'Cryptographic Algorithm' :> Algorithm, Cryptological;

// Software engineering.
abstract item def id PL 'Programming Language' :> Language;
abstract item def 'Source Code';
abstract part def C :> 'Programming Language';
abstract part def C_Source :> C, 'Source Code';
abstract item def 'Object Code';
abstract item def id IR 'Intermediate Representation';
abstract item def id LLVM 'Low-Level Virtual Machine' :> IR;
abstract item def Compiler {
  item input: Language[1..*];
  item output: Language[1..*];
}

// Hardware design.
abstract item def Hardware;
```

```
abstract item def SWaP {
  // attribute size:
  // attribute weight:
  // attribute power:
}
abstract item def Hard :> SWaP;
abstract item def 'Soft Core Hardware' :> Hardware, 'Soft Core';
abstract item def 'Physical Hardware' :> Hardware, Hard;
abstract part def Synthesizer :> Compiler;
abstract item def id HDL 'Hardware Design Language';
abstract part def BluespecSystemVerilog :> HDL;
abstract part def SystemVerilog :> HDL;
abstract part def Verilog :> SystemVerilog;
abstract part def Chisel :> HDL;
abstract part def id CPU 'Central Processing Unit';

// Hardware engineering concepts.
abstract part def Component;
abstract part def Switch :> Component;
abstract part def Button :> Component;
abstract part def Header :> Component;
abstract part def Interface :> Component;
abstract part def Connector :> Component;
abstract part def Memory :> Component;
abstract part def ASIC :> Component;
abstract item def id IO 'I/O';
abstract part def id GPIO 'General Purpose I/O';
abstract part def Sensor;
abstract part def 'Temperature Sensor';
abstract part def 'Pressure Sensor';
abstract part def Actuator;
abstract part def Solenoid :> Actuator;
abstract item def Bus;
abstract part def id USB 'Universal Serial Bus' :> Bus;
abstract part def LED;
abstract part def Cable;
abstract part def id FPGA 'Field-Programmable Gate Array' :> ASIC;
abstract part def 'ECP-5' :> FPGA;
abstract part def id PCB 'Printed Circuit Board' {
  part components: Component[*];
}
abstract part def 'USB Connector' :> USB, Connector;
abstract part def id USB_Mini 'USB Mini Connector' :> 'USB Connector';
abstract part def PMOD;
abstract part def JTAG:> Protocol;
abstract part def Driver;
port def USB_In {
```

```
  in item 'USB Connector';
}
port def USB_Out {
  out item 'USB Connector';
}
/** A normal USB cable. */
abstract part def 'USB Cable' :> USB, Cable {
  /** What kind of USB connector is on the start of the cable? */
  port start_connector: USB_In;
  /** What kind of USB connector is on the end of the cable? */
  port end_connector: USB_Out;
}
port def 'Output LED' :> LED;

// Safety-critical concepts.
abstract item def Voting;

// Artifacts specific to RDE.
abstract part def id CryptolSpec 'Cryptol System Specification'
 :> Cryptol, 'System Specification' {
   attribute literate: Boolean;
 }
attribute def Languages {
  attribute languages: String[*];
}
abstract part def id Impl 'Implementation' {
  attribute languages: Languages[*];
}
abstract part def id Software 'Software Implementation'
  :> Implementation;
abstract part def id SWImpl 'Hand-written Software Implementation'
  :> Software, 'Hand-written';
abstract part def id SynthSW 'Synthesized Software Implementation'
  :> Software, 'Machine-generated';
abstract part def 'Hardware Implementation';
abstract part def id HWImpl 'Hand-written Hardware Implementation';
abstract part def id SynthHW 'Synthesized Hardware Implementation';
abstract part def id Binary 'Software Binaries' {
  attribute verified_compilation: Boolean;
  attribute secure_compilation: Boolean;
  attribute isa: ISA;
}
part def RISCV_Binary :> Binary {
  // :>> isa = RISC_V_ISA;
}
abstract part def id Bitstream 'FPGA Bitstream' {
  attribute proprietary_flow: Boolean;
```

```
  }

  // NRC concepts.
  abstract part def 'NRC Certifcation Regulations';
}
```

[10]:
```
Comment (72a5c8cf-f4ce-40be-a97c-06f56917b8cb)
Package [Glossary] Project Glossary (74eb07c6-e4fb-4c57-bc4e-546894c000ec)
```

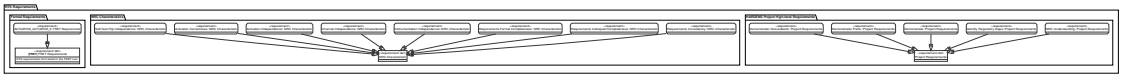
[11]:
```
/**
 * All requirements that the RTS system must fulfill, as driven by the
 * IEEE 603-2018 standards and the NRC RFP.
 */
package id Requirements 'RTS Requirements' {
  // Note that we do not specify documentation comments here as they
  // are specified in the Lando specification.  If we do not include
  // additional specifications here on the refinement from the higher-level
  // specification (in this case, SysML refines Lando), then the higher-level
  // specification's comments/specifications refine too (an hence are
  // just copied verbatim).
  package id Requirements 'HARDENS Project High-level Requirements' {
    import 'Project Glossary'::*;
    import 'RTS Stakeholders'::*;

    requirement def 'Project Requirements' {
      subject 'NRC staff' : 'NRC Customer';
    }
    requirement 'NRC Understanding' : 'Project Requirements';
    requirement 'Identify Regulatory Gaps' : 'Project Requirements';
    requirement Demonstrate : 'Project Requirements';
    requirement 'Demonstrator Parts' : 'Project Requirements';
    requirement 'Demonstrator Groundwork' : 'Project Requirements';
 }
 // @todo kiniry These requirements must be made consistent, or merged with
 // those found in the Characteristics specification.
 package id Characteristics 'NRC Characteristics' {
   import 'Project Glossary'::*;
   import 'RTS Stakeholders'::*;

   requirement def 'NRC Characteristic' {
     //subject expert: 'NRC Certification SME';
     //subject regulation: 'NRC Certification Regulations';
   }
   requirement 'Requirements Consistency' : 'NRC Characteristic';
   requirement 'Requirements Colloquial Completeness' : 'NRC Characteristic';
   requirement 'Requirements Formal Completeness' : 'NRC Characteristic';
```

```
    requirement 'Instrumentation Independence' : 'NRC Characteristic';
    requirement 'Channel Independence' : 'NRC Characteristic';
    requirement 'Actuation Independence' : 'NRC Characteristic';
    requirement 'Actuation Correctness' : 'NRC Characteristic';
    requirement 'Self-Test/Trip Independence' : 'NRC Characteristic';
  }
  // Note that formal requirements expressed externally must be traceable
  // to this system model, but need not be repeated in whole here.  Model
  // elements that are expressed in both the SysML and FRET models must
  // be in a refinement relationship with each other (e.g,, in this case study,
  // SysML  FRET.
  package 'Formal Requirements' {
    import 'Project Glossary'::*;
    requirement def id FRET 'FRET Requirements' {
      doc /* RTS requirements formalized in the FRET tool. */
    }
    // @todo kiniry Complete remaining FRET requirements.
    requirement ACTUATION_ACTUATOR_0 : FRET;
  }
}
```

[11]: Package [Requirements] RTS Requirements (360eb9b5-7d2c-449a-a87f-8bebe68b16d6)

[12]:
```
%viz "RTS Requirements"
```

[12]:



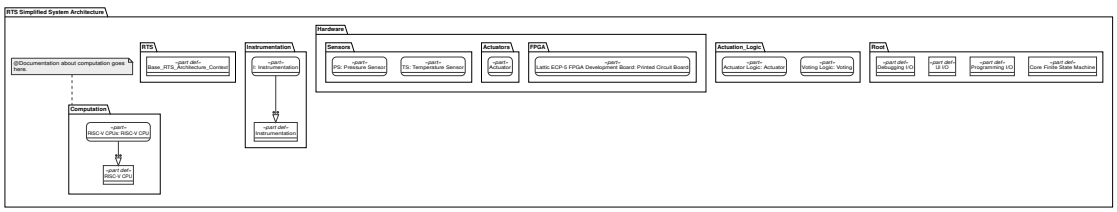
[38]:
```
package 'RTS Simplified System Architecture' {
  private import 'Project Glossary'::*;

  // doc id sys_arch_doc
  package Root {
    part def id CFSM 'Core Finite State Machine' :> FSM;
    part def id Programming_IO 'Programming I/O' :> IO;
    part def id UI_IO 'UI I/O' :> IO;
    part def id Debugging_IO 'Debugging I/O' :> IO;
  }
  package 'Actuation_Logic' {
    part 'Voting Logic': 'Voting'[2];
    part 'Actuator Logic': 'Actuator'[2];
  }
  /** Documentation about computation goes here. */
  package Computation {
```

```
    part def 'RISC-V CPU':> CPU, RISC_V_ISA;
    part id CPUs 'RISC-V CPUs': 'RISC-V CPU'[3];
  }
  package Hardware {
    package FPGA {
      part id DevBoard 'Lattic ECP-5 FPGA Development Board': PCB;
    }
    package Actuators {
      part 'Actuator'[2];
    }
    package Sensors {
      part TS: 'Temperature Sensor'[2];
      part PS: 'Pressure Sensor'[2];
    }
  }
  package Instrumentation {
    part def Instrumentation;
    part I: 'Instrumentation'[4];
  }

  package RTS {
    part def Base_RTS_Architecture_Context {
      // part ref
    }
  }
}
```

[38]:
```
Package RTS Simplified System Architecture
(973650cc-05e2-4650-822b-8c381c2b4f16)
```

[39]:
```
%viz "RTS Simplified System Architecture"
```

[39]:



[40]:
```
/** The physical hardware components that are a part of the HARDENS RTS
    demonstrator. */
package 'RTS Instrumentation Physical Architecture' {

  package TempSensor {
```

```
    import ISQThermodynamics::TemperatureValue;

    /** Generic temperature port */
    port def TemperatureOutPort {
        out temp : TemperatureValue;
    }

    /** Temperature interface */
    interface def TemperatureInterface {
      // each sensor is connected to two instrumentation units
      end supplierPort : TemperatureOutPort;
      end consumerPort1 : ~TemperatureOutPort;
      end consumerPort2 : ~TemperatureOutPort;
    }

    /** A sensor that is capable of measuring the temperature of its environment.
↪ */
    part def 'Temperature Sensor' {
      /** What is your temperature reading in Celcius (C)? */
      attribute currentTemp: TemperatureValue;
      port tempOut: TemperatureOutPort;
    }
  }

  package PressureSensor {
    import ISQMechanics::PressureValue;

    /** Generic pressure port */
    port def PressureOutPort {
        out pressure : PressureValue;
    }

    /** Pressure sensor interface */
    interface def PressureInterface {
      // each sensor is connected to two instrumentation units
      end supplierPort : PressureOutPort;
      end consumerPort1 : ~PressureOutPort;
      end consumerPort2 : ~PressureOutPort;
    }

    /** A sensor that is capable of measuring the air pressure of its␣
↪environment. */
    part def 'Pressure Sensor' {
      /** What is your pressure reading in Pascal (P)? */
      attribute currentPressure: PressureValue;
      port pressureOut: PressureOutPort;
    }
```

```
}

package Instrumentation {
  import ScalarValues::Real;
  import ScalarValues::Boolean;
  import TempSensor::*;
  import PressureSensor::*;

  port def TripPort {
    out trip : Boolean;
  }

  interface def TripInterface {
    // Each trip interface has 1 trip sources, and 1 consumer
    end supplierPort : TripPort;
    end consumerPort : ~TripPort;
  }


  part def InstrumentationUnit {
    // setpoints
    attribute tempSetpoint : TemperatureValue;
    attribute pressureSetpoint : PressureValue;
    attribute saturationLimit : Real;

    // mode selectors
    attribute maintenanceMode : Boolean;

    // Inputs
    port temperatureInput: ~TemperatureOutPort;
    port pressureInput: ~PressureOutPort;

    // Outputs
    port pressureTripOut:TripPort;
    port temperatureTripOut:TripPort;
    port saturationTripOut:TripPort;
  }
}

package Actuation {
  import Instrumentation::*;

  port def ActuationPort {
    out actuate: Boolean;
  }

  part def CoincidenceLogic {
```

```
    port channel1: ~TripPort;
    port channel2: ~TripPort;
    port actuate: ActuationPort;
  }
  part def OrLogic {
    port channel1: ~TripPort;
    port channel2: ~TripPort;
    port actuate: ActuationPort;
  }

 part def ActuationUnit {
   part tempLogic : CoincidenceLogic;
   part pressureLogic : CoincidenceLogic;
   part saturationLogic : OrLogic;
  }
  part def Actuator {
     port input: ActuationPort;
   }
  interface def OrGate {
    end inputA: ActuationPort;
    end inputB: ActuationPort;
    end output: ~ActuationPort;
  }
}

part RTS {
  import TempSensor::*;
  part tempSensor1 : 'Temperature Sensor';
  part tempSensor2 : 'Temperature Sensor';

  import PressureSensor::*;
  part pressureSensor1 : 'Pressure Sensor';
  part pressureSensor2 : 'Pressure Sensor';

  import Instrumentation::*;
  part InstrumentationUnit1 : InstrumentationUnit;
  part InstrumentationUnit2 : InstrumentationUnit;
  part InstrumentationUnit3 : InstrumentationUnit;
  part InstrumentationUnit4 : InstrumentationUnit;

  import Actuation::*;
  part ActuationUnit1: ActuationUnit;
  part ActuationUnit2: ActuationUnit;
  part Actuator1 : Actuator;
  part Actuator2 : Actuator;

  // connect sensors
```

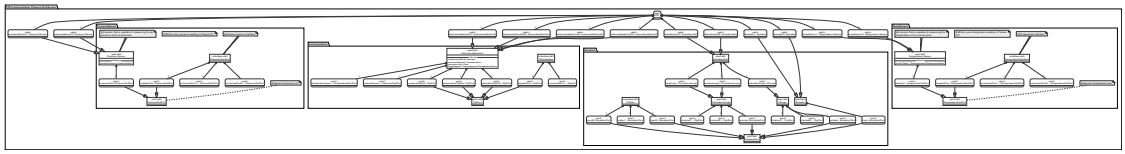
```
  interface T1 : TemperatureInterface
    connect tempSensor1.tempOut to InstrumentationUnit1.temperatureInput;
    connect tempSensor1.tempOut to InstrumentationUnit2.temperatureInput;

  interface T2 : TemperatureInterface
    connect tempSensor2.tempOut to InstrumentationUnit3.temperatureInput;
    connect tempSensor2.tempOut to InstrumentationUnit4.temperatureInput;

  interface P1 : PressureInterface
    connect pressureSensor1.pressureOut to InstrumentationUnit1.pressureInput;
    connect pressureSensor1.pressureOut to InstrumentationUnit2.pressureInput;

  interface P2 : PressureInterface
    connect pressureSensor2.pressureOut to InstrumentationUnit3.pressureInput;
    connect pressureSensor2.pressureOut to InstrumentationUnit4.pressureInput;

  // connect actuation logic
  // Temperature channels
  interface TripTemp1 : TripInterface
    connect InstrumentationUnit1.temperatureTripOut to ActuationUnit1.
↪tempLogic.channel1;
  interface TripTemp2 : TripInterface
    connect InstrumentationUnit2.temperatureTripOut to ActuationUnit1.
↪tempLogic.channel2;
  interface TripTemp3 : TripInterface
    connect InstrumentationUnit3.temperatureTripOut to ActuationUnit2.
↪tempLogic.channel1;
  interface TripTemp4 : TripInterface
    connect InstrumentationUnit4.temperatureTripOut to ActuationUnit2.
↪tempLogic.channel2;
  // Pressure channels
  interface TripPressure1 : TripInterface
    connect InstrumentationUnit1.pressureTripOut to ActuationUnit1.
↪pressureLogic.channel1;
  interface TripPressure2 : TripInterface
    connect InstrumentationUnit2.pressureTripOut to ActuationUnit1.
↪pressureLogic.channel2;
  interface TripPressure3 : TripInterface
    connect InstrumentationUnit3.pressureTripOut to ActuationUnit2.
↪pressureLogic.channel1;
  interface TripPressure4 : TripInterface
    connect InstrumentationUnit4.pressureTripOut to ActuationUnit2.
↪pressureLogic.channel2;
  // Saturation channels
  interface TripSaturation1 : TripInterface
```

```
      connect InstrumentationUnit1.saturationTripOut to ActuationUnit1.
↪saturationLogic.channel1;
    interface TripSaturation2 : TripInterface
      connect InstrumentationUnit2.saturationTripOut to ActuationUnit1.
↪saturationLogic.channel2;
    interface TripSaturation3 : TripInterface
      connect InstrumentationUnit3.saturationTripOut to ActuationUnit2.
↪saturationLogic.channel1;
    interface TripSaturation4 : TripInterface
      connect InstrumentationUnit4.saturationTripOut to ActuationUnit2.
↪saturationLogic.channel2;
    // Connect outputs to actuators
    interface SaturationOut : OrGate
      connect ActuationUnit1.saturationLogic.actuate to Actuator2.input;
      connect ActuationUnit2.saturationLogic.actuate to Actuator2.input;
    }
    // TODO : connect the remaining logic
}
```

[40]:
```
Package RTS Instrumentation Physical Architecture
(943a3e8f-d231-4604-8b7e-d1d2bba0398e)
```

[41]:
```
%viz "RTS Instrumentation Physical Architecture"
```

[41]:



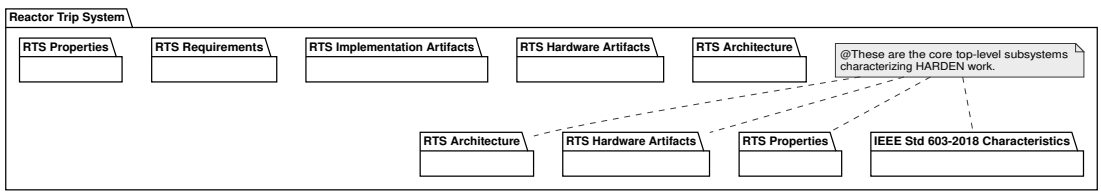
[18]:
```
/*
 # Reactor Trip System (RTS) High-assurance Demonstrator
 ## project: High Assurance Rigorous Digital Engineering for Nuclear Safety␣
 ↪(HARDENS)
 ### copyright (C) 2021 Galois
 ### author: Joe Kiniry <kiniry@galois.com>
*/

/**
 * The overall shape of the Reactor Trip System (RTS) is an archetypal
 * *sense-compute-actuate* architecture. Sensors are in the `Sensors`
 * subsystem. They are read by the `Instrumentation` subsystem, which
 * contains four separate and independent `Instrumentation`
 * components. The "Compute" part of the architecture is spread across
```

```
 * the `Actuation Logic` subsystem-which contains the two `Voting`
 * components which perform the actuation logic itself-and the `Root`
 * subsystem which contains the core computation and I/O components, and
 * the two separate and independent devices that drive actuators.
 */
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. */
}
```

[18]:
```
Comment (8aa95680-ec61-4251-b8d0-65242c9e4777)
Package [RTS] Reactor Trip System (2004b5b3-0d8e-4074-ba1f-70830c2262f1)
```

[19]:
```
%viz "Reactor Trip System"
```

[19]:



[20]:
```
/**
 * The set of all atomic external or internal actions that the RTS
 * system can take.  Note that every scenario must be describable by
 * a sequence of actions.
 */
package id Actions 'RTS Actions' {
```

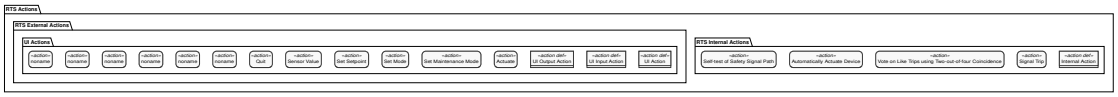
```
package id Internal 'RTS Internal Actions' {
  action def id IA 'Internal Action' {
    doc /* Actions internal to the RTS */
  }
  action id Trip 'Signal Trip' : IA {
    in item division;
    in item channel;
  }
  action id Vote 'Vote on Like Trips using Two-out-of-four Coincidence' : IA {
    in item divisions[2];
    in item channel;
  }
  action id A 'Automatically Actuate Device' : IA {
    in item device;
  }
  action id T 'Self-test of Safety Signal Path' : IA;
}
package id External 'RTS External Actions' {
  package 'UI Actions' {
    action def id UIA 'UI Action' {
      doc /* Actions exhibited by the RTS UI, either inbound or outbound. */
    }
    // @design kiniry These should both be specializations of UIA, but I
    // don't know how to write that in SysML yet.
    action def id UI_IA 'UI Input Action';
    action def id UI_OA 'UI Output Action';

    // Input actions.
    // @todo kiniry Add features to correspond to UI parameters.
    action id A Actuate : UI_IA {
      in item actuator;
      in item on_off;
    }
    action id M 'Set Maintenance Mode' :  UI_IA {
      in item division;
      in item on_off;
    }
    action id B 'Set Mode' : UI_IA {
      in item division;
      in item trip_mode;
    }
    action id S 'Set Setpoint' : UI_IA {
      in item division;
      in item channel;
      in item value;
    }
    action id V 'Sensor Value' : UI_IA {
```

```
        doc /* Simulate a sensor reading */
        in item division;
        in item channel;
        in item value;
      }
      action id Q 'Quit' : UI_IA;

      // Output actions.
      action id 'Display Pressure' : UI_OA;
      action id 'Display Temperature' : UI_OA;
      action id 'Display Saturation Margin': UI_OA;
      action id 'Display Trip Ouput Signal State': UI_OA;
      action id 'Display Indication Of Channel in Bypass': UI_OA;
      action id 'Display Actuation State': UI_OA;
    }
  }
}
```

[20]: Package [Actions] RTS Actions (a042f3e7-b0a1-4a80-ad1b-20e0b52d1797)

[21]:
```
%viz "RTS Actions"
```

[21]:



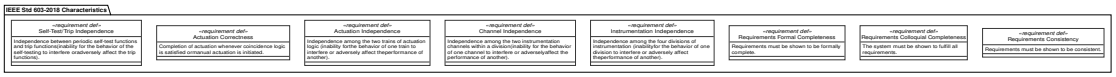
[22]:
```
/**
 * The IEEE 603-2018 requirements (known as "characteristics" in
 * the standard) which the RTS demonstrator system must fulfill.
 */
package id Characteristics 'IEEE Std 603-2018 Characteristics' {
  requirement def 'Requirements Consistency' {
    doc /* Requirements must be shown to be consistent. */
  }
  requirement def 'Requirements Colloquial Completeness' {
    doc /* The system must be shown to fulfill all requirements. */
  }
  requirement def 'Requirements Formal Completeness' {
    doc /* Requirements must be shown to be formally complete. */
  }
  requirement def 'Instrumentation Independence' {
    doc /* Independence among the four divisions of instrumentation (inability
           for the behavior of one division to interfere or adversely affect the
           performance of another). */
```

```
  }
  requirement def 'Channel Independence' {
    doc /* 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). */
  }
  requirement def 'Actuation Independence' {
    doc /* Independence among the two trains of actuation logic (inability for
           the behavior of one train to interfere or adversely affect the
           performance of another). */
  }
  requirement def 'Actuation Correctness' {
    doc /* Completion of actuation whenever coincidence logic is satisfied or
           manual actuation is initiated. */
  }
  requirement def 'Self-Test/Trip Independence' {
    doc /* 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). */
  }
}
```

[22]: Package [Characteristics] IEEE Std 603-2018 Characteristics (b3398d4f-934f-41dc-a806-cd60b1236dfa)

[24]:
```
%viz "IEEE Std 603-2018 Characteristics"
```

[24]:



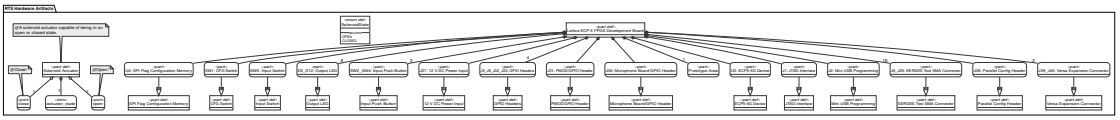
[43]:
```
/**
 * The physical hardware components that are a part of the HARDENS RTS
 * demonstrator.
 */
package 'RTS Hardware Artifacts' {
  private import 'Project Glossary'::*;
  import 'RTS Instrumentation Physical Architecture'::*;
  private import ScalarValues::*;

  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;
    part J38 : 'Parallel Config Header' subsets components;
    part J39_J40 : 'Versa Expansion Connector'[2] subsets components;
    part U4 : 'SPI Flag Configuration Memory' subsets components;
    part SW1 : 'CFG Switch' subsets components;
    part SW5 : 'Input Switch' subsets components;
    part D5_D12 : 'Output LED'[8] subsets components;
    part SW2_SW4 : 'Input Push Button'[3] subsets components;
    part J37 : '12 V DC Power Input' subsets components;
    part J5_J8_J32_J33 : 'GPIO Headers'[4] subsets components;
    part J31 : 'PMOD/GPIO Header' subsets components;
    part J30 : 'Microphone Board/GPIO Header' subsets components;
    part 'Prototype Area';
    part U3 : 'ECP5-5G Device' subsets components;
    part J1 : 'JTAG Interface' subsets components;
    part J2 : 'Mini USB Programming' subsets components;
  }

  enum def SolenoidState {
    OPEN;
    CLOSED;
  }
  /** A solenoid actuator capable of being in an open or closed state. */
  part def 'Solenoid Actuator':> Actuator {
    item actuator_state;
    /** Open! */
    port open;
    /** Close! */
    port close;
  }
}
```

[43]: Package RTS Hardware Artifacts (f29e2a46-7354-43f4-94cf-cacd8305b17e)

[44]:
```
%viz "RTS Hardware Artifacts"
```

[44]:

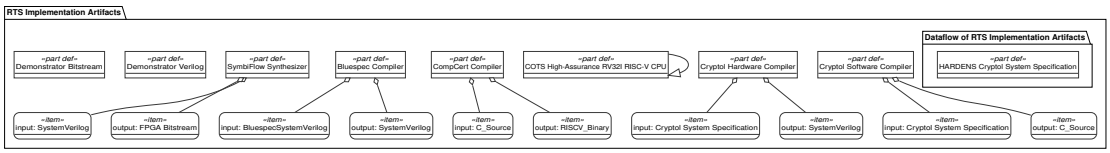


[28]:
```
package id Artifacts 'RTS Implementation Artifacts' {
  private import ScalarValues::*;
  private import 'Project Glossary'::*;

  // @design Remove concepts in general Glossary that duplicate or
  // overlap with these concepts.  Move abstract items to Glossary.
  part def id CryptolToC 'Cryptol Software Compiler':> Compiler {
    ref item input: CryptolSpec redefines input;
    ref item output: C_Source redefines output;
  }
  part def id CryptolToSystemVerilog 'Cryptol Hardware Compiler':> Compiler {
    ref item input: CryptolSpec redefines input;
    ref item output: SystemVerilog redefines output;
  }
  part def id CPU 'COTS High-Assurance RV32I RISC-V CPU':> CPU, RISC_V_ISA;
  part def id CompCert 'CompCert Compiler':> Compiler {
    ref item input: C_Source redefines input;
    ref item output: RISCV_Binary redefines output;
  }
  part def id BSC 'Bluespec Compiler':> Compiler {
    ref item input: BluespecSystemVerilog redefines input;
    ref item output: SystemVerilog redefines output;
  }
  part def id SymbiFlow 'SymbiFlow Synthesizer' :> Synthesizer {
    ref item input: SystemVerilog redefines input;
    ref item output: Bitstream redefines output;
  }
  part def id RTL 'Demonstrator Verilog';
  part def 'Demonstrator Bitstream':> Bitstream;
  package id Dataflow 'Dataflow of RTS Implementation Artifacts' {
    private import 'RTS Implementation Artifacts'::*;

    part def 'HARDENS Cryptol System Specification':> CryptolSpec {
     // :>> literate = true;
    }
    // bind 'HARDENS Cryptol System Specification'.output = CryptolToC.input;
  }
}
```

[28]: Package [Artifacts] RTS Implementation Artifacts
(47e22895-4eeb-409f-813c-7b1d417ae8eb)

[29]:
```
%viz "RTS Implementation Artifacts"
```

[29]:



[31]:
```
/**
 * This RTS architecture specification includes all of the core
 * concepts inherent to NPP Instrumentation and Control systems.
 * A system architecture specification often includes a software,
 * hardware, network, and data architecture specifications.
 */
package id Architecture 'RTS Architecture' {
  //import RTS::*;
  //import 'Project Glossary'::*;
  //import Artifacts::*;
  //import 'RTS Hardware Artifacts'::*;

  part def Base_RTS_System_Architecture_Context {
    // overall RTS architecture shape from
  }

  /**
   * Note that this is the *systems* architecture, which is different
   * than our software, hardware, or data architectures.
   */
  package id RTS_System_Arch 'RTS System Architecture' {

    package Sensor {
      private import Quantities::*;

      /** Generic sensor port */
      port def SensorOutPort {
          out value : ScalarQuantityValue;
      }

      /** Generic sensor */
      part def GenericSensor {
        attribute currentValue : ScalarQuantityValue;
        attribute sensorAddress : ScalarValues::Integer;
```

```
    port output: SensorOutPort;
  }

  /**
   * A demultiplexer for sending one sensor signal to multiple
   * outputs.
   */
  part def Demux {
    port input: ~SensorOutPort;
    // Using vector notation doesn't seem to work in connections
    port output1: SensorOutPort;
    port output2: SensorOutPort;
  }
}

/** A generic temperature sensor. */
package TempSensor {
  import Sensor::*;
  import ISQThermodynamics::TemperatureValue;

  /** Temperature port */
  port def TemperatureOutPort :> SensorOutPort {
      redefines value: TemperatureValue;
  }

  /** A sensor that is capable of measuring the temperature of its␣
↪environment. */
  part def 'Temperature Sensor' :> GenericSensor {
    /** What is your temperature reading in Celsius (C)? */
    redefines currentValue: TemperatureValue;
    redefines output: TemperatureOutPort;
  }

  part def TempDemux :> Demux {
    redefines input: ~TemperatureOutPort;
    redefines output1: TemperatureOutPort;
    redefines output2: TemperatureOutPort;
  }
}

/** A generic pressure sensor. */
package PressureSensor {
  import Sensor::*;
  import ISQMechanics::PressureValue;

  /** Pressure port */
  port def PressureOutPort :> SensorOutPort {
```

```
        redefines value: PressureValue;
    }

    /** A sensor that is capable of measuring the air pressure of its␣
↪environment. */
    part def 'Pressure Sensor' :> GenericSensor {
      /** What is your pressure reading in Pascal (P)? */
      redefines currentValue: PressureValue;
      redefines output: PressureOutPort;
    }

    part def PressureDemux :> Demux {
      redefines input: ~PressureOutPort;
      redefines output1: PressureOutPort;
      redefines output2: PressureOutPort;
    }
  }

  /**
   * The Instrumentation subsystem contains all of the sensors for an
   * NPP I&C system.
   */
  package Instrumentation {
    private import ScalarValues::Real;
    private import ScalarValues::Boolean;
    private import TempSensor::*;
    private import PressureSensor::*;

    port def TripPort {
      out trip : Boolean;
    }

    port def BypassPort {
      out trip : Boolean;
    }

    enum def TripMode {
      enum Bypass;
      enum Operate;
      enum Manual;
    }

    enum def Channel {
      enum Temperature;
      enum Pressure;
      enum Saturation;
    }
```

```
    attribute def TripModeCommand {
      attribute mode: TripMode;
      attribute channel: Channel;
    }

    port def TripModePort {
      out mode: TripModeCommand;
    }

    part def PressureTable {
             doc /* The saturation table gives us the saturation pressure␣
↪margin given
                  * the current temperature and pressure, hence the output is␣
↪pressure
                  */
             port temperatureInput: ~TemperatureOutPort;
             port pressureInput: ~PressureOutPort;
            port saturationMarginOutput: PressureOutPort;
    }

    part def InstrumentationUnit {
      // setpoints
      attribute tempSetpoint : TemperatureValue;
      attribute pressureSetpoint : PressureValue;
      attribute saturationLimit : Real;

      // mode selectors
      attribute maintenanceMode : Boolean;
      attribute temperatureTripMode: TripMode;
      attribute pressureTripMode: TripMode;
      attribute saturationTripMode: TripMode;


      // Inputs
      port temperatureInput: ~TemperatureOutPort;
      port pressureInput: ~PressureOutPort;
      port saturationMarginInput: ~PressureOutPort;
      port tripMode: ~TripModePort;

      // Outputs
      port pressureTripOut:TripPort;
      port temperatureTripOut:TripPort;
      port saturationTripOut:TripPort;

      port setMaintenanceMode: ~EventControl::MaintenancePort;
```

```
    port newTemperatureSetpoint: ~TemperatureOutPort;
    port newPressureSetpoint: ~PressureOutPort;
    port newSaturationSetpoint : ~SensorOutPort;
  }
}

package Actuation {
  import Instrumentation::*;

  port def ActuationPort {
    out actuate: ScalarValues::Boolean;
  }

  part def CoincidenceLogic {
    port channel1: ~TripPort;
    port channel2: ~TripPort;
    port channel3: ~TripPort;
    port channel4: ~TripPort;
    port actuate: ActuationPort;
  }
  part def OrLogic {
    port channel1: ~TripPort;
    port channel2: ~TripPort;
    port actuate: ActuationPort;
  }

 part def ActuationUnit {
   part temperatureLogic : CoincidenceLogic;
   part pressureLogic : CoincidenceLogic;
   part saturationLogic : CoincidenceLogic;

   part tempPressureTripOut: OrLogic;

   connect temperatureLogic.actuate to tempPressureTripOut.channel1;
   connect pressureLogic.actuate to tempPressureTripOut.channel2;
  }
  part def Actuator {
     // Actuate if either of these are true
     port input: ActuationPort;
     port manualActuatorInput: ~ActuationPort;
   }
}

package EventControl {
import ScalarValues::Boolean;

  port def MaintenancePort {
```

```
    out maintenance: Boolean;
  }

  part def ControlUnit {
    // Maintenance mode select x 4 instrumentation units
    port maintenanceMode: MaintenancePort[4];
    // Trip mode select x 4 instrumentation units
    port tripMode: Instrumentation::TripModePort[4];
    // New setpoints x 4 instrumentation units
    port newPressureSetpoint: PressureSensor::PressureOutPort[4];
    port newTemperatureSetpoint: TempSensor::TemperatureOutPort[4];
    port newSaturationSetpoint: PressureSensor::PressureOutPort[4];
    // Toggle actuator x2 actuators
    port manualActuatorInput: Actuation::ActuationPort[2];
  }
}

part RTS {
  part eventControl : EventControl::ControlUnit;

  import Instrumentation::*;
  part instrumentationAndSensing {
    part pressureSensor1 : PressureSensor::'Pressure Sensor';
    part pressureSensor2 : PressureSensor::'Pressure Sensor';

    part tempSensor1 : TempSensor::'Temperature Sensor';
    part tempSensor2 : TempSensor::'Temperature Sensor';

    part instrumentationUnit1 : InstrumentationUnit;
    part instrumentationUnit2 : InstrumentationUnit;
    part instrumentationUnit3 : InstrumentationUnit;
    part instrumentationUnit4 : InstrumentationUnit;

    part table1 : PressureTable;
    part table2 : PressureTable;
    part table3 : PressureTable;
    part table4 : PressureTable;

    part tempDemux1 : TempSensor::Demux;
    part tempDemux2 : TempSensor::Demux;

    part pressureDemux1 : PressureSensor::Demux;
    part pressureDemux2 : PressureSensor::Demux;

    // Temp sensor 1
    connect tempSensor1.output to tempDemux1.input;
    connect tempDemux1.output1 to instrumentationUnit1.temperatureInput;
```

```
        connect tempDemux1.output2 to instrumentationUnit2.temperatureInput;

        // Temp sensor 2
        connect tempSensor2.output to tempDemux2.input;
        connect tempDemux2.output1 to instrumentationUnit3.temperatureInput;
        connect tempDemux2.output2 to instrumentationUnit4.temperatureInput;

        // Pressure sensor 1
        connect pressureSensor1.output to pressureDemux1.input;
        connect pressureDemux1.output1 to instrumentationUnit1.pressureInput;
        connect pressureDemux1.output2 to instrumentationUnit2.pressureInput;

        // Pressure sensor 2
        connect pressureSensor2.output to pressureDemux2.input;
        connect pressureDemux1.output1 to instrumentationUnit3.pressureInput;
        connect pressureDemux1.output2 to instrumentationUnit4.pressureInput;

        // Saturation Margin Calculation
        connect pressureDemux1.output1 to table1.pressureInput;
        connect tempDemux1.output1 to table1.temperatureInput;
        connect table1.saturationMarginOutput to instrumentationUnit1.
↪saturationMarginInput;

        connect pressureDemux1.output2 to table2.pressureInput;
        connect tempDemux1.output2 to table2.temperatureInput;
        connect table2.saturationMarginOutput to instrumentationUnit2.
↪saturationMarginInput;

        connect pressureDemux2.output1 to table3.pressureInput;
        connect tempDemux2.output1 to table3.temperatureInput;
        connect table3.saturationMarginOutput to instrumentationUnit3.
↪saturationMarginInput;

        connect pressureDemux2.output2 to table4.pressureInput;
        connect tempDemux2.output2 to table4.temperatureInput;
        connect table4.saturationMarginOutput to instrumentationUnit4.
↪saturationMarginInput;
    }


    import Actuation::*;
    part actuation {
      part actuationUnit1: ActuationUnit;
      part actuationUnit2: ActuationUnit;

      part actuator1 : Actuator;
      part actuator2 : Actuator;
```

```
        part actuateActuator1: OrLogic;
        part actuateActuator2: OrLogic;

        // connect actuators
        // Actuator 1 - temp or pressure trip
        connect actuationUnit1.tempPressureTripOut.actuate to actuateActuator1.
↪channel1;
        connect actuationUnit2.tempPressureTripOut.actuate to actuateActuator1.
↪channel2;
        connect actuateActuator1.actuate to actuator1.input;

        // Actuator 2 - Saturation
        connect actuationUnit1.saturationLogic.actuate to actuateActuator2.
↪channel1;
        connect actuationUnit2.saturationLogic.actuate to actuateActuator2.
↪channel2;
        connect actuateActuator2.actuate to actuator2.input;
    }

    // connect Control units
    // Actuators manual override
    connect eventControl.manualActuatorInput[1] to actuation.actuator1.
↪manualActuatorInput;
    connect eventControl.manualActuatorInput[2] to actuation.actuator2.
↪manualActuatorInput;

    // Instrumentation mode select
    connect eventControl.maintenanceMode[1] to instrumentationAndSensing.
↪instrumentationUnit1.setMaintenanceMode;
    connect eventControl.maintenanceMode[2] to instrumentationAndSensing.
↪instrumentationUnit2.setMaintenanceMode;
    connect eventControl.maintenanceMode[3] to instrumentationAndSensing.
↪instrumentationUnit3.setMaintenanceMode;
    connect eventControl.maintenanceMode[4] to instrumentationAndSensing.
↪instrumentationUnit4.setMaintenanceMode;


    // Instrumentation pressure setpoint
    connect eventControl.newPressureSetpoint[1] to instrumentationAndSensing.
↪instrumentationUnit1.newPressureSetpoint;
    connect eventControl.newPressureSetpoint[2] to instrumentationAndSensing.
↪instrumentationUnit2.newPressureSetpoint;
    connect eventControl.newPressureSetpoint[3] to instrumentationAndSensing.
↪instrumentationUnit3.newPressureSetpoint;
```

```
    connect eventControl.newPressureSetpoint[4] to instrumentationAndSensing.
↪instrumentationUnit4.newPressureSetpoint;

    // Instrumentation temperature setpoint
    connect eventControl.newTemperatureSetpoint[1] to␣
↪instrumentationAndSensing.instrumentationUnit1.newTemperatureSetpoint;
    connect eventControl.newTemperatureSetpoint[2] to␣
↪instrumentationAndSensing.instrumentationUnit2.newTemperatureSetpoint;
    connect eventControl.newTemperatureSetpoint[3] to␣
↪instrumentationAndSensing.instrumentationUnit3.newTemperatureSetpoint;
    connect eventControl.newTemperatureSetpoint[4] to␣
↪instrumentationAndSensing.instrumentationUnit4.newTemperatureSetpoint;

    // Instrumentation saturation setpoint
    connect eventControl.newSaturationSetpoint[1] to instrumentationAndSensing.
↪instrumentationUnit1.newSaturationSetpoint;
    connect eventControl.newSaturationSetpoint[2] to instrumentationAndSensing.
↪instrumentationUnit2.newSaturationSetpoint;
    connect eventControl.newSaturationSetpoint[3] to instrumentationAndSensing.
↪instrumentationUnit3.newSaturationSetpoint;
    connect eventControl.newSaturationSetpoint[4] to instrumentationAndSensing.
↪instrumentationUnit4.newSaturationSetpoint;

    // Instrumentation trip mode
    // Bypass temperature
    connect eventControl.tripMode[1] to instrumentationAndSensing.
↪instrumentationUnit1.tripMode;
    connect eventControl.tripMode[2] to instrumentationAndSensing.
↪instrumentationUnit2.tripMode;
    connect eventControl.tripMode[3] to instrumentationAndSensing.
↪instrumentationUnit3.tripMode;
    connect eventControl.tripMode[4] to instrumentationAndSensing.
↪instrumentationUnit4.tripMode;

    // Trip on pressure above the setpoint
    // Actuation unit 1
    connect instrumentationAndSensing.instrumentationUnit1.pressureTripOut to␣
↪actuation.actuationUnit1.pressureLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.pressureTripOut to␣
↪actuation.actuationUnit1.pressureLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.pressureTripOut to␣
↪actuation.actuationUnit1.pressureLogic.channel3;
    connect instrumentationAndSensing.instrumentationUnit4.pressureTripOut to␣
↪actuation.actuationUnit1.pressureLogic.channel4;
    // Actuation unit 2
```

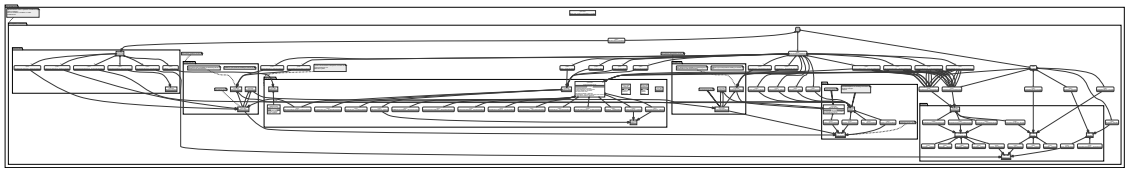
```
    connect instrumentationAndSensing.instrumentationUnit1.pressureTripOut to␣
↪actuation.actuationUnit2.pressureLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.pressureTripOut to␣
↪actuation.actuationUnit2.pressureLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.pressureTripOut to␣
↪actuation.actuationUnit2.pressureLogic.channel3;
    connect instrumentationAndSensing.instrumentationUnit4.pressureTripOut to␣
↪actuation.actuationUnit2.pressureLogic.channel4;

    // Trip on temperature above the setpoint
    // Actuation unit 1
    connect instrumentationAndSensing.instrumentationUnit1.temperatureTripOut␣
↪to actuation.actuationUnit1.temperatureLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.temperatureTripOut␣
↪to actuation.actuationUnit1.temperatureLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.temperatureTripOut␣
↪to actuation.actuationUnit1.temperatureLogic.channel3;
    connect instrumentationAndSensing.instrumentationUnit4.temperatureTripOut␣
↪to actuation.actuationUnit1.temperatureLogic.channel4;
    // Actuation unit 2
    connect instrumentationAndSensing.instrumentationUnit1.temperatureTripOut␣
↪to actuation.actuationUnit2.temperatureLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.temperatureTripOut␣
↪to actuation.actuationUnit2.temperatureLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.temperatureTripOut␣
↪to actuation.actuationUnit2.temperatureLogic.channel3;
    connect instrumentationAndSensing.instrumentationUnit4.temperatureTripOut␣
↪to actuation.actuationUnit2.temperatureLogic.channel4;

    // Trip on saturation above the setpoint
    // Actuation unit 1
    connect instrumentationAndSensing.instrumentationUnit1.saturationTripOut␣
↪to actuation.actuationUnit1.saturationLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.saturationTripOut␣
↪to actuation.actuationUnit1.saturationLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.saturationTripOut␣
↪to actuation.actuationUnit1.saturationLogic.channel3;
    connect instrumentationAndSensing.instrumentationUnit4.saturationTripOut␣
↪to actuation.actuationUnit1.saturationLogic.channel4;
    // Actuation unit 2
    connect instrumentationAndSensing.instrumentationUnit1.saturationTripOut␣
↪to actuation.actuationUnit2.saturationLogic.channel1;
    connect instrumentationAndSensing.instrumentationUnit2.saturationTripOut␣
↪to actuation.actuationUnit2.saturationLogic.channel2;
    connect instrumentationAndSensing.instrumentationUnit3.saturationTripOut␣
↪to actuation.actuationUnit2.saturationLogic.channel3;
```

```
      connect instrumentationAndSensing.instrumentationUnit4.saturationTripOut␣
 ↪to actuation.actuationUnit2.saturationLogic.channel4;


    }
  } // package id RTS_System_Arch 'RTS System Architecture'
} // package id Architecture 'RTS Architecture'
```

[31]:
```
Package [Architecture] RTS Architecture (160a1b7f-aa0e-486c-88f5-83f972306fad)
```

[32]:
```
%viz "RTS Architecture"
```

[32]:



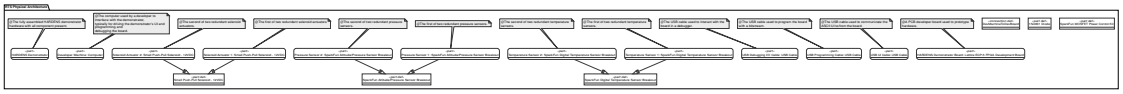
[57]:
```
///** The physical architecture of the HARDENS RTS demonstrator. */
package 'RTS Physical Architecture' {
  import 'Project Glossary'::*;
  import 'RTS Hardware Artifacts'::*;

  /** A PCB developer board used to prototype hardware. */
  part 'HARDENS Demonstrator Board' : DevBoard;
      /** The USB cable used to communicate the ASCII UI to/from the board. */
  part id UI_C 'USB UI Cable' : 'USB Cable';
  /** The USB cable used to program the board with a bitstream. */
  part id Prog_C 'USB Programming Cable' : 'USB Cable';
  /** The USB cable used to interact with the board in a debugger. */
  part id Debug_C 'USB Debugging I/O Cable' : 'USB Cable';
  // @trace #11 https://github.com/GaloisInc/HARDENS/issues/11
  // @todo Add attributes for URL traceability.
  part def id MPL3115A2 'SparkFun Altitude/Pressure Sensor Breakout' :>
    PCB, 'Pressure Sensor';
  // 4x https://www.sparkfun.com/products/11084
  part def 'SparkFun MOSFET Power Control Kit' :> PCB, Power;
  // 4x https://www.sparkfun.com/products/12959
  part def id TMP102 'SparkFun Digital Temperature Sensor Breakout' :>
    PCB, 'Temperature Sensor';
  // 4x https://www.sparkfun.com/products/13314
  part def 'Small Push-Pull Solenoid - 12VDC' :> 'Solenoid Actuator';
  // 4x https://www.adafruit.com/product/412
  part def '1N4001 Diode';
```

```
  // 1x https://www.adafruit.com/product/755
  /** The first of two redundant temperature sensors. */
  part id TS1 'Temperature Sensor 1' : TMP102;
  /** The second of two redundant temperature sensors. */
  part id TS2 'Temperature Sensor 2' : TMP102;
  /** The first of two redundant pressure sensors. */
  part id PS1 'Pressure Sensor 1' : MPL3115A2;
  /** The second of two redundant pressure sensors. */
  part id PS2 'Pressure Sensor 2' : MPL3115A2;
  /** The first of two redundant solenoid actuators. */
  part id SA1 'Solenoid Actuator 1' : 'Small Push-Pull Solenoid - 12VDC';
  /** The second of two redundant solenoid actuators. */
  part id SA2 'Solenoid Actuator 2' : 'Small Push-Pull Solenoid - 12VDC';
  // @todo kiniry Add ports for external connectors.

  /** The computer used by a developer to interface with the demonstrator,
      typically for driving the demonstrator's UI and programming and
      debugging the board. */
  part 'Developer Machine' : Computer;

  /** The fully assembled HARDENS demonstrator hardware with all component␣
↪present. */
  part id Demonstrator 'HARDENS Demonstrator';

  connection def DevMachineToDevBoard {
    end: Computer;
    end: PCB;
  }
  // connection: DevMachineToDevBoard connect 'Developer Machine' to DevBoard;
}
```

[57]: Package RTS Physical Architecture (201c1cd4-38c8-48b2-8f96-cf861831a1f5)

[58]:
```
%viz "RTS Physical Architecture"
```

[58]:



[33]:
```
/**
 * The set of all scenarios that describe interesting end-to-end executions
 * of the RTS system.  The full set of scenarios must include all normal
 * behavior (online and during self-test) and exceptional behavior.
 */
```

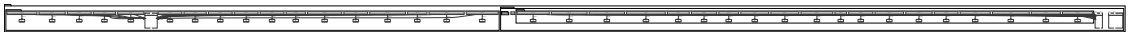
```
package id Scenarios 'RTS Scenarios' {
  package id Normal 'RTS Normal Behavior Scenarios' {
    import 'RTS Architecture'::'RTS System Architecture'::RTS;

    item def 'RTS User';
    use case def id NB 'Normal Behavior' {
      subject RTS;
      actor user : 'RTS User';
      objective {
        doc /* @see test_scenarios.lando */
      }
    }
    // @design kiniry Actually, ST should specialize NB.  I don't know how
    // to express that in SysML yet.
    use case def id ST 'Normal Behavior Under Self-Test' {
        subject RTS;
        actor tester : 'RTS User';
    }
    package 'Self-test Scenarios' {
      import Normal::ST;
      use case '1a - Trip on Mock High Pressure Reading from that Pressure␣
↪Sensor' : NB;
      use case '1b - Trip on Environmental High Pressure Reading from that␣
↪Pressure Sensor' : NB;
      use case '2a - Trip on Mock High Temperature Reading from that Temperature␣
↪Sensor' : NB;
      use case '2a - Trip on Environmental High Temperature Reading from that␣
↪Temperature Sensor' : NB;
      use case '3a - Trip on Mock Low Saturation Margin' : NB;
      use case '3a - Trip on Environmental Low Saturation Margin' : NB;
      use case '4 - Vote on Every Possible Like Trip' : NB;
      use case '5a - Automatically Actuate All Mock Devices in Sequence' : NB;
      use case '5b - Automatically Actuate All Mock Devices in Sequence' : NB;
      use case '6 - Manually Actuate Each Device in Sequence' : NB;
      use case '7a - Select Maintenance Operating Mode for each Division' : NB;
      use case '7b - Select Normal Operating Mode for each Division' : NB;
      use case '8 - Perform Each Kind of Setpoint Adjustment' : NB;
      use case '9 - Configure Bypass of Each Instrument Channel in Sequence' :␣
↪NB;
      use case '10 - Configure Active Trip Output State of Each Instrument␣
↪Channel in Sequence' : NB;
      use case '11 - Display Pressure, Temperature, and Saturation Margin' : NB;
      use case '13 - Display Indication of Every Channel in Bypass in Sequence' :
↪ NB;
      use case '14 - Demonstrate Periodic Continual Self-test of Safety Signal␣
↪Path' : NB;
```

```
      use case 'Full Self-Test' : NB;
    }
    package 'RTS Scenarios' {
      // @todo Still need to enumerate these.
    }
  }
  package id Exceptional 'RTS Exceptional Behavior Scenarios' {
    use case def id EB 'Exceptional Behavior' {
      subject RTS;
      objective {
        doc /* @see test_scenarios.lando */
      }
    }
    use case '1a - Cause Actuator 1 to Fail' : EB;
    use case '1b - Cause Actuator 2 to Fail' : EB;
    use case '1c - Non-determinisitically Cause an Actuator to Eventually Fail' :
↪ EB;
    use case '2a - Cause Temperature Sensor 1 to Fail' : EB;
    use case '2b - Cause Temperature Sensor 2 to Fail' : EB;
    use case '2c - Non-deterministically Cause a Temperature Sensor to␣
↪Eventually Fail' : EB;
    use case '3a - Cause Pressure Sensor 1 to Fail' : EB;
    use case '3b - Cause Pressure Sensor 2 to Fail' : EB;
    use case '3c - Non-deterministically Cause a Pressure Sensor to Eventually␣
↪Fail' : EB;
    use case '4a - Cause Instrumentation Unit 1 to Fail' : EB;
    use case '4b - Cause Instrumentation Unit 2 to Fail' : EB;
    use case '4c - Cause Instrumentation Unit 3 to Fail' : EB;
    use case '4d - Cause Instrumentation Unit 4 to Fail' : EB;
    use case '4e - Non-Deterministically Cause Instrumentation Unit to␣
↪Eventually Fail' : EB;
    // @review kiniry I actually don't know if we are fault tolerant to' : EB;
    // failure in these components.  Please review @abakst.
    use case '5a - Cause Temperature Demultiplexor 1 to Fail' : EB;
    use case '5b - Cause Temperature Demultiplexor 2 to Fail' : EB;
    use case '5b - Cause a Temperature Demultiplexor to Eventualy Fail' : EB;
  }
}
```

[33]: Package [Scenarios] RTS Scenarios (e5ae5f58-0ab2-4d0f-9f17-a443eb366a20)

[34]:
```
%viz "RTS Scenarios"
```

[34]:

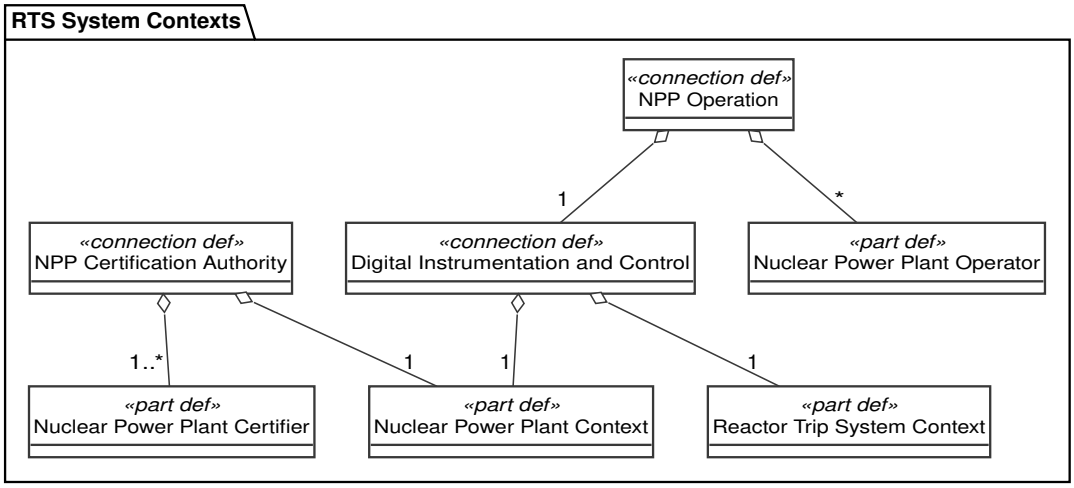


[36]:
```
package 'RTS System Contexts' {
  import 'RTS Architecture'::*;
  import 'Kiniry RTS System Architecture Draft'::*;

  // This specification is meant to frame the RTS in the larger context of
  // deployment into a Nuclear Power Plant.

  // Definitions of system contexts.

  // Introduce the RTS context.
  part def 'Reactor Trip System Context';
  alias RTSC for 'Reactor Trip System Context';

  // Introduce the NPP context.
  part def 'Nuclear Power Plant Context';
  alias NPPC for 'Nuclear Power Plant Context';

  part def 'Nuclear Power Plant Operator';
  part def 'Nuclear Power Plant Certifier';

  // Definitions of relevant connections
  connection def 'Digital Instrumentation and Control' {
    end: RTSC[1];
    end: NPPC[1];
  }
  alias DIandC for 'Digital Instrumentation and Control';

  connection def 'NPP Operation' {
    end: 'Nuclear Power Plant Operator'[*];
    end: DIandC[1];
  }

  connection def 'NPP Certification Authority' {
    end: 'NPPC'[1];
    end: 'Nuclear Power Plant Certifier'[1..*];
  }
}
```

[36]: Package RTS System Contexts (9b2d0afe-99ce-4068-900f-897491e4d45b)

[37]:
```
%viz "RTS System Contexts"
```

[37]:


RTS System Contexts
«connection def»
NPP Operation
1
*
«connection def»
NPP Certification Authority
«connection def»
Digital Instrumentation and Control
«part def»
Nuclear Power Plant Operator
1..*
1
1
1
«part def»
Nuclear Power Plant Certifier
«part def»
Nuclear Power Plant Context
«part def»
Reactor Trip System Context