Redundant Sensors: A Technical Report

Danielle Stewart and Todd Carpenter

{danielle, todd}@galois.com, Galois, Inc.

Abstract

In this technical report, we investigate a redundancy and fault management subsystem frequently used in mission-critical systems. We began our investigation by identifying requirements. We used these requirements to drive a model-based development of the subsystem. We incorporated multiple fault management strategies and discussion of commonly occurring failures and design mistakes into our study. The scope of this study is an illustration of Rigorous Digital Engineering (RDE) and Model-Based Engineering (MBE) that spans from requirements down to verified code that can be deployed on a separation microkernel such as seL4.

Contents

1 Introduction

As the sun breaks over the hazy blue of the sphere, a ray of light reflects from the edge of the cube. The cube tumbles along its path and follows the draw of that sphere. For many turns of the sphere below, the cube watched carefully and performed its purpose well. On this fateful morning, the cube would die. No one would know the details of the final moments. Without warning its trajectory began to inexorably decay as the drag of the atmosphere slowed the cube's orbit. It looked out across the convex blue one last time as it fell towards Earth and burned its farewell.

This story depicts the final cry of many CubeSats despite our advancements in research and technology. In this technical report, we investigate a redundancy and fault management subsystem frequently used in mission-critical systems. We began our investigation by identifying requirements. We used these requirements to drive a model-based development of the subsystem. We incorporated multiple fault management strategies and discussion of commonly occurring failures and design mistakes into our study. The scope of this study is an illustration of Rigorous Digital Engineering (RDE) and Model-Based Engineering (MBE) that spans from requirements down to verified code that can be deployed on a separation microkernel such as seL4 [32].

RDE is a methodological approach used to design complex systems. It incorporates model-based engineering and mathematical analyses into the development process. RDE does not focus on a single aspect of system development, but rather the holistic view. RDE concerns itself with the mapping from requirements to verifiable specifications, a rigorous approach to software engineering, assurance cases that span the complexity of the system, and particular attention to safety and security goals in both hardware and software. One aspect of RDE is Model-Based Engineering (MBE) in which a model provides a conceptual abstraction of physical or digital phenomena in which multiple verification and assurance activities can be performed. In this document, we describe a model-based approach to develop a sensor system with redundancy management that can be used within a plethora of critical systems, including CubeSats.

1.1 The CubeSat and Redundancy

Let us return to the CubeSat to illustrate a RDE pathway that may result in a longer lifespan. An Earth observing CubeSat has complex interacting subsystems, with many potential failure modes due to the complexity of the interacting subsystems and the harsh environments in which it operates. High assurance is key to mission success.

CubeSats may operate in different modes within different mission phases. Various configurations may be statically analyzed to verify correctness and carefully uploaded. Due to size and weight constraints, the dynamic reconfiguration options are limited. Factors that impact resiliency and survivability of the overall mission include radiation induced upsets, timing interactions, and the capabilities of sensing and propulsion subsystems [1, 13, 24]. Any given mission step in an observation sequence might have different dependability requirements which may drive the assurance requirements on specific components.

Cyber Physical Systems (CPSs) in general often have complex interacting subsystems and demanding security, safety, and performance requirements. Correct and safe operation is required, even in the presence of faults. Multiple forms of fault detection and mitigation can be incorporated into safety critical system design. One commonly used engineering mechanism is redundancy. For example, if a hardware component has a failure rate higher than what is allowable given system safety requirements, redundancy may be a viable option to bring reliability into an acceptably safe range — although at some cost, such as increased size, power, latency, and perhaps even reduced overall mission lifespan.

When designing a mission-critical system (e.g., satellites, commercial and military avionics, or medical devices), system engineers and developers often assess whether redundancy is necessary, and if so, how much redundancy is required, such as *simplex*, *duplex*, *triplex*, or *quad* [16]. The selection depends on the expected failure modes and effects. Simplex may be used for non-critical, easy to restart functions. Duplex provides availability if one component fails silent, while triplex is useful when a component fails in an unknown way. Quad redundant systems tolerate some classes of Byzantine errors [14, 37].

We refer to an implementation without redundancy as a simplex implementation. As we expand the concept of redundancy into duplex, triplex, or quad, simply stating whether it has duplex or triplex redundancy is insufficient to convey the specific architecture that has been selected. The following duplex design choices for a Sense-Process-Actuate model illustrate the complexity of the design space:

- *Duplicate sensing:* One may choose to design a system with multiple sensing components. There will be a management system incorporated into the design that will choose the value to report and process.
- *Duplicate processing:* In some cases, a system engineer may implement redundancy within the processing components. This could be two computational processes, two threads within a single process, etc. Unlike redundancy of sensing, the redundancy could be potentially be implemented just within software, rather than hardware.
- *Duplicate actuation:* Similar to sensing, this is duplication of the actuating components.
- *Cross-connections:* One must also specify how the connections are made between components when redundancy is introduced. One may have two separate and distinct flows of Sense-Process-Actuate or there may be cross-connections between components at each stage.

To determine which form of redundancy is appropriate for the system under development, the specific mission requirements should be considered, as well as the expected failure modes. There will be a tradeoff between the design decisions and the resulting expected performance profile. For example, adding redundant components and connections to a system may increase mission availability for a portion of the mission lifecycle. Those additions may also increase complexity and introduce new failure modes that *reduce* mission availability at other times. Model-based analyses may aid making design decisions and trading off alternatives.

In any redundant system, eventually the redundant streams must come together. For example, consider the left side of Figure 1 with redundant sensors and actuators and a single process that decides how to drive the actuators. This might be a scenario where have high sensing error rates, but having multiple sensors is acceptable within the mission context. On the right side of the figure, the processing is redundant. This might be appropriate for situations when the processors or memories are subject to errors. In both cases, voters down-select from the available inputs. Ideally, these voters should be high assurance and not induce additional errors at unacceptable rates.

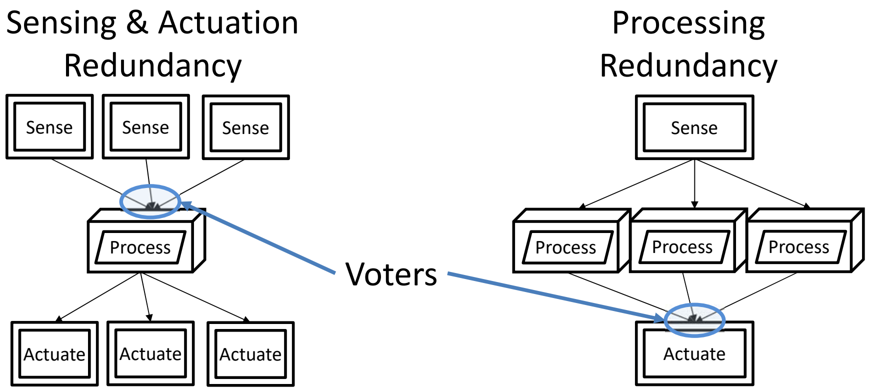


Figure 1: *Different redundancy schemes showing redundancy in sensors and actuators (left) vs redundant processing (right). In both cases, high assurance voters down-select from the available inputs.*

This report documents a fault management system involving three redundant sensors and a voter. It is a triplex redundant sensing system (gracefully degrading to duplex under certain failure conditions) with fault management between the sensing redundancy and the processing and actuation components of the system. We describe the development of the architectural and behavioral models based on the requirements document (Appendix A), and use these models to automatically derive code that can be deployed on a time and space separation microkernel. Developers specify behavioral contracts at the model-level. Next, the developers can invoke an automatic code generator that automatically weaves those contracts into the generated code. Developers can apply a program verifier to formally verify that the application code satisfies the specifications. We investigate various algorithms that can be implemented to perform the fault management and uncover real world complexities within what seems to be a straightforward system.

2 System Description

Redundancy architectures may rely on voting logic to manage the redundancy and provide the required fault tolerance. Assurance expectations of these voters can be driven by the criticality of their outputs. As such, these voters can be a useful representative example of high assurance software development techniques required for mission-critical CPSs. This report documents the design and implementation of a reconfigurable redundancy management system. The resulting voter model and software is design-time reconfigurable to accept varying numbers of inputs for a variety of input conditions. The voter configuration is potentially applicable for multiple domains, including avionics and medical devices, as well as the motivating CubeSat example.

The redundant sensor subsystem supplies a high integrity, high availability temperature reading of the environment. Each sensor senses the environmental temperature and sends the resulting temperature data to a redundancy management subsystem. This subsystem has two modes of operation: (1) a triplex mode in which when a single sensor disagrees with its counterparts and the temperature output of the system corresponds with majority agreement, and (2) a degraded duplex mode that it can use under specific failure scenarios involving multiple sensors.

The redundancy is provided by multiple sensing components. There are three sensors with three corresponding software sensing processes, each containing a single thread. The three sensing components are connected to a fault management system called the voter. This voter is a majority voter that provides a temperature to the rest of the system.

As simple as the description sounds, there are complications in the real world one must consider. For example, the relative location of the three sensors may result in slightly different sensed temperatures. A sudden rise or fall in environmental temperature may exceed the rate at which those changes can be read and reported by all three devices. In this study, we investigated multiple potential failure paths. We implemented a two-mode system to manage a large subset of the potential failures.

2.1 Requirements

As a system is being developed, design decisions should trace back to system- and component-level requirements. As code is being tested and formally verified, the requirements can provide clear behavioral insight and the ability for specifications to be derived. Of course, these statements apply in a perfect world with unambiguous and complete requirements, which is often not the case. Nonetheless, we created a requirements document for the redundant sensor system that captures environmental assumptions, use cases, and component and system safety and security requirements. As we developed the system, we discovered it became necessary to refactor certain requirements, refine the statements of their meaning and intent, and update the models and code appropriately. In the real-world, multiple teams likely have responsibility for these activities. This drives the need for clear and unambiguous communications, as well as the ability to track and manage versions of all these documents.

The operational context of the redundant sensor system is shown in Figure 2. The sensors interact with the ambient environment. The sensing threads query the sensors (e.g., read the sensor values that are available via General Purpose Input Output (GPIO)), and logically interact with the redundancy management system. The redundant sensor system is meant to be incorporated into an external system (such as a CubeSat) which we call the “user.” The full requirements document can be found in the Appendix A. It includes the high-level goals of the system, operational concepts with use cases, and the assumptions and requirements of each component. We refer to the system and component requirements throughout the remaining technical report.

2.2 Triplex Mode

Within the triplex mode of the redundancy management system, three sensors are operational and supply temperature readings to the voting subsystem. This mode of operation can handle a certain subset of sensor failures, including out of range and out of bounds errors. If a sensor provides a value that disagrees with the other two sensor values, the voter can perform a majority ruling to ascertain the current environmental temperature. The pseudocode describing the behavior of the triplex voter is shown in Algorithm 1.

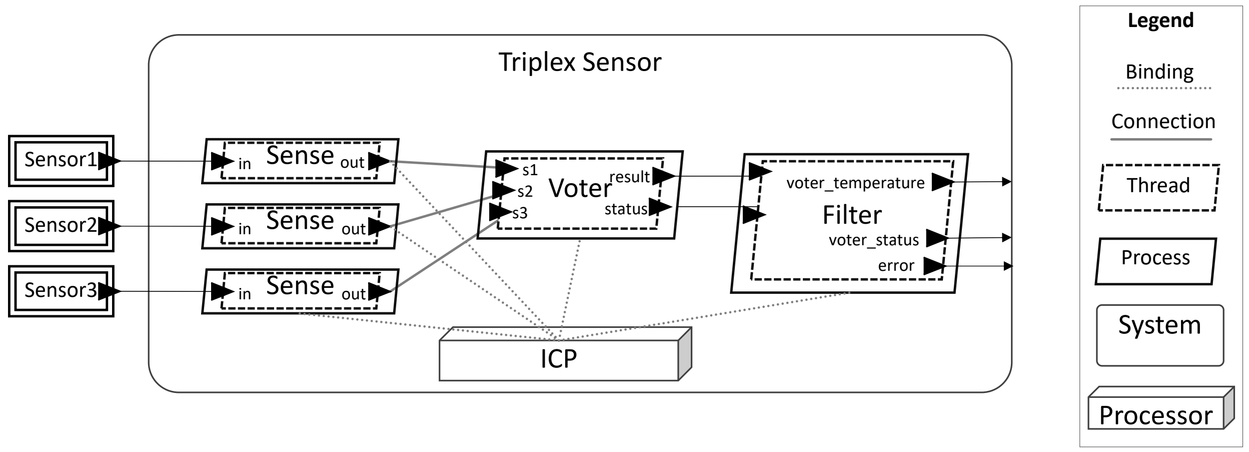


Figure 2: *Context diagram for the redundant sensor system.*

Algorithm 1 The behavior of the triplex majority voter.

```
if any pair of sensors agree then
    temperature ← agreed upon value
end if
if all sensors agree then
    status ← good
else if only two agree then
    status ← single fault
else
    status ← faulty
end if
```

2.3 Duplex Mode

There are cases in which a single sensor has failed and the system should still operate, albeit in a duplex mode. The following requirements specify the conditions that must be met to switch into duplex mode of operation:

- Req-RM-8: Redundancy management will switch to duplex mode only when all of the following conditions are met:
  - A single sensor perceived temperature has disagreed with the majority for 3 temperature readings (this reading inclusive).
  - The erroneous value on this sensor has been equivalent for all 3 temperature readings (this reading inclusive).
  - The remaining two sensors agree on a perceived temperature value for all 3 temperature readings (this reading inclusive).
- Req-RM-9: If redundancy management switches to duplex mode, then the redundancy management status will indicate which sensor displays a stuck value.
- Req-RM-10: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the temperature reported by the voter will be the midpoint value of the two servicing sensor input values.
- Req-RM-11: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the status of the voter will

indicate that it is in a degraded mode of operation.

The three conditions outlined in Req-RM-8 must be met for the system to initially enter duplex mode: a single sensor reading has been in disagreement for three readings, this value has been equivalent all three readings, and the other servicing sensors are in agreement, albeit on a different value. Once the system is in duplex mode, Req-RM-10 states that the system can enter a degraded mode of operation if the two servicing sensors report values that disagree. The pseudocode implementation of the duplex mode of operation is shown in Algorithm 2.

Algorithm 2 The behavior of the duplex mode of the voter.

```
if duplex conditions apply then
    status ← stuck
    temperature ← agreed upon temperature from servicing sensors
end if
if degraded duplex conditions apply then
    status ← degraded
    temperature ← midpoint of temperatures from servicing sensors
end if
if triplex conditions apply then
    status ← triplex status
    temperature ← majority
end if
```

2.4 Background

The following background information includes a short description of the modeling language and toolset used throughout this investigation, as well as commonly used terminology for clarity.

Architecture Analysis and Design Language (AADL): SAE International standard AS5506C [33] defines the AADL core language for expressing the structure of embedded real-time systems via definitions of software and hardware components, their interfaces, and their communication. The AADL provides a precise, tool-independent, and standardized modeling vocabulary of common embedded software and hardware elements using a component-based approach [15]. The AADL standard includes a Run Time Specification (RTS) that is a collection of run-time libraries that provide key aspects of threading and communication behavior. A major subset of the RTS has been formalized and a reference implementation has been developed [21]. We designed our contract language and associated translation to code-level contracts with these definitions in mind.

High-Assurance Model-based Rapid engineering for embedded systems (HAMR): The HAMR framework generates high assurance software from AADL architectural models for multiple execution platforms [19, 5]. HAMR encodes the system-level execution semantics as specified in standardized models with clear and unambiguous specifications, into infrastructure code suitable for the given target platform. This includes generating threading, port communication, and scheduling infrastructure code that conforms to the AADL RTS as well as application code skeletons that developers fill in to complete the behavior of the system. For the Java Virtual Machine (JVM) platform, HAMR generates code in Slang [20], a high-integrity subset of Scala, which can be integrated with support code written in Scala and Java. Mixed Slang/Scala-based HAMR systems can also be translated to JavaScript (e.g., for simulation and prototyping) and run in a web browser or on the NodeJS platform. HAMR generates C infrastructure and application skeletons when targeting Linux and the seL4 micro-kernel [32]. HAMR factors its C code through a Slang-based reference implementation of the AADL run-time and application code skeletons. Using the Logika verification framework for Slang (described below), Slang code can be verified with a high-degree of automation. This provides a basis for starting with MBE to develop high-assurance systems using Slang directly or via translation of Slang to C. C code transpiled from Slang can be compiled using standard C compilers, as well as the CompCert Verified C compiler [28]. See Section 5 for more on HAMR.

Logika: Logika is a highly automated program verifier for Slang [27]. Slang's integrated contract language enables developers to formally specify method pre- and post-conditions, data type invariants, and global invariants for global states. Logika performs compositional verification of code conformance to contracts. It employs multiple back-end solvers in parallel, including Alt-Ergo [11], CVC4 [4], CVC5 [3], and Z3 [31]. Incremental and parallel (distributable) verification algorithms supports Logika's scalability. For situations where automated solvers cannot provide full verification, Slang includes an extensible proof language that supports manual deduction techniques. Verification results, developer feedback on verification status, and contract or proof editing are supported in the Sireum Integrated Development and Verification Environment (IDVE), a customization of the popular IntelliJ Integrated Development Environment (IDE).

The code-level contracts for methods include a *Requires* clause (pre-conditions), an *Ensures* clause (post-conditions), and a *Modifies* clause (frame conditions). Within the Integrated Verification Environment (IVE), the developer can access program state, verification conditions, and solver interactions.

The Logika verification engine uses asynchronous communications between the plugin client and tool server. This enables a seamless, on-the-fly integration similar to static type checking analysis usually offered by IDEs. Logika's usability features include an as-you-type well-formedness analysis and verification of Slang programs by sending the checking tasks to a background server process. Its usability is increased via visualizations of feedback propagated from the server as responses of the verification requests.

Faults, Errors, and Failures: The terminology in this report follows that described in Laprie, et al. [2]. A *failure* is an event that occurs when the delivered service deviates from correct service. Since a service is a sequence of the system's external states, a service failure means that at least one or more external states of the system deviates from the correct service state. The deviation is called an *error.* The hypothesized cause of an error is called a *fault.* A fault is *active* when it causes an error, otherwise it is *dormant.*

3 Architectural Model

Based on the system requirements, we developed the architectural representation of the triplex redundant sensor system in AADL. Environmental inputs are modeled as an abstraction of the sensor devices. The environment also includes the consumer which receives input from the triplex sensor system. This abstraction also serves as a test harness for the system model. The triplex voter system contains the sensor processes and threads, the voter, and a filter on the output of the voter. All of these processes (with containing threads) are bound to an Integrated Core Processor (ICP).

The environment sensor outputs are sampled by sensing threads and then sent to a voter process. These are modeled as 32-bit floating points and are of type Kelvin. The voter thread, a subcomponent of the voter process, implements majority voting. A structural diagram of the system model is shown in Figure 3.

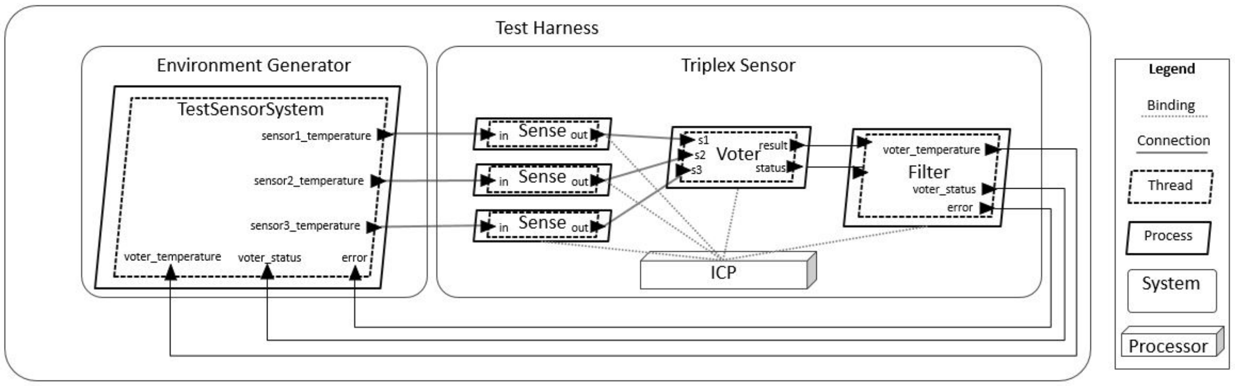


Figure 3: *Redundant sensors architecture*

The triplex majority voter output includes temperature. The implementation of the voter is that of majority rule within allowed bounds. The voter output also includes a status that specifies whether the voter is uninitialized, all sensors agree, a single fault is present, or the output is faulty (i.e., multiple faults were detected and unhandled). The filter process receives the voter outputs and restricts the accepted values to be within the appropriate temperature range. If an error is detected, it will report this error to the consumer along with the voter status output and a default error temperature reading.

Model Assumptions

We made some simplifying design decisions for the redundant sensor system, motivated by approaches often used for embedded systems.

- All processes are run on a single core processor.
- There is a single writer to all data ports.
- Threads are periodic.
- The voter is designed to operate on a static cyclic schedule.
- The rate of sampling of the sensors is greater than the rate of expected change of environmental temperature.

3.1 Model-level Analyses

Model-based development of the architecture provides a way for stakeholders, such as Program Offices and product vendors to specify requirements, as well as a framework against which refinements of the architecture can be checked against the requirements. While this study did not explore the details and nuances of each possible level of architecture, we illustrated feature analysis that may be applied at multiple levels. Early analysis of system components and their relationships can uncover potential errors and integration issues. Not identifying those issues early, and therefore needing to address them late in the process can costs orders of magnitude more than if they had been fixed earlier [38, 39]. Given the distributed nature of major acquisitions, early identification of integration issues can reduce the risk of unplanned schedule and cost impacts.

During the early stages of system development, model-level analysis can help developers uncover requirement, architectural, and behavioral issues before it becomes very costly to refactor the implementation. Model-based analyses address interactions that can – among other things – impact the non-functional properties of a system, such as timing, safety, security, and reliability. Issues involving non-functional properties are often not uncovered until integration. This makes them a key contributor of those early to be introduced, late to be fixed errors. The errors are often due to the interaction complexity between the software components and their deployment on the hardware platform [8]. Model-level analysis can provide assurance on the compatibility and interoperability of components, provide insight into the system under development, and allow for ease of restructuring, respecifying, and reanalyzing system components and features.

On this study, we performed model-based analysis on the compatibility and interoperability of components using Assume Guarantee REasoning Environment (AGREE) [10]. We then extended the nominal (fault-free) analysis with the Safety Annex [36] and Error Modeling Annex Version 2 (EMv2) [12] to gain insights into the system in the presence of faults.

3.1.1 Component Interface Analysis

AGREE is an AADL component annotation for formal behavioral contracts. Each component's contracts can include assumptions and guarantees about the component's inputs and outputs respectively. The interface specification describes the externally visible required behavior of the component without divulging how this will be achieved.

The AGREE analysis tool transforms an AADL model and the behavioral contracts into Lustre [18] and then queries a user-selected model checker to conduct the back-end analysis. The analysis can be performed compositionally following the architecture hierarchy. When compared to monolithic analysis (i.e., analysis of the flattened model), the compositional approach allows the analysis to scale to much larger systems.

We defined the nominal behavioral model using AGREE interface specifications. AGREE allows the user to define assumptions on component inputs and guarantees on the outputs. Such specifications mean that if the assumptions hold now and historically, then the output can be guaranteed. The AGREE contracts do not define the implementation, but rather guarantee component interoperability with respect to component interface specifications at the model level. This kind of analysis may be performed very early in the design phase, as well as throughout the lifecycle, such as during updates and maintenance actions. The analysis can provide insight into the compatibility of components and their inter-connections. We narrowed our AGREE analysis to the triplex sensor system. The sensor process and its associated AGREE contracts are shown in Listing 1.

Listing 1: *The AGREE contract for the sensor process in AADL.*

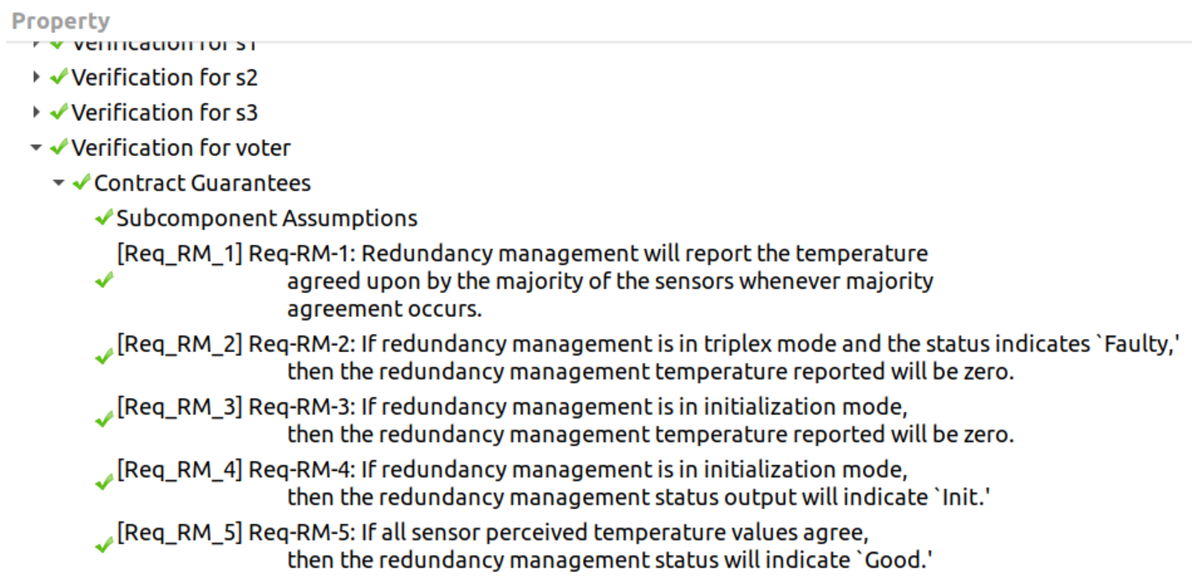
```
 1  process sensorProcess
 2    features
 3      temperature_input : in data port Datatypes::Kelvin.impl;
 4      reported_temperature : out data port Datatypes::Kelvin.impl;
 5
 6    annex agree{**
 7      -- Full Military temperature grade: -55 degrees C to 125 degrees C
 8      -- In kelvin (floor/ceiling): 218K to 399K
 9      assume EA_TS_1 "The environmental temperature will always
10        be greater than 218 Kelvin." :
11        (Constants::MIN_SENSOR_RANGE < temperature_input.temperature);
12
13      assume EA_TS_2 "The environmental temperature will always
14        be less than 399 Kelvin." :
15        (temperature_input.temperature < Constants::MAX_SENSOR_RANGE);
16
17      -- Req-TS-1: The perceived temperature will be provided to the
18      -- redundancy management system in degrees Kelvin.
19      -- This requirement is handled through type checking.
20
21      guarantee Req_TS_2 "The perceived temperature provided to the voter
22        will be strictly within the range of 218 K to 399 K. " :
23        (Constants::MIN_SENSOR_RANGE < reported_temperature.temperature)
24          and (reported_temperature.temperature < Constants::MAX_SENSOR_RANGE);
25
26      guarantee Req_TS_4 "The perceived temperature provided to the voter will be the
          same
27        as the environmental temperature. " :
28        true -> (reported_temperature.temperature = temperature_input.temperature);
29
30    **};
31  end sensorProcess;
```

We assume that the environmental temperature is within the full military temperature grade expressed in Kelvin (between 218K and 399K, lines 9 and 13). We defined guarantees over the outputs of each component (lines 21 and 26). The guarantees for the sensor thread specify an initial temperature output and that after the initial state, the temperature output reflects that of the input.

Each identifier and description string provides traceability to the requirements document. The assumptions on the sensor process component provide expected temperature ranges, and the guarantee reflects the same operational temperature for the component output. The nominal AGREE model can be verified using either monolithic or compositional analysis. This verification shows that the components within the architectural model are mutually compatible as long as assumptions are met and guarantees are enforced at the boundaries of each component. Figure 4 shows part of the results of AGREE verification for the specifications of the triplex mode of operation.

To perform similar analysis, the developer will run *Verify all layers* AGREE analysis on the *filter* implementation of the triplex sensor system[1].

[1] AGREE installation instructions and releases may be found at `https://github.com/loonwerks/AGREE/`


Property

✓ Verification for s1

✓ Verification for s2

✓ Verification for s3

✓ Verification for voter

✓ Contract Guarantees

✓ Subcomponent Assumptions

✓ [Req_RM_1] Req-RM-1: Redundancy management will report the temperature agreed upon by the majority of the sensors whenever majority agreement occurs.

✓ [Req_RM_2] Req-RM-2: If redundancy management is in triplex mode and the status indicates `Faulty,' then the redundancy management temperature reported will be zero.

✓ [Req_RM_3] Req-RM-3: If redundancy management is in initialization mode, then the redundancy management temperature reported will be zero.

✓ [Req_RM_4] Req-RM-4: If redundancy management is in initialization mode, then the redundancy management status output will indicate `Init.'

✓ [Req_RM_5] Req-RM-5: If all sensor perceived temperature values agree, then the redundancy management status will indicate `Good.'

Figure 4: *A subset of AGREE verification results for the sensor system.*

3.2 Lessons Learned

In the process of specifying AGREE contracts and performing both nominal and fault analysis, we uncovered multiple issues with the requirements and model. This exercise forced us to consider a number of common issues that can be easily overlooked.

- *Unclear requirements:* As we converted the requirements into interface specifications, some of the natural language statements in the requirements document were vague and difficult to understand. Despite our greatest efforts to write concise and complete requirements, the manual conversion into specifications uncovered issues. We rewrote those requirements to be more specific, as well as include certain initial states.
- *Necessary textual redundancy required to use compositional verification:* AGREE analysis makes use of the hierarchical nature of AADL models. The analysis starts at the leaf level and uses the results to prove the contracts at the next highest level. Many of the AGREE annex elements were repeated almost exactly from one layer to the next. Part of this is a refinement and strengthening of the specifications, but in some cases it is repeating equivalent statements. From a user's perspective, this may seem like extra redundant effort, but it is necessary for AGREE's compositional analysis.
- *Ensuring that top-level specifications adequately capture what you want to state about the system:* One must specify what is being verified. In our case, we wanted to show that the voter and filter performed fault management correctly. If the system requirements are not adequately expressive, it can be challenging to state exactly what system level property you wish to prove. Using AGREE forced us to clearly, correctly, and unambiguously state these properties and requirements. This took multiple iterations in some cases.

releases

4 Model-based Safety Assessment

System safety assessment and analysis applied during the development life cycle of critical systems can provide evidence that the design satisfies the stated safety requirements, and that the design complies with applicable standards. A prerequisite for successful safety analysis is a thorough understanding of the system architecture and the behavior of its components. Safety engineers use such understanding to explore the system behavior and evaluate its ability to safely operate, assess the effect of failures on the overall safety objectives, and construct the accompanying safety analysis artifacts. Developing adequate understanding, especially for software components, can be a difficult and time consuming endeavor. The lack of precise models of the system architecture and its failure modes often forces safety analysts to devote significant effort to gathering architectural details about the system behavior from multiple sources. Comprehensive identification of all unsafe interactions in increasingly complex software-intensive systems is also a challenge. Leveraging model-based system development in critical systems and use of a common formal model shared between system development and safety analysis may ameliorate many of these challenges. Model-based safety assessment can help eliminate ambiguity, promote artifact consistency, analysis accuracy, and minimize design iterations [35, 26, 9, 29].

The author has previously recommended a model-based safety assessment process backed by formal methods to help safety engineers detect design issues early in the design lifecycle [35]. This methodology is summarized in Figure 5.

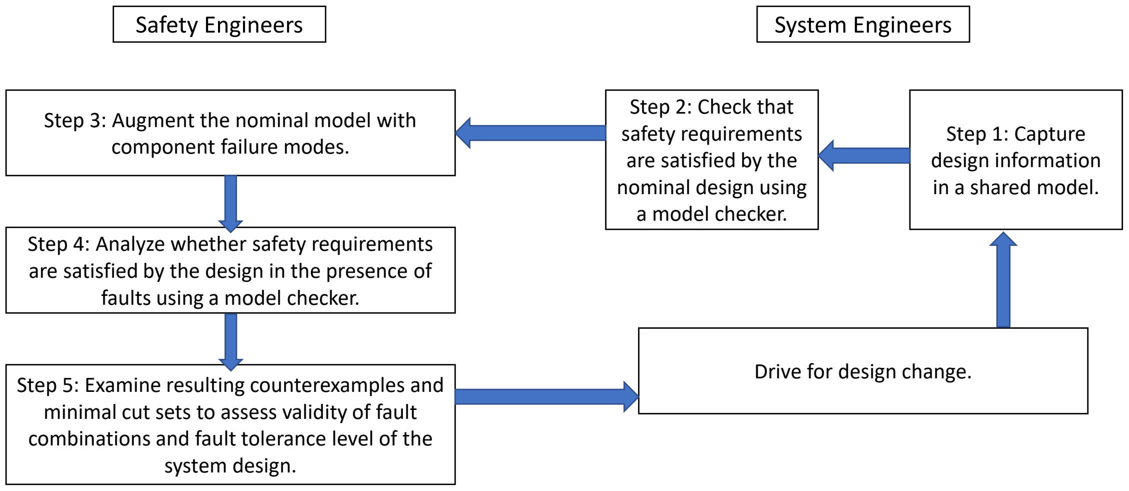

Safety Engineers
System Engineers
Step 3: Augment the nominal model with component failure modes.
Step 2: Check that safety requirements are satisfied by the nominal design using a model checker.
Step 1: Capture design information in a shared model.
Step 4: Analyze whether safety requirements are satisfied by the design in the presence of faults using a model checker.
Step 5: Examine resulting counterexamples and minimal cut sets to assess validity of fault combinations and fault tolerance level of the system design.
Drive for design change.

Figure 5: *Safety assessment process backed by formal methods.*

1. System engineers capture the critical information in a shared model that includes high-level hardware and software architecture, nominal behavior at the component level, and safety requirements at the system level.
2. System engineers use an automated model checker to verify that the nominal model supports the requirements.
3. Safety engineers augment the nominal model with the component failure modes.
4. Safety engineers use the model checker to analyze if the safety requirements and fault tolerance objectives are satisfied by the model in the presence of faults. If the model design does not tolerate the specified number of faults (or probability

threshold of fault occurrence), then the model checker produces artifacts specific to safety requirement violations.

5. The safety engineers examine the results to assess the validity of the fault combinations and the fault tolerance level of the system design. If a design change is warranted, system engineers will update the model with the necessary design change and the above process will be repeated.

Model-based safety assessment approaches have been developed for a variety of modeling languages, including SysML [23, 30], AADL [36, 34], NuSMV [6], and Simulink [25]. Each language has a targeted domain of application and contains different levels of formalism. For our approach, we determined that AADL was suitable. However, the proposed safety assessment approach can be deployed using a variety of models, languages, and formal method tools. We describe one such interpretation of this process. We also extended the safety assessment process into our software verification activities as described in Section 6.

4.1 Fault Modeling with the Safety Annex

The *safety annex* is an extension to AADL that supports modeling of system behavior under failure conditions [36, 35]. The annex enables the modeling of component failures and allows safety engineers to weave various types of fault behavior into the nominal system model. The accompanying tool support uses model checking to automatically propagate errors from their source to their effect on top-level safety properties (written in AGREE) without the need to add separate propagation specifications.

Analysis options include the compositional computation of minimal cut sets [37] and either monolithic or compositional verification in the presence of faults [36]. In all cases, the analysis requires AGREE contracts to be present and verify completely in the absence of faults. The safety annex tool incorporates fault definitions into the backend Lustre model [18], triggers the faults based on thresholds specified in the model, and the JKind model checker [17] performs analysis using the Lustre code as input.

Fault specifications are defined for component outputs as shown in Listing 2.

Listing 2: *A fault definition in the safety annex for the sensing thread.*

```
fault temp_sensor_above_range "Temp sensor above range.": Common_Faults::fail_to_int {
  inputs: val_in <- reported_temperature, alt_val <- Constants::ABOVE_RANGE;
  outputs: reported_temperature <- val_out;
  disable : false;
  probability: 1.0E-5 ;
  duration: permanent;
}
```

The parameters within the fault definition specify the inputs to the fault definition: `val_in` and `alt_val`. The parameter `val_in` corresponds to the nominal component output. The parameter `alt_val` corresponds to the faulty value or range of values that could be found on that output port. The output parameter allows one to specify which output port will be overwritten with a faulty value if this fault is active. The fault may be disabled, which allows for adjustable analysis if multiple faults are potentially active. Safety engineers may also specify the probability of the fault activating and the duration (transient or permanent) of the fault.

The fault definition must also refer to an AGREE node (this is specified on line 1 as `Common_Faults::fail_to_int`). A subset of the parameters of the fault definition are used as node parameters. The `fail_to_int` node is shown in Listing 3.

Listing 3: *A fault node in the safety annex that determines fault activation behavior.*

```
node fail_to_int(val_in: int, alt_val: int, trigger: bool) returns (val_out: int);
let
  val_out = if trigger then alt_val else val_in;
tel;
```

The parameter `trigger` is not defined by the user, but are unconstrained variables within the SMT encoding. The thresholds defined in the model are called *fault hypotheses*. An example of a fault hypothesis is shown in Listing 4 and constrains the number of active faults to a maximum of three. In line 2, one can see a commented out probabilistic hypothesis: the combined probabilities of active faults must not exceed 10^{-8}.

Listing 4: *A fault hypothesis in the safety annex.*

```
1 annex safety {**
2   --analyze : probability 1.0E-8
3   analyze : max 3 fault
4 **};
```

The safety annex analysis tools automatically encode these hypotheses as constraints in the backend SMT model. The SMT analysis uses these constraints to limit the number of faults active (or combined probabilities) and iterates through the possible fault combinations to determine if the model is satisfied or not. If an active set of faults fall within the threshold provided and AGREE behavior specifications are violated due to these faults, then the analysis results show violated guarantees and provide counterexamples, or will compositionally derive minimal cut sets.

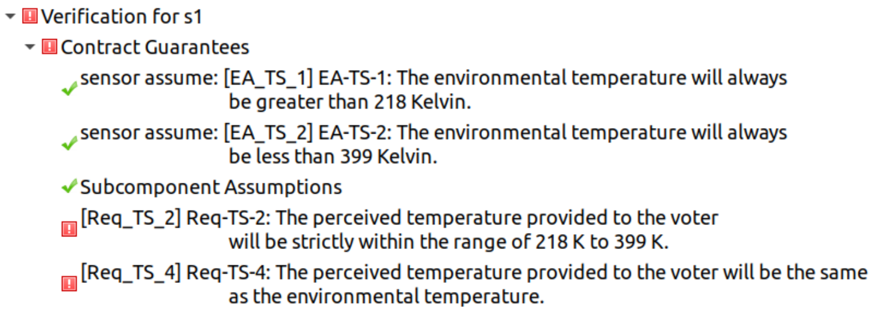
For the redundant sensors example, we chose a subset of value related faults and defined them within the safety annex library. These included *above range*, *below range*, and *stuck at*. We set a threshold at the top system level of the model (the `TriplexSensorSystem` in this case) that specifies either a maximum number of active faults or a probabilistic threshold. We chose to perform analysis using only a maximum number of faults and left probabilistic safety analysis for future explorations. We defined faults for each of the sensing threads in the model and allowed those faults to range between all fault types.

As with AGREE analysis, the verification can be performed compositionally. We defined a safety "hypothesis" for each layer of the system hierarchy. This constrains the number of faults active at each layer (or the probabilistic threshold). The analysis proceeds from the bottom up and enforces each hypothesis at each layer, rolling up the analysis results as it proceeds. With the constraint that one fault is active within the system, the analysis of our model showed that several sensor guarantees are violated, as expected. These included:

- Req-TS-2: The perceived temperature provided to the voter will be strictly within the range of 218 K to 399 K.

- Req-TS-4: The perceived temperature provided to the voter will be the same as the environmental temperature.

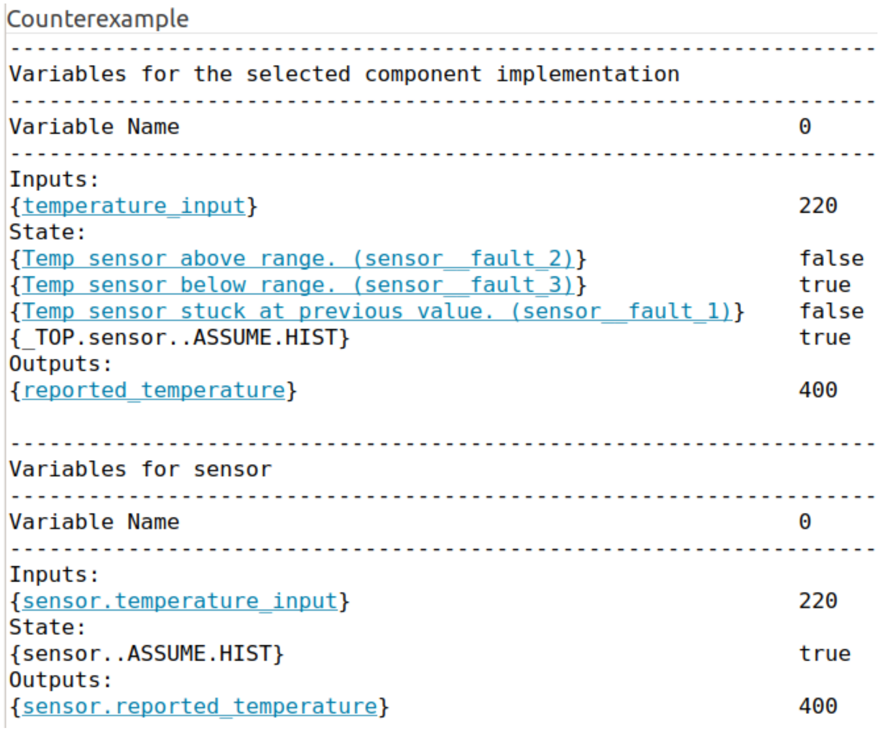
The analysis results are shown in Figure 6. These results make sense for scenarios such as when the physical sensors fail, or bits flip in the communications path; this is the precisely why the system has redundant sensors, as well as the voter to manage the faults.



```
▾ 🟥 Verification for s1
  ▾ 🟥 Contract Guarantees
      ✓ sensor assume: [EA_TS_1] EA-TS-1: The environmental temperature will always
                 be greater than 218 Kelvin.
      ✓ sensor assume: [EA_TS_2] EA-TS-2: The environmental temperature will always
                 be less than 399 Kelvin.
      ✓ Subcomponent Assumptions
      🟥 [Req_TS_2] Req-TS-2: The perceived temperature provided to the voter
                 will be strictly within the range of 218 K to 399 K.
      🟥 [Req_TS_4] Req-TS-4: The perceived temperature provided to the voter will be the same
                 as the environmental temperature.
```

Figure 6: *Violated guarantees within the sensor component upon activation of a single fault.*

The analysis tool allows the analysis to select the guarantee that specifies Req-TS-2 and query for a counterexample. This counterexample displays the active fault, the faulty value, and the state of the system when that fault is present. This view is shown in Figure 7.



```
Counterexample
------------------------------------------------------------------
Variables for the selected component implementation
------------------------------------------------------------------
Variable Name                                                    0
------------------------------------------------------------------
Inputs:
{temperature_input}                                              220
State:
{Temp sensor above range. (sensor__fault_2)}                     false
{Temp sensor below range. (sensor__fault_3)}                     true
{Temp sensor stuck at previous value. (sensor__fault_1)}         false
{_TOP.sensor..ASSUME.HIST}                                       true
Outputs:
{reported_temperature}                                           400

------------------------------------------------------------------
Variables for sensor
------------------------------------------------------------------
Variable Name                                                    0
------------------------------------------------------------------
Inputs:
{sensor.temperature_input}                                       220
State:
{sensor..ASSUME.HIST}                                            true
Outputs:
{sensor.reported_temperature}                                    400
```

Figure 7: *A counterexample for Req-TS-2 guarantee.*

As the verification rolls up to higher levels, we can see which guarantees are violated when a single fault is active among the sensors. The filter component guarantees that the reported temperature is either zero or in operational range before passing the status and resulting temperature to the top level system. Given the supporting guarantees of all subcomponents, the only violated guarantees are at the sensors as shown in Figure 7. This provides additional assurance that the fault management incorporated into the voter and filter are sufficient to manage the system in the presence of faults.

In order to test the fault activations at the top level, we added a lemma stating that each sensor output always agreed with each other. This lemma is verified when no faults are active and is unable to be verified otherwise[2].

5 Uniting the Model with an Implementation

While model-based approaches have many benefits when used correctly, the relationship between the model and the actual implementation must also be considered. If the model does not adequately represent the actual system that will be deployed, the model-level analyses may no longer apply to that system. For example, if the software developers who write the software do not implement the model-level specifications, including but not limited to the component interfaces, architectural relationships, communications patterns, and runtime execution models, the model-level analyses may not apply to the software as deployed.

To ameliorate this problem, the HAMR toolset (see Section 2.4) allows us to automatically generate verifiable code from the system-level architecture model that we analyzed. We can specify behavioral and non-functional requirements within the model, and HAMR will weave these specifications into the code-base. As the software developer writes implementation code within this automatically generated infrastructure code, the developer can formally verify these specifications using Logika (see Section 2.4), ensuring that the behavior of the implemented code satisfies the specified requirements. This provides a link between the model and early stage analyses and the code that will be deployed on the system.

5.1 HAMR Generated Code

HAMR is a code generation and system build framework for embedded systems whose architecture is specified using AADL [5]. HAMR encodes the system-level execution semantics – as specified in standardized models – into infrastructure code suitable for a given target platform [19]. These execution semantics include component interfaces, threading semantics, inter-component communication semantics, and scheduling, as shown in Figure 8.

Working off of an AADL instance model as generated by Open Source AADL Tool Environment (OSATE), HAMR generates specification files pertinent to the deployment topology as required by backend target and other kernel configuration information (e.g., CAmkES specifications in the case of seL4 backend target). For each thread defined in the AADL model, HAMR generates infrastructure code that implements the AADL thread dispatch semantics. The infrastructure code includes elements required for linking entry point application code to the kernel's underlying scheduling

[2]Safety annex installation instructions and releases may be found here: https://github.com/loonwerks/AMASE

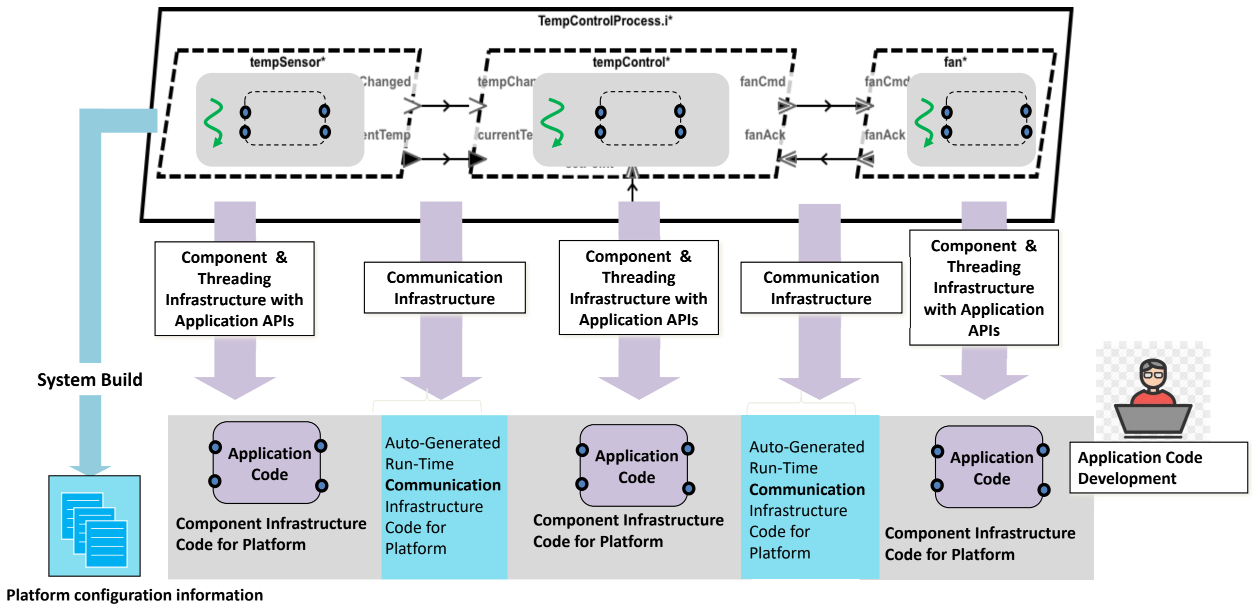

TempControlProcess.i*
tempSensor*
tempControl*
fan*
Changed
entTemp
tempChan
currentTe
fanCmd
fanAck
Component & Threading Infrastructure with Application APIs
Communication Infrastructure
System Build
Application Code
Component Infrastructure Code for Platform
Auto-Generated Run-Time Communication Infrastructure Code for Platform
Application Code Development
Platform configuration information

Figure 8: *HAMR architecture*

framework. It also includes code that implements the storage associated with ports, as well as the buffering and notification semantics associated with event and event data ports. The infrastructure code also includes developer facing code, including thread code skeletons in which the developer will write application code, and port Application Programming Interfaces (APIs) that the application code uses to send and receive messages over ports.

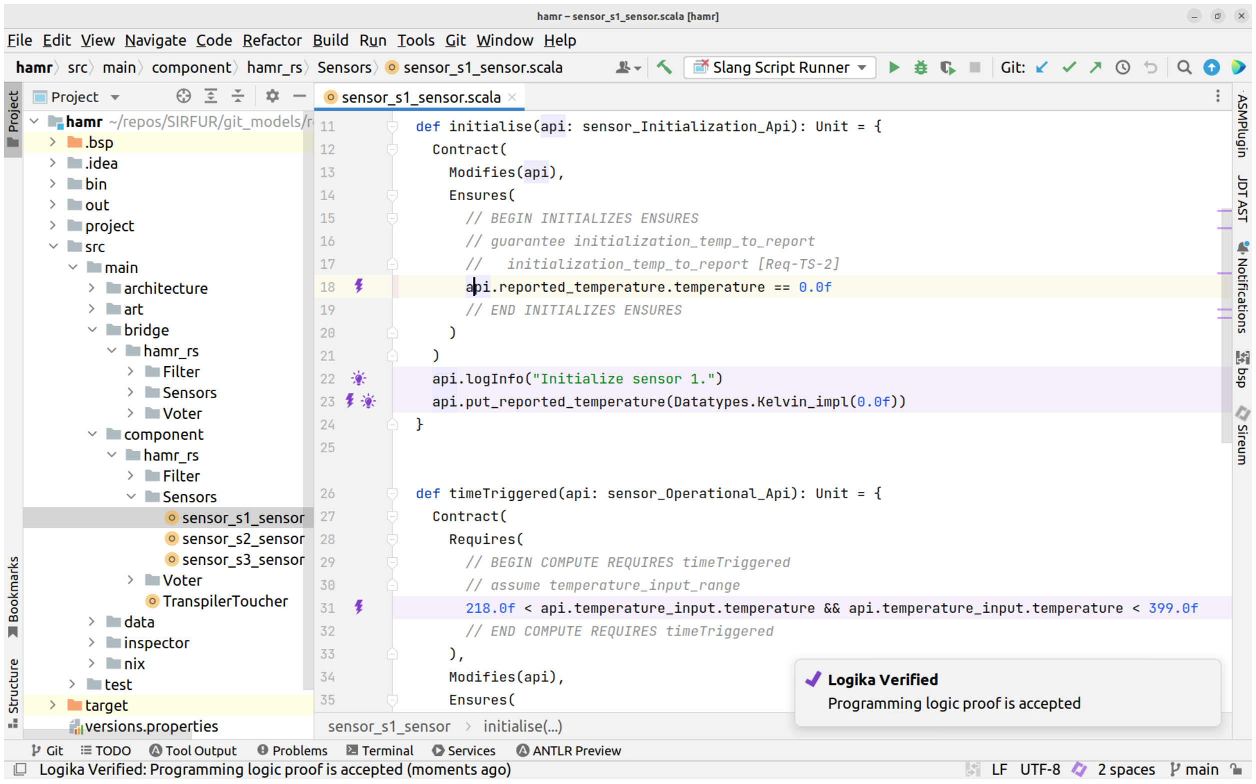
In a simplified view of an AADL model, `features` describe the model elements that are found on `ports` and reach destination components through `connections`. The HAMR-generated API files encode intermediary methods between the developer's application code and the AADL run-time (`Art`) methods that dictate scheduling, connections, ports, and other run-time elements. The API provides the developer access to run-time services and infrastructure, for instance, access to data on incoming ports and setters for data on outgoing ports. Listing 5 shows the auto-generated section of the API code that a thread would use to put a value on an outgoing port, in this case the use of the method `put_reported_temperature`.

Listing 5: *The PutValue API method auto-generated by HAMR.*

```
@sig trait sensor_Api {
  def id: Art.BridgeId
  def temperature_input_Id : Art.PortId
  def reported_temperature_Id : Art.PortId

  def put_reported_temperature(value : Datatypes.Kelvin_impl) : Unit = {
    Art.putValue(reported_temperature_Id, Datatypes.Kelvin_impl_Payload(value))
  }
```

HAMR generates multiple files to represent the interface and implementation of an AADL project, including all threads defined in the system model. The HAMR generated project imported into the Sireum IVE is shown in Figure 9.



```
def initialise(api: sensor_Initialization_Api): Unit = {
  Contract(
    Modifies(api),
    Ensures(
      // BEGIN INITIALIZES ENSURES
      // guarantee initialization_temp_to_report
      //   initialization_temp_to_report [Req-TS-2]
      api.reported_temperature.temperature == 0.0f
      // END INITIALIZES ENSURES
    )
  )
  api.logInfo("Initialize sensor 1.")
  api.put_reported_temperature(Datatypes.Kelvin_impl(0.0f))
}


def timeTriggered(api: sensor_Operational_Api): Unit = {
  Contract(
    Requires(
      // BEGIN COMPUTE REQUIRES timeTriggered
      // assume temperature_input_range
      218.0f < api.temperature_input.temperature && api.temperature_input.temperature < 399.0f
      // END COMPUTE REQUIRES timeTriggered
    ),
    Modifies(api),
    Ensures(
```

Logika Verified
Programming logic proof is accepted

Logika Verified: Programming logic proof is accepted (moments ago)

Figure 9: *HAMR generated redundant sensors project.*

The left side of the figure shows the project organization and directories. This includes thread implementation files (in *component* directory), an API from component implementation to run-time services infrastructure is in *bridge* directory, data types (in *data* directory), and the AADL run-time (Art) services implementation (in *art* directory). The component application code is generated as method stubs, initially lacking an implementation. Note that Figure 9 displays sensor initialize and compute methods with contracts and implementation. The contracts are automatically generated from model-level specifications (see Section 5.2) and the application code was inserted by a developer.

5.2 The GUMBO Contract Language and HAMR

After the HAMR code is generated, a developer writes the application code for each thread in the system. One can manually specify requirements by adding Logika contracts to the application code and API methods, but we chose to further connect the model to the implementation through the GUMBO Contract Language (GCL).

The GCL is an AADL annex that supports model-level behavioral specification [22]. HAMR automatically extracts GCL contracts from AADL models and weaves them into the HAMR-generated code skeletons in Slang. Logika supports contract reasoning at the code level and is used to verify that the thread implementations conform to contracts at both the code and model levels. The GCL was designed in tandem with a formalization of the AADL run-time services and is closely aligned with the semantics of HAMR generated infrastructure code [21].

The following is a high-level overview of the GCL to help understand the language in context. Figure 10 illustrates how these contracts fit into a model view of a component.

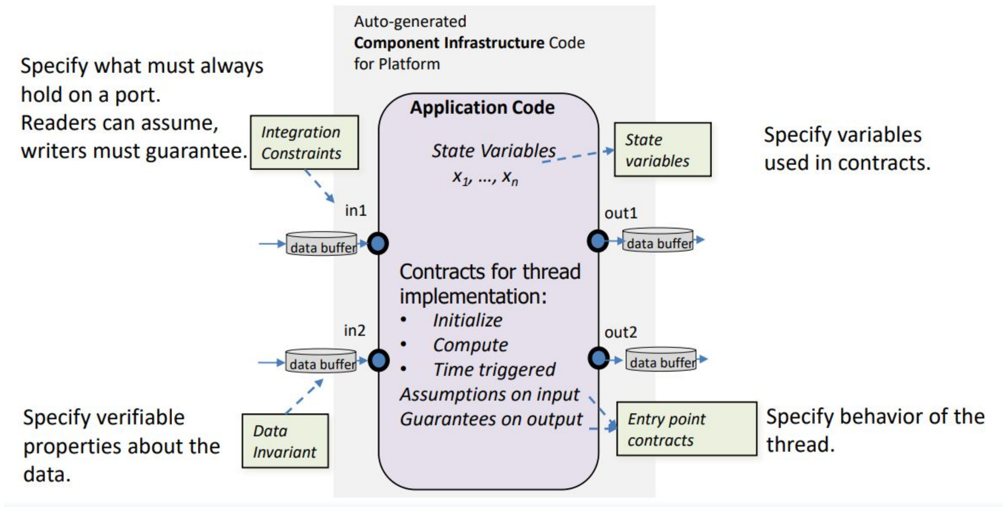

Auto-generated
Component Infrastructure Code
for Platform
Specify what must always hold on a port. Readers can assume, writers must guarantee.
Integration Constraints
Application Code
State Variables
$x_1, \dots, x_n$
State variables
Specify variables used in contracts.
in1
out1
data buffer
Contracts for thread implementation:
Initialize
Compute
Time triggered
in2
out2
Assumptions on input
Guarantees on output
Specify verifiable properties about the data.
Data Invariant
Entry point contracts
Specify behavior of the thread.

Figure 10: *GCL contracts in the context of HAMR infrastructure code elements.*

Contract categories are specified based on the component, data, or port under analysis.

State Variables: A state variable is declared and used within a GCL annex and can be referenced within any contract. It is local to the thread in which it is declared and is translated into a *spec variable* in the code base. Spec variables are specific to the verification and are not compiled with the rest of the source code. This is shown in the application code box in the center of Figure 10.

Data Invariants: specify verifiable and unchanging (hence invariant) properties about data. This is illustrated in the bottom left of Figure 10. One can also specify constraints over more than one field of a record or struct type. These contracts are defined on a data subcomponent within the model and are verified each time a variable with this particular data type is assigned a value.

Entry Point Contracts: Each thread component in an AADL model corresponds to application code method stubs in the HAMR generated code. The thread application code is organized into entry points. These are subprograms that are executed when a thread is dispatched. For instance, a periodic thread has an initialize method that is executed when the thread is first initialized. An initialization method is allowed to initialize local variables and write an initial value to output ports. The nominal run-time entry point for a periodic thread is called *time-triggered*. Once initialization is complete, threads enter the await dispatch state. Initial values of in and out ports are available at first dispatch. Therefore, during the initialization phase, input ports are unavailable and cannot be read. In either of these cases, the developer may specify verifiable behaviors within the contract structures that are specific to each entry point. The structure of the entry point contracts in the GCL is specific to the dispatch protocol. Both sporadic and periodic dispatch protocols are supported in the GCL. This is indicated in Figure 10 in the center of the application code box.

Integration Constraints: Integration constraints are invariants on the values flowing across ports. This is illustrated in the top left of Figure 10. Integration constraints specify the requirements and assumptions that refer to a single port. These contracts limit to what other ports the component can correctly connect when it is integrated into a system context. It can be helpful during the integration phase of a multi-vendor development effort to determine if component connections are compatible. An input port integration constraint specifies the components assumptions about what it expects to read on the port. A component must guarantee integration constraints on its output ports.

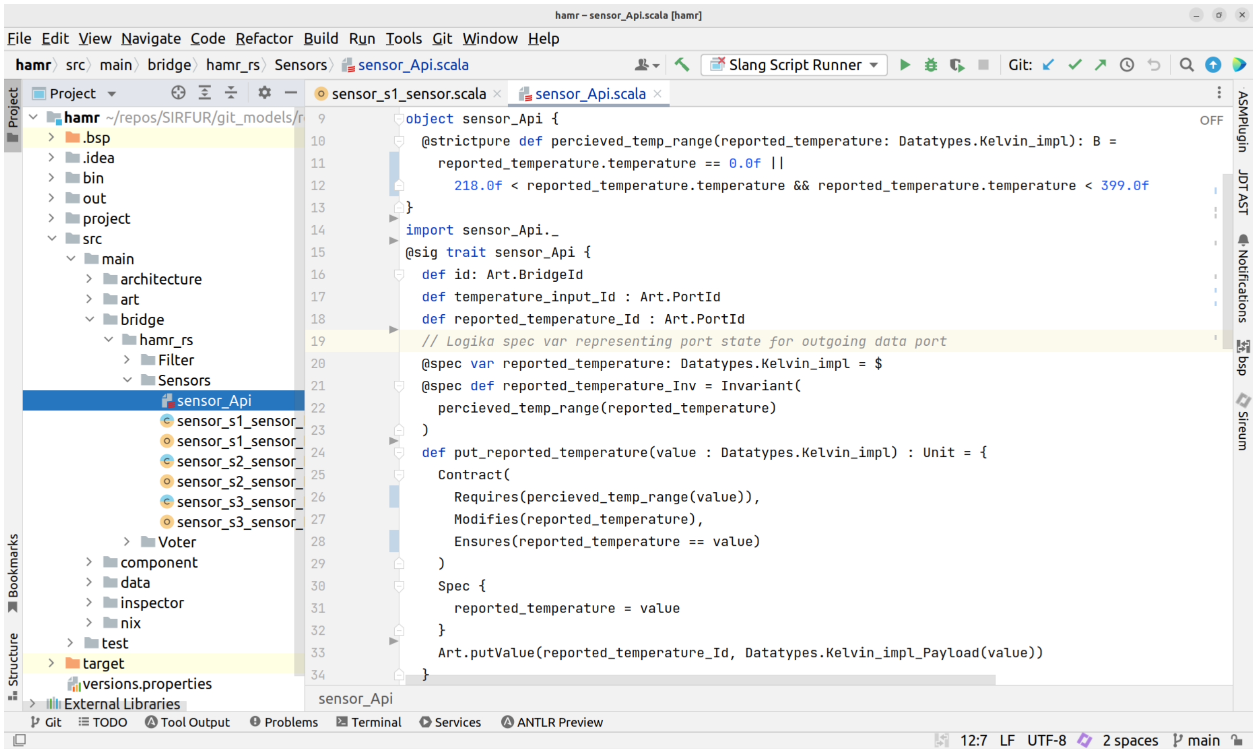
5.3 Verification of Infrastructure Code

We extended the AADL model of the redundant sensors with GCL contracts. This included data invariants, integration constraints, state variables, and entry point contracts. Listing 6 shows the annex that is associated with the sensor thread.

Listing 6: *The GCL annex for the sensor thread showing integration constraints, initialize, and compute entry point contracts.*

```
 1 annex gumbo {**
 2   integration
 3     guarantee perceived_temp_range:
 4       reported_temperature.temperature == f32"0.0" ||
 5       (f32"218.0" < reported_temperature.temperature
 6         && reported_temperature.temperature < f32"399.0");
 7
 8   initialize
 9     modifies reported_temperature;
10     guarantee initialization_temp_to_report "initialization_temp_to_report [Req-TS-2]":
11       reported_temperature.temperature == f32"0.0";
12
13   compute
14     modifies reported_temperature;
15     assume temperature_input_range:
16       f32"218.0" < temperature_input.temperature && temperature_input.temperature < f32
             "399.0";
17
18   guarantee reported_temp:
19     reported_temperature.temperature == temperature_input.temperature;
20 **};
```

HAMR weaves these integration constraints into the API code base as code-level property specifications that mirror the system requirements. Formal verification provided by Logika will check that the behavior of the developer's application code which accesses port values via the API and declares or uses data will satisfy these integration constraints. For example, the integration constraint (`perceived_temp_range` on lines 3–6) is auto-generated as a Logika spec method within the sensor API and the setter method for that port ensures that the reported temperature is constrained by the temperature range defined, as shown in Figure 11.

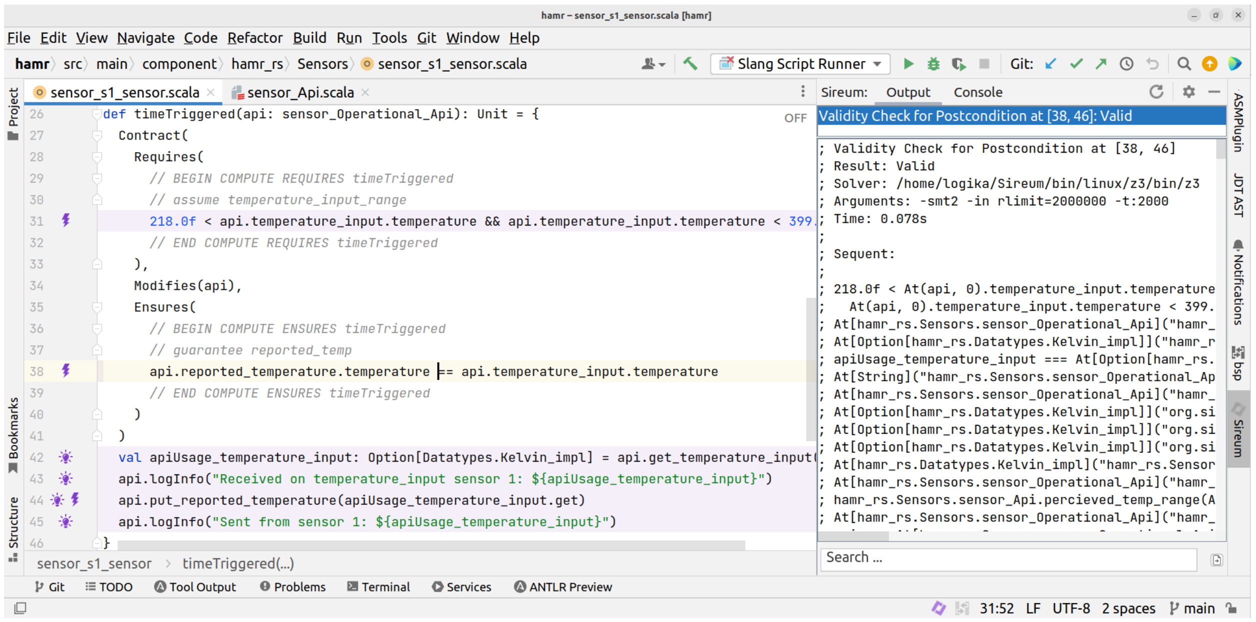


```scala
object sensor_Api {
  @strictpure def percieved_temp_range(reported_temperature: Datatypes.Kelvin_impl): B =
    reported_temperature.temperature == 0.0f ||
      218.0f < reported_temperature.temperature && reported_temperature.temperature < 399.0f
}
import sensor_Api._
@sig trait sensor_Api {
  def id: Art.BridgeId
  def temperature_input_Id : Art.PortId
  def reported_temperature_Id : Art.PortId
  // Logika spec var representing port state for outgoing data port
  @spec var reported_temperature: Datatypes.Kelvin_impl = $
  @spec def reported_temperature_Inv = Invariant(
    percieved_temp_range(reported_temperature)
  )
  def put_reported_temperature(value : Datatypes.Kelvin_impl) : Unit = {
    Contract(
      Requires(percieved_temp_range(value)),
      Modifies(reported_temperature),
      Ensures(reported_temperature == value)
    )
    Spec {
      reported_temperature = value
    }
    Art.putValue(reported_temperature_Id, Datatypes.Kelvin_impl_Payload(value))
  }
```

Figure 11: *The API spec variable, spec method, and setter for sensor output port associated with the integration constraints.*

The entry-point contracts defined on lines 8–19 of Listing 6 are inserted into the appropriate entry-point method for the thread. Each thread method must be defined by a developer. The sensor thread with completed behavioral code and automatically translated GCL contracts for the compute entry point is shown in Figure 12.

The compute assumption (line 15 in Listing 6) is encoded as a *Requires* clause (line 31 of Figure 12) and the compute guarantee is encoded as the *Ensures* clause (line 38 of Figure 12). As the developer inserts application code, Logika verifies the contracts automatically and displays verification results, resulting in immediate feedback as the icons shown on the left of the editor. If a developer chooses to dig deeper into the *summonings* lightning bolt icon, one can view the results of the SMT2 call, the verified sequent, and the symbolic encoding of the source code. A small part of what is available is shown on the right of Figure 12.


hamr – sensor_s1_sensor.scala [hamr]
File Edit View Navigate Code Refactor Build Run Tools Git Window Help
hamr src main component hamr_rs Sensors sensor_s1_sensor.scala
Slang Script Runner
Git:
sensor_s1_sensor.scala
sensor_Api.scala
Sireum: Output Console
OFF
Project
Bookmarks
Structure
ASMPlugin
JDT AST
Notifications
bsp
Sireum
sensor_s1_sensor > timeTriggered(...)
Git TODO Tool Output Problems Terminal Services ANTLR Preview
31:52 LF UTF-8 2 spaces main

```
26  def timeTriggered(api: sensor_Operational_Api): Unit = {
27    Contract(
28      Requires(
29        // BEGIN COMPUTE REQUIRES timeTriggered
30        // assume temperature_input_range
31        218.0f < api.temperature_input.temperature && api.temperature_input.temperature < 399.
32        // END COMPUTE REQUIRES timeTriggered
33      ),
34      Modifies(api),
35      Ensures(
36        // BEGIN COMPUTE ENSURES timeTriggered
37        // guarantee reported_temp
38        api.reported_temperature.temperature == api.temperature_input.temperature
39        // END COMPUTE ENSURES timeTriggered
40      )
41    )
42    val apiUsage_temperature_input: Option[Datatypes.Kelvin_impl] = api.get_temperature_input(
43    api.logInfo("Received on temperature_input sensor 1: ${apiUsage_temperature_input}")
44    api.put_reported_temperature(apiUsage_temperature_input.get)
45    api.logInfo("Sent from sensor 1: ${apiUsage_temperature_input}")
46  }
```

Validity Check for Postcondition at [38, 46]: Valid

```
; Validity Check for Postcondition at [38, 46]
; Result: Valid
; Solver: /home/logika/Sireum/bin/linux/z3/bin/z3
; Arguments: -smt2 -in rlimit=2000000 -t:2000
; Time: 0.078s
;
; Sequent:
;
; 218.0f < At(api, 0).temperature_input.temperature
;   At(api, 0).temperature_input.temperature < 399.
; At[hamr_rs.Sensors.sensor_Operational_Api]("hamr_
; At[Option[hamr_rs.Datatypes.Kelvin_impl]]("hamr_r
; apiUsage_temperature_input === At[Option[hamr_rs.
; At[String]("hamr_rs.Sensors.sensor_Operational_Ap
; At[hamr_rs.Sensors.sensor_Operational_Api]("hamr_
; At[Option[hamr_rs.Datatypes.Kelvin_impl]]("org.si
; At[Option[hamr_rs.Datatypes.Kelvin_impl]]("org.si
; At[Option[hamr_rs.Datatypes.Kelvin_impl]]("org.si
; At[hamr_rs.Datatypes.Kelvin_impl]("hamr_rs.Sensor
; At[hamr_rs.Sensors.sensor_Operational_Api]("hamr_
; hamr_rs.Sensors.sensor_Api.percieved_temp_range(A
; At[hamr_rs.Sensors.sensor_Operational_Api]("hamr_
```

Search ...

Figure 12: *The sensor time-triggered compute entry point with Logika contracts and SMT2 summoning results shown.*

6 Incorporating Faults into HAMR Generated Model

There are multiple ways that one could verify that the management system responds appropriately to faults. Testing is a way to validate the system: the behavior is what you expect given specific conditions or inputs. The HAMR toolset can automatically generate testing frameworks for the system infrastructure and application code. We prototyped an extension of the framework to provide safety-based verification: the behavior formally conforms to the specifications in the presence of faults. We based our approach on previous model-based safety analysis work [36, 37] which specifies faults in first order logic, constraints outputs to conform to these specifications, and then attempts to verify system properties when those faults are active in the system.

To accomplish this within the HAMR framework, we prototyped an approach that introduces *havoc* (i.e., injected faults) into the code base. We constrain the output port value of the component with the active fault. Any receiving component could receive a faulty value from the faulty component. To constrain port values, we extended the HAMR-generated API code-base.

The AADL run-time (`Art`) module in the HAMR-generated code contains the architecture description files, collections defining connections and ports, and registers dispatch properties over threads. When a thread accesses or sends data to a port, it uses an API as a bridge to the `Art` modules. Each thread has an API that links to the AADL run-time (`Art`) classes as shown on the left of Figure 13. The sensor thread API, for example, declares and initializes the port identifiers, and defines the methods used to write to and read from ports. We defined a new intermediate fault API that can easily be woven into HAMR generated infrastructure code. This is shown on the right of Figure 13. The nominal (fault-free) API is unchanged, as is the `Art` module for the system.

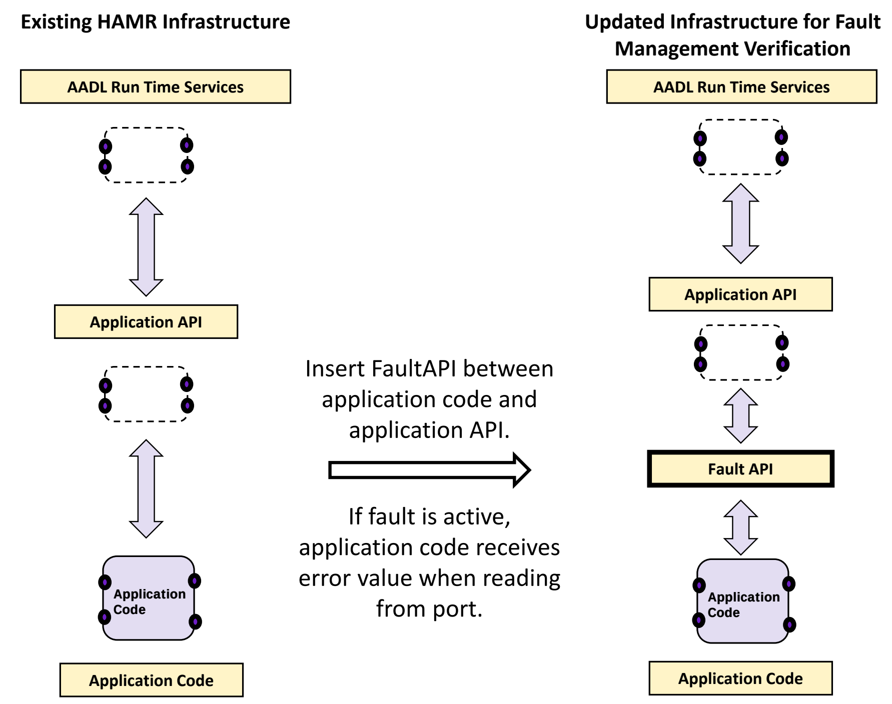

Existing HAMR Infrastructure
AADL Run Time Services
Application API
Application Code
Application Code
Updated Infrastructure for Fault Management Verification
AADL Run Time Services
Application API
Fault API
Application Code
Application Code
Insert FaultAPI between application code and application API.
If fault is active, application code receives error value when reading from port.

Figure 13: *HAMR infrastructure communications, showing where the fault API is defined.*

A fault API must be specific for each component that will be analyzed using this approach. Each fault API will define alternate `get` methods to access values on input ports. For our prototype, we focused on the voter thread. The voter component has three input ports, one for each sensor value. The fault API defines three *getter* methods specific to these input ports that will be called within the voter thread implementation code. Listing 7 shows one of the three methods. The other two are very similar.

Listing 7: *The fault API for the voter component.*

```
object triplex_majority_voter_thread_havoc_api {
  var hs1_reported_temperature: Datatypes.Kelvin_impl = Datatypes.Kelvin_impl(f32"0")
  var hs2_reported_temperature: Datatypes.Kelvin_impl = Datatypes.Kelvin_impl(f32"0")
  var hs3_reported_temperature: Datatypes.Kelvin_impl = Datatypes.Kelvin_impl(f32"0")

  def get_hs1_reported_temperature(): Datatypes.Kelvin_impl = {
    Contract(
      Modifies(hs1_reported_temperature),
      Ensures(hs1_reported_temperature.temperature <= f32"218.0" || f32"399.0" <=
          hs1_reported_temperature.temperature)
    )
    var faultyVar: F32 = f32"0.0"
    faultyVar = Faults.FaultDefs.outOfRangeF32(Datatypes.Constraints.getLowerBound(),
        Datatypes.Constraints.getUpperBound())
    hs1_reported_temperature = Datatypes.Kelvin_impl(faultyVar)
    return hs1_reported_temperature
  }
  ...
}
```

The application code of a thread calls the fault API when accessing a value from an input port. The fault API returns an error value to the thread (lines 12–14). The fault defined for the redundant sensors is`OutOfRange`: the values read from the sensor input ports will be outside of the sensors' operating (and expected) range. The call to the fault definition can be seen on line 12. The contracts on the havoc method ensure that the result of the method call results in an erroneous temperature value that is out of range.

In the fault analysis prototype, the application code calls the intermediate API to get the input port values. Likewise, the fault management contracts refers to the intermediate API sensor values to reason about the behavior of the component. Future automation can hide these details from the developer and allow them to easily turn off and on the fault analysis without requiring application code to change.

To test our prototype, we adjust the application code to call the intermediate API to get the value from the data port. We added contracts to verify that the value received from the intermediate API was out of range. This is shown in Listing 8 on lines 11–13.

Listing 8: *The voter thread contracts and behavior code adjusted for havoc.*

```
1  def timeTriggered(api: triplex_majority_voter_thread_Operational_Api): Unit = {
2    Contract(
3      Requires(),
4      Modifies(hs1_reported_temperature, hs2_reported_temperature,
         hs3_reported_temperature),
5      Ensures(
6        hs1_reported_temperature.temperature == hs2_reported_temperature.temperature
7          imply_: (api.result_temperature.temperature == hs1_reported_temperature.
             temperature), //Req-RM-1
8        ... // All other contracts refer to intermediate API values
9
10         // Testing fault injection: sensor values *not* within operational range.
11         hs1_reported_temperature.temperature <= f32"218.0"
12             || f32"399.0" <= hs1_reported_temperature.temperature,
13         hs2_reported_temperature.temperature <= f32"218.0"
14             || f32"399.0" <= hs2_reported_temperature.temperature,
15         hs3_reported_temperature.temperature <= f32"218.0"
16             || f32"399.0" <= hs3_reported_temperature.temperature
17     ))
18
19   val s1_temperature_inputOpt = get_hs1_reported_temperature()
20   val s2_temperature_inputOpt = get_hs2_reported_temperature()
21   val s3_temperature_inputOpt = get_hs3_reported_temperature()
22   ... // Implementation code remains the same
```

The contracts fully verify: the havoc methods are injecting erroneous values into the application code. The rest of the implementation code remains the same. The goal is to formally verify that the fault management component will manage the errors appropriately. To accomplish this, we rewrote the contracts to point to the error-adjusted port readings to perform verification[3]. All triplex mode contracts for the voter fault management thread verify when sensors report values that are out of range. This provides additional confidence that the triplex implementation of the thread performs fault mitigation as per the requirements.

[3]Future work will also hide these adjustments from the developers.

7 Code-level Design of Duplex Mode Implementation

While model-based approaches provide numerous benefits, this is not the only entry point to this toolset. We implemented the duplex redundancy management methods directly in Slang to illustrate an approach that moves from requirements directly to developing source code. We developed the code using the Sireum IDVE, a customization of the popular IntelliJ IDE. Logika verification capabilities are incorporated into the IDVE. Developers can insert specifications directly into the code base as contracts. We focused our attention on the duplex mode of the voter thread.

The following requirements guide development of the implementation to address a "stuck at" sensor that will drive the system into operating in a degraded duplex mode:

- Req-RM-8: Redundancy management will switch to duplex mode only when all of the following conditions are met:
  - A single sensor perceived temperature has disagreed with the majority for 3 temperature readings (this reading inclusive).
  - The erroneous value on this sensor has been equivalent for all 3 temperature readings (this reading inclusive).
  - The remaining two sensors agree on a perceived temperature value for all 3 temperature readings (this reading inclusive).
- Req-RM-9: If redundancy management switches to duplex mode, then the redundancy management status will indicate which sensor displays a stuck value.
- Req-RM-10: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the temperature reported by the voter will be the midpoint value of the two servicing sensor input values.
- Req-RM-11: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the status of the voter will indicate that it is in a degraded mode of operation.

Three conditions outlined in Req-RM-8 must be met for the system to initially enter duplex mode. A single sensor reading has been in disagreement for three readings, this value has been equivalent all three readings, and the other servicing sensors are in agreement, albeit on a different value. Once the system is in duplex mode, Req-RM-10 states that the system can enter a degraded mode of operation if the two servicing sensors report values that are in disagreement.

Some classes of errors cannot be detected or mitigated without a knowledge of past state in the system. For example, sensors or buffers that are stuck at a particular value, or sensors that are out of calibration. For example, a `StuckAt` error cannot be detected without observing previous temperature readings and inferring that the sensor output may be stuck. Error mitigation may be performed in multiple places. For example, it could be performed in the component that receives voter input, examines previous states, and performs some decision strategy. However, without being able to examine other input streams, an external stable environmental condition (e.g., temperature) cannot be distinguished from a `StuckAt`, unless yet another mechanism such as noise, is added to the input signal at the sensor.

Mitigating these errors at the voter also accounts for errors that would mimic `StuckAt`, such as permanent errors in the memory buffers for the data transfers. To mitigate the errors, the voter component requires access to past state. We implemented state history as a queue in which the voter saves temperature readings from each sensor as input is sampled on the voter data port. Queued state information can be used to mitigate multiple types of errors, including `StuckAt`, `OutOfCalibration`, and `BoundedValueChange`.

We include here assumptions and simplifications we made to support our implementation:

- We assume multiple faults cannot be active on the same port simultaneously.
- State history will be required for any error-mitigating component and for any fault injection scheme to accurately model common critical system faults and errors.
- History is specific to each port of interest in the model, so the developer will need to determine state history requirements for each port. In this example, sensor 1–3 output ports will each have its own associated history.

7.1 Using a Circular Queue to Model State History

A basic ring buffer requires an extension to be used for state history. We defined a *peek* method that returns the value at a given location in the queue, but does not dequeue that value. Given that n steps of state history can be saved on the queue, checking history does not use the *dequeue* method, but rather needs to only peek at elements in the queue.

Elements are enqueued at the rear of the queue and the rear pointer is incremented. The front pointer index holds the oldest element. Two subsequent calls to the enqueue method (*enq*) shown in Figure 14 result in the queue to the right. As we "peek" at states in the queue, we wish to associate `THIS` state with the rear pointer, which is the relationship: `THIS = rear - 1`.

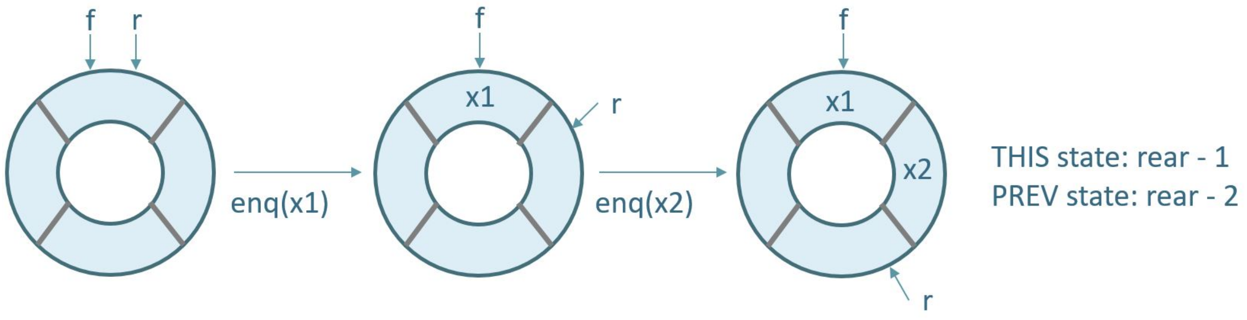


Figure 14: *Enqueue of elements in circular queue and relationship to states.*

The implementation of the peek method takes as a parameter a value i which will be decremented from THIS state. In other words, if one wishes to view THIS state, then $i = 0$. The parameter takes the values from 0 to n inclusively, where n is the number of states the queue will hold. The implementation is shown in Listing 9.

We used Logika to formally verify this peek method, as well as the rest of the ring buffer implementation.

Listing 9: *The peek method for a circular queue. The parameter enumerates the newest element (0), previous element (1), and up to the n^{th} newest element.*

```
def peek(i: Z): N16 = {
    Contract(
      Requires(0 <= i && i <= MAX),
      Modifies(),
      Ensures(Res[N16] == queue((rear-1-i)%QueueSize))
    )
    return queue((rear-1-i)%QueueSize)
  }
```

7.2 Mitigation Strategies & State History

While an error such as `StuckAt` cannot be definitively determined, it can be implied by the resulting erroneous behavior of the system. For example, if a given sensor data stream remains unchanged for n steps, and a majority of the other sensors readings are both changing and seem to be agreeing, then one may infer that the unchanging sensor (or something in the sensor stream, such as communications or intervening memories or buffers) is stuck. The determination of what n should be in a given system depends on the system dynamics. We therefore developed an approach that allows n to be specified in the architecture model.

We also considered how to compare port values in each of the n port readings. If the temperature remains constant and all sensors report the same value for multiple calls, we do not wish to report that a possible `StuckAt` error has occurred. When a single sensor reports the same value for n steps and the remaining sensors agree on a different value in each of those steps, then we infer `StuckAt`. To report a `StuckAt` error, we defined three additional status values: `sensor1:StuckAt`, `sensor2:StuckAt`, and `sensor3:StuckAt`. The voter will indicate which sensor may be problematic via the status output.

To implement the mitigation, we also studied requirements for the state history queue. A queue of size *n+1* for each incoming sensor port is declared local to the voter component. The state of the incoming ports will be saved on their respective local queues at each dispatch of the voter thread. Recall that `THIS` state corresponds with the newest element inserted in the queue, equivalently at index $rear - 1$. The n^{th} newest state may be found at $rear - n$ index. Instead of dequeueing elements at the rear of the queue (as performed in the oldest-first implementation of the ring buffer), we will peek at elements from newest to oldest as described in Appendix 7.1.

The implementation of the fault management thread is shown in Listing 10.

Listing 10: *Duplex-triplex voter implementation in Slang.*

```
def voter_timeTriggered(): (Status.Type, N16) = {
  sensor1PortHistory.enqueue(sensor1Temperature)
  sensor2PortHistory.enqueue(sensor2Temperature)
  sensor3PortHistory.enqueue(sensor3Temperature)

  // Inject faults on ports:
  // Previous state corresponds to portHistory.peek(1)
  sensor1Temperature = havocPort1(sensor1outOfRange,sensor1stuckAt,sensor1PortHistory
      .peek(1))
  sensor2Temperature = havocPort2(sensor2outOfRange,sensor2stuckAt,sensor2PortHistory
      .peek(1))
  sensor3Temperature = havocPort3(sensor3outOfRange,sensor3stuckAt,sensor3PortHistory
      .peek(1))

  // Duplex Implementation

  // if conditions apply, then stuck at
  // if prev(stuckAt) and current(stuckAt) and others disagree, then degraded
  if(singleSensorDisagreesNReadings(sensor1PortHistory, sensor2PortHistory,
      sensor3PortHistory)
      && valueEquivalentNReadings(sensor1PortHistory)
      && servicingSensorsAgreeNReadings(sensor2PortHistory, sensor3PortHistory)){
    statusHistory.enqueue(Status.Sensor1StuckAt)
    return (Status.Sensor1StuckAt, midpoint(sensor2Temperature, sensor3Temperature))
  } else if(!servicingSensorsAgreeNReadings(sensor2PortHistory, sensor3PortHistory)
      && valueEquivalentNReadings(sensor1PortHistory)
      && (statusHistory.peek(1) == Status.Sensor1StuckAt || statusHistory.peek(1) ==
          Status.Sensor1StuckAtDegraded)){
    statusHistory.enqueue(Status.Sensor1StuckAtDegraded)
    return (Status.Sensor1StuckAtDegraded, midpoint(sensor2Temperature,
        sensor3Temperature))
  }

  // ... similar blocks for sensor 2 and 3 state history queues

  // Triplex implementation

  // Majority voting implementation
  val voter_result_temperature: N16 =
  if(sensor1Temperature == sensor2Temperature) {
    sensor1Temperature
  } else if(sensor1Temperature == sensor3Temperature) {
    sensor3Temperature
  } else if (sensor2Temperature == sensor3Temperature) {
    sensor2Temperature
  } else {
    n16"0"
  }

  // Voter status implementation
  val voter_status:Status.Type =
    if(allAgree(sensor1Temperature, sensor2Temperature, sensor3Temperature)) {
      Status.Good
    } else if (onlyTwoAgree(sensor1Temperature, sensor2Temperature,
        sensor3Temperature)){
      Status.SingleFault
    } else {
      Status.Faulty
    }
  val results = (voter_status, voter_result_temperature)
  return results
}
```

7.3 Havoc for Duplex Mode Faults

With a `StuckAt` error, the output of the sensor does not change over time despite temperature changes in the environment. A simple Slang implementation of such an error is shown in Listing 11. If the error is active, the output of the sensor is the same as its previous temperature output.

Listing 11: *An initial Slang implementation of the* `StuckAt` *fault for 16-bit unsigned integers.*

```
def stuckAtUnsigned(previousValue:N16): N16 = {
  Contract(
    Requires(),
    Modifies(),
    Ensures(Res[N16] == previousValue)
  )
  var faultyValue: N16 = previousValue
  return faultyValue
}
```

We extended the previously defined havoc method to allow for multiple fault definitions as shown in Listing 12.

Listing 12: *The havoc method for port 1: this method injects a fault into the port when fault is activated.*

```
def havocPort1(outOfRangeHavoc: B, stuckAtHavoc:B, prevValue: N16): N16 = {
  var havocTemperature: N16 = sensor1Temperature
  // Prevent multiple faults on a single port
  if(outOfRangeHavoc && stuckAtHavoc){
    return havocTemperature
  }
  if(outOfRangeHavoc){
    havocTemperature = outOfRangeUnsigned(LOW, HIGH)
  } else if(stuckAtHavoc){
    havocTemperature = stuckAtUnsigned(prevValue)
  }
  return havocTemperature
}
```

We introduced havoc within the voter code base and tested our implementation using Slang unit tests. The contract clauses corresponding to triplex mode requirements were verified, but any specifications of the duplex mode requirements demanded insight into the implementation details. Without the ability to state temporal logic, we could not fully verify the duplex implementation. The queue itself is verified, as are all helper methods. Future work will focus on specification of temporal requirements such as these.

8 Real World Cyber Physical System Challenges: Floating Points and Units

Research into CPS development, verification, and fault analysis may rely on gross simplifications, such as use of Real number ($\mathbb{R}$) theory, assuming nominal behavior, and consistent physical units. Since we did not want to gloss over important real-world complexities, we analyzed our redundant sensor system assumptions and simplifications to determine how the toolset and algorithms needed to support real world CPS development.

Fuzzy Comparators:

Since the results of floating-point operations are rounded to the nearest value available in the floating point representation, math doesn't behave like it does with real numbers or integers. In particular, comparing their exact values does not work as expected. As long as the imprecision due to rounding stays small (where "small" is domain and use-case specific), it may be ignored. However, when comparing floating point values for equality, such as when calculating the same result through different correct methods, we must account for this imprecision. For example, language, hardware, compiler, and function differences may produce slightly different results for the product 0.1×10 than the sum $\sum_{n=1}^{10} 0.1$. While these values should be equal, they may not be at the bit level, and a simple equality comparison will fail.

The redundant sensor voter and sensor specifications require many comparisons and equality statements. To handle the imprecision, we defined fuzzy comparator strictpure methods to reference in the contracts, as shown in Listing 13.

Listing 13: *Fuzzy comparators for floating point values.*

```
@strictpure def fabs(num: F32): F32 = {
  return if(num < f32"0.0") -num else num
}
@strictpure def approximatelyEqual(a: F32, b: F32, epsilon: F32): B = {
  return fabs(a - b) <= ((if (fabs(a) < fabs(b)) fabs(b) else fabs(a)) * epsilon)
}
@strictpure def essentiallyEqual(a: F32, b: F32, epsilon: F32): B = {
  return fabs(a - b) <= ((if (fabs(a) > fabs(b)) fabs(b) else fabs(a)) * epsilon)
}
@strictpure def fuzzyGreaterThan(a: F32, b: F32, epsilon: F32): B = {
  return (a - b) > ((if (fabs(a) < fabs(b)) fabs(b) else fabs(a)) * epsilon)
}
@strictpure def fuzzyLessThan(a: F32, b: F32, epsilon: F32): B = {
  (b - a) > ((if (fabs(a) < fabs(b)) fabs(b) else fabs(a)) * epsilon)
}
```

The `aboveRange` method definition in Listing 14 shows the use of the fuzzy comparators within the method contract and conditional statements in the implementation.

Other than the change to using fuzzy comparators, the contract and implementation of the `aboveRange` method remains unchanged from earlier versions of the voter fault model. These methods and contracts allow us to verify that the 32-bit floating point version of the redundancy management system is implemented according to the specifications.

Listing 14: *Fuzzy comparators used in the "above range" fault definition.*

```
def aboveRange(bound:F32): F32 = {
 Contract(
   Requires(FuzzyComparatorsFloat32.fuzzyLessThan(ABSOLUTE_ZERO, bound, PRECISION),
        FuzzyComparatorsFloat32.fuzzyLessThan(bound, ABSOLUTE_MAXIMUM, PRECISION)),
   Modifies(),
   Ensures(FuzzyComparatorsFloat32.fuzzyGreaterThan(Res[F32], bound, PRECISION))
 )
 var faultyVal: F32 = F32.random
 if(!FuzzyComparatorsFloat32.fuzzyGreaterThan(faultyVal, bound, PRECISION)){
   faultyVal = ABSOLUTE_MAXIMUM
 }
 return faultyVal
}
```

Conversions to SI: The Mars Climate Orbiter was launched on December 11th, 1998 on a mission to orbit Mars as the first interplanetary weather satellite and was to provide a communications relay for another spacecraft, the Mars Polar Lander. The Climate Orbiter was lost on September 23rd, 1999 when it failed to enter an orbit around Mars. It crashed into Mars, destroying the $125 million craft, part of a $328 million mission. The root cause of the failure was a software function that was, according to the software interface specifications, to provide readings using Newton-Seconds (N*s), but instead provided output in pound-force-seconds [7]. This is but one of many examples where unit confusion within a system resulted in loss-of-mission.

When developing portable, reusable CPSs, or components for those systems, unit specification and conversions must be addressed. Systems using these components may differ in their specifications. Portable component specifications must explicitly capture input units and guaranteed output units. To demonstrate one way to formally accomplish this, we developed a reusable object to handle unit conversions, and then refactored the redundant sensor triplex model as shown in Figure 15.

The sensing devices used on a system may be configured to report the temperature using Fahrenheit, Celsius, or Kelvin. The fault management system supports any of these units. The threads that sample the readings from the devices will convert the value (if needed) to International System of Units (SI), and the voter component will perform all computations using SI. The configurable converter thread takes the result of the voter and converts it to whatever unit is expected by the consumer. We implemented a `UnitConversion` object that takes as input a value to convert and the unit as an enumerated type (Figure 16). The method `toSI` converts this value to SI.

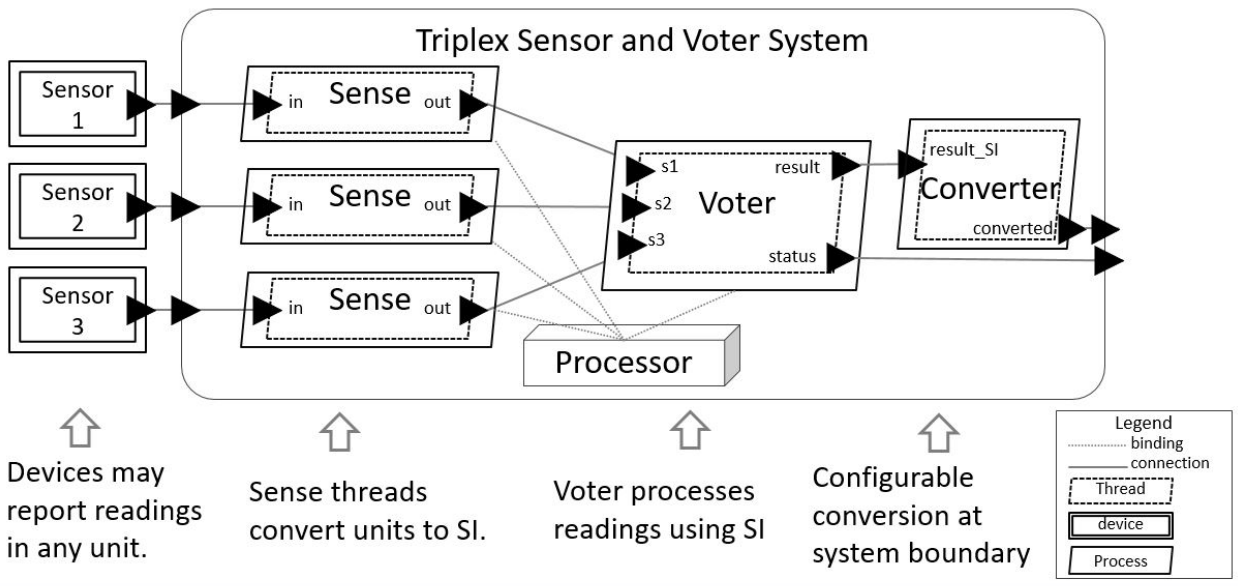
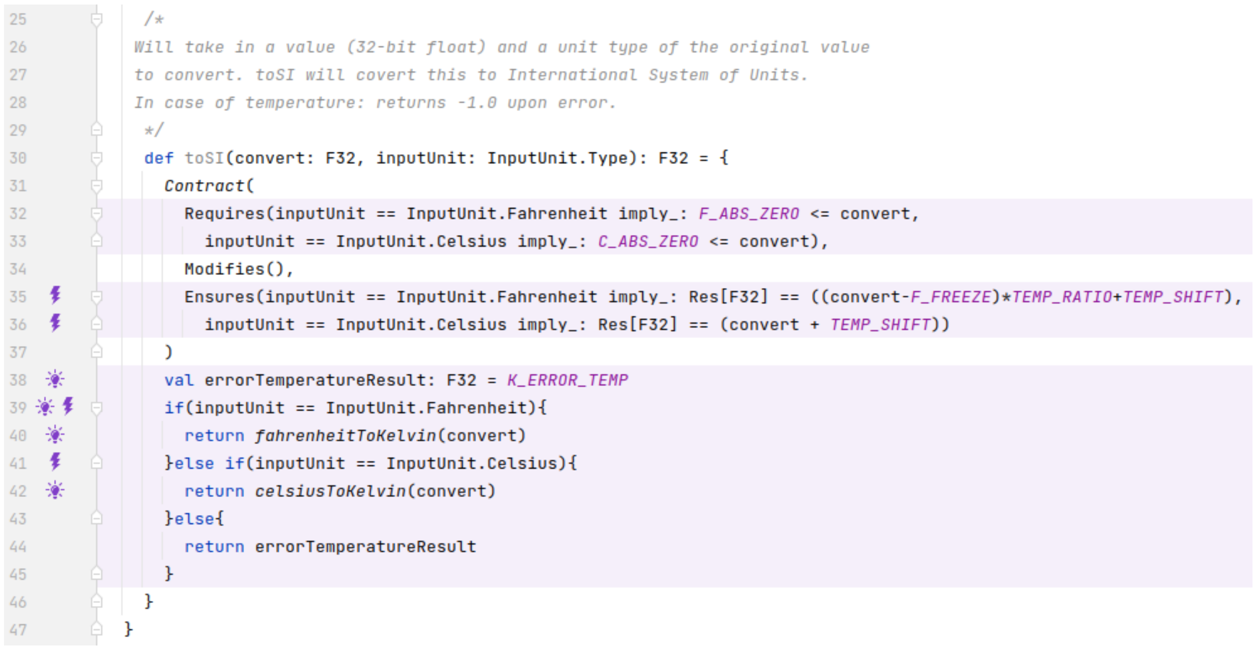

Triplex Sensor and Voter System
Sensor 1
Sensor 2
Sensor 3
in
Sense
out
s1
s2
s3
result
status
Voter
result_SI
Converter
converted
Processor
Devices may report readings in any unit.
Sense threads convert units to SI.
Voter processes readings using SI
Configurable conversion at system boundary
Legend
binding
connection
Thread
device
Process

Figure 15: *The redundant sensor system reconfigured to support any unit, but perform computations using SI.*



```
 /*
Will take in a value (32-bit float) and a unit type of the original value
to convert. toSI will covert this to International System of Units.
In case of temperature: returns -1.0 upon error.
 */
 def toSI(convert: F32, inputUnit: InputUnit.Type): F32 = {
   Contract(
     Requires(inputUnit == InputUnit.Fahrenheit imply_: F_ABS_ZERO <= convert,
       inputUnit == InputUnit.Celsius imply_: C_ABS_ZERO <= convert),
     Modifies(),
     Ensures(inputUnit == InputUnit.Fahrenheit imply_: Res[F32] == ((convert-F_FREEZE)*TEMP_RATIO+TEMP_SHIFT),
       inputUnit == InputUnit.Celsius imply_: Res[F32] == (convert + TEMP_SHIFT))
   )
   val errorTemperatureResult: F32 = K_ERROR_TEMP
   if(inputUnit == InputUnit.Fahrenheit){
     return fahrenheitToKelvin(convert)
   }else if(inputUnit == InputUnit.Celsius){
     return celsiusToKelvin(convert)
   }else{
     return errorTemperatureResult
   }
 }
}
```

Figure 16: *The toSI method takes a value and unit and converts to SI. The current implementation only deals with temperature, but can be extended easily for other domains and units.*

9 Future Steps and Conclusion

In this work, we focused on model-based early analysis and software verification. HAMR generated code includes configuration files for deployment on various operating systems and microkernels (e.g., Linux, seL4).

As we performed this work, we were pleased at how well the formal verification identified issues that traced back to both requirements and implementation errors and edge cases that are easy to miss in traditional testing-based approaches.[4]

As we developed and refined the system from the MBE level down through the implementation, analyzing and verifying along the way, we observed that our requirements, design, and implementation became more robust and amenable to modularization and extension rather than brittle and inflexible. For example, when we added additional fault handling capabilities, or switched to use the unit conversion, the changes were localized, and we did not have to update internal contracts, and the former verification approaches still held. As we refactored components, we did not have to rewrite tests, since the contracts and automated formal verification followed the code.

Future work includes incorporating this subsystem into a larger context, deploying it on hardware and devices, and extending the testing and verification into that larger CPS context. Other future work includes complete verification of duplex implementation using interprocedural verification, automated user-friendly support for fault injection and safety analysis, and support for analyzing transient fault occurrence.

A Redundant Sensor Requirements

B System Goals

The high level goals (G) of the system are:

- G1–System provides a logical representation of environmental temperature with a specified accuracy and precision (integrity) in presence of 0 or 1 arbitrary faults (availability).
- G2—System should be able to operate in a degraded mode with a reduction of accuracy and precision in the presence of certain dual uncorrelated failures.
- G3—The internal state of the sensors should not cross the system boundary (confidentiality).
- G4—System should indicate to the user whenever a degraded mode is not possible and an accurate and precise logical representation of environmental temperature cannot be provided.

C Operational Concepts

The following use and exception cases describe how the redundancy management system determines temperature output. The use cases are summarized in Table 1 according to the system goal the use case covers.

[4]At the same time, we remained somewhat dismayed at how easy it is to make those mistakes in the first place.

Redundant Sensor System Use Cases		
ID	System Goal	Description
C.1	G1	Normal fault-free operation of sensor system
C.2	G1	Single faulty sensor value
C.3	G4	Dual sensor errors in triplex mode
C.4	G2	Dual sensor errors in duplex mode
C.5	G4	Undetected error in triplex or duplex mode

Table 1: Summary of use cases for redundant sensor system

C.1 Use Case: Normal Fault-free Operation of Sensor System

This use case describes the normal operation of the redundant sensor system by the system user in the absence of active faults.

- Related system goals: G1
- Primary actor: Sensor system user
- Precondition:
  - Sensor system is ready for use
  - Sensor system is turned off
- Postcondition:
  - Sensor system is turned off
- Main success scenario:
  1. User starts up the sensor system
  2. Sensors initialize and enter normal mode of operation
  3. Redundancy management initializes and enters triplex mode of operation
  4. Sensors provides environmental temperature to redundancy management
  5. Redundancy management provides temperature value to user
  6. Redundancy management status reflects 'Good.'
  7. Repeat steps 4 through 6 until user shuts system down

C.2 Use Case: Single Faulty Sensor Value

This use case describes the triplex mode of operation of the redundant sensor system by the system user wherein a single sensor value is in disagreement. The redundancy management system assumes that this corresponds to a faulty sensor.

- Related system goals: G1
- Primary actor: Sensor system user
- Precondition:
  - Sensor system is ready for use
  - Sensor system is turned off
- Postcondition:
  - Sensor system is turned off

- Main success scenario:
  1. User starts up the sensor system
  2. Sensors initialize and enter normal mode of operation
  3. Redundancy management initializes and enters triplex mode of operation
  4. Sensors provides environmental temperature to redundancy management
  5. Redundancy management provides temperature value to user
  6. Redundancy management status reflects 'Good.'
  7. Repeat steps 4 through 6 until redundancy management determines that a single sensor is in disagreement
  8. Redundancy management provides majority agreement value to user
  9. Redundancy management status reflects 'SingleFault'
  10. Repeat steps 4 and 6–8 until user powers down system

C.3 Use Case: Dual Sensor Errors Triplex Mode

This use case describes the triplex mode of operation of the redundant sensor system by the system user wherein no sensor values are in agreement and the conditions have not been met to switch to duplex degraded mode.

- Related system goals: G4
- Primary actor: Sensor system user
- Precondition:
  - Sensor system has been turned on
  - Sensor system has been initialized
  - Redundancy management system status is 'Good'
- Postcondition:
  - Sensor system is turned off
- Main success scenario:
  1. Sensors provides environmental temperature to redundancy management
  2. Redundancy management determines that a single sensor is in disagreement
  3. Redundancy management provides majority agreement value to user
  4. Redundancy management status reflects 'SingleFault'
  5. Repeat steps 1 through 4 until redundancy management determines that no majority consensus can be reached
  6. Redundancy management provides temperature value of zero to user
  7. Redundancy management status reflects 'Faulty'
  8. Repeat steps 1 and 6–7 until user shuts system down

C.4 Use Case: Dual Sensor Errors Duplex Mode

This use case describes the duplex mode of operation of the redundant sensor system by the system user wherein no sensor values are in agreement and the conditions have been met to switch to duplex degraded mode.

- Related system goals: G2
- Primary actor: Sensor system user
- Precondition:
  - Sensor system has been turned on
  - Sensor system has been initialized
  - Redundancy management system status is 'Good'

- Postcondition:
  - Sensor system is turned off
- Main success scenario:
  1. Sensors provides environmental temperature to redundancy management
  2. Redundancy management determines that a single sensor is in disagreement
  3. Redundancy management provides majority agreement value to user
  4. Redundancy management status reflects 'SingleFault'
  5. Repeat steps 1 through 4 until redundancy management determines that the criteria has been met for degraded duplex mode
  6. Redundancy management provides as temperature value the midpoint of the two non-faulty sensors
  7. Redundancy management status reflects 'StuckAt' for the sensor whose temperature value input is erroneous
  8. Repeat steps 1 and 6–7 until user shuts system down

C.5 Use Case: Undetected error in triplex or duplex mode

This use case describes an undetected error during either triplex or duplex mode wherein the sensors report out of range erroneous values that the voter agrees upon. The redundancy management system assumes that this corresponds to more than one faulty sensor.

- Related system goals: G4
- Primary actor: Sensor system user
- Precondition:
  - Sensor system has been turned on
  - Sensor system has been initialized
  - Redundancy management system status is 'Good'
- Postcondition:
  - Sensor system is turned off
- Main success scenario:
  1. Sensors provides environmental temperature to redundancy management where that temperature is outside of the range of operational temperature
  2. Redundancy management determines that multiple sensors are in agreement
  3. Redundancy management provides majority agreement value to user
  4. Redundancy management status reflects either 'Good' or 'SingleFault'
  5. The filter component tests temperature against operational range and finds error
  6. The filter changes system output temperature to zero
  7. The filter changes system status output to 'Faulty'
  8. The filter reports an error
  9. Repeat steps 1–8 until user shuts system down

D Redundant Sensor System Assumptions and Requirements

The following sections describe the entities of the redundant sensor system: the sensors, redundancy management, and the filter. The environmental assumptions made about the system and each entity are listed, along with requirements of the entities.

D.1 System Environmental Assumptions

The following Environmental Assumptions (EAs) are made in the Sensor System (SS):

- EA-SS-1: The operational temperature will always be greater than 218 Kelvin.
- EA-SS-2: The operational temperature will always be less than 399 Kelvin.

The Redundant Sensor System (RSS) adheres to the following requirements (Req):

- Req-RSS-1 If the environmental temperature falls outside of the operational range, then the system will report an error.

D.2 Temperature Sensor

The temperature sensor provides the current ambient air temperature to the redundancy management component. The sensed value is of type N16 (unsigned 16-bit integer) and the units are Kelvin. The temperature range is $(218, 399)$. The sensor component is only rated for environmental values within this range. If the environmental temperature is beyond this range, no guarantees can be made regarding the behavior of the sensor. The physical interpretation of the temperature value is the current ambient air temperature outside of the sensor.

The following EA are made in the temperature sensor (TS):

- EA-TS-1: The environmental temperature will always be greater than 218 Kelvin.
- EA-TS-2: The environmental temperature will always be less than 399 Kelvin.

The temperature sensor (TS) adheres to the following requirements (Req):

- Req-TS-1: The perceived temperature will be provided to the redundancy management system in degrees Kelvin.
- Req-TS-2: If the sensor is in initialization mode, then the perceived temperature provided to the voter will be zero.
- Req-TS-3: If the sensor is in operational mode, then the perceived temperature provided to the voter will be strictly within the range of 218 K to 399 K.
- Req-TS-4: If the sensor is in operational mode, then the perceived temperature provided to the voter will be the same as the environmental temperature.

D.3 Redundancy Management System

The redundancy management system, or voter, takes input from three temperature sensors and will output a temperature value. When all sensors agree, the temperature output is equivalent with agreed-upon temperature. The redundancy management has two modes of operation. In triplex mode, the temperature output is based on majority voting. In certain failure scenarios, the management system will switch to duplex mode of operation and the temperature output is the midpoint value of two temperature inputs. The conditions required for duplex mode operation indicate that there may be a stuck value error where the temperature sensor delivers service whose value stays constant starting with a given service item. When these conditions are met, we state that the status of the voter is stuck with regard to a specific sensor. The duplex mode and associated failure scenarios is defined in the requirements. The voter also sends a status output that indicates faulty status of the sensors.

The following EA are made in the Redundancy Management (RM) system:

- EA-RM-1: The perceived temperature will always be greater than or equal to 0 Kelvin.
- EA-RM-2: The perceived temperature will always be less than or equal to 65,535 Kelvin.

The RM system adheres to the following requirements (Req):

- Req-RM-1: Redundancy management will report the temperature agreed upon by the majority of the sensors whenever majority agreement occurs.
- Req-RM-2: If redundancy management is in triplex mode and the status indicates 'Faulty,' then the redundancy management temperature reported will be zero.
- Req-RM-3: If redundancy management is in initialization mode, then the redundancy management temperature reported will be zero.
- Req-RM-4: If redundancy management is in initialization mode, then the redundancy management status output will indicate 'Init.'
- Req-RM-5: If all sensor perceived temperature values agree, then the redundancy management status will indicate 'Good.'
- Req-RM-6: If a single perceived temperature value disagrees, then the redundancy management status will indicate 'SingleFault.'
- Req-RM-7: If more than one perceived temperature value disagrees while in triplex model, then the redundancy management status will indicate 'Faulty.'
- Req-RM-8: Redundancy management will switch to duplex mode only when all of the following conditions are met:
  - A single sensor perceived temperature has disagreed with the majority for 3 temperature readings (this reading inclusive).
  - The erroneous value on this sensor has been equivalent for all 3 temperature readings (this reading inclusive).
  - The remaining two sensors agree on a perceived temperature value for all 3 temperature readings (this reading inclusive).
- Req-RM-9: If redundancy management switches to duplex mode, then the redundancy management status will indicate which sensor displays a stuck value.
- Req-RM-10: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the temperature reported by the voter will be the midpoint value of the two servicing sensor input values.
- Req-RM-11: If redundancy management is in duplex mode and the servicing sensors report values that are in disagreement, then the status of the voter will indicate that it is in a degraded mode of operation.

D.4 Filter

The filter takes input from the redundancy management system and ensures that the temperature output falls within the expected range of sensor values and that the status of the voter correctly indicates trustworthy readings. The filter outputs include the resulting temperature, the voter status, and a filter error flag. In certain failure scenarios, the voter may agree on sensor readings that are out of range. The filter ensures that these values are not reported out of the redundancy management system framework.

There are no environmental assumptions made in the filter component.

The Filter (F) component adheres to the following requirements (Req):

- Req-F-1: The filter will ensure that the reported temperature is either zero or greater than 218 Kelvin.
- Req-F-2: The filter will ensure that the reported temperature is always less than 399 Kelvin.
- Req-F-3: If the received voter status is not faulty and the reported temperature is out of acceptable range (see Req-F-1–2), then the filter will report a faulty status and the temperature output will be zero.
- Req-F-4: If the received voter status is not faulty and the reported temperature is out of acceptable range (see Req-F-1–2), then the filter will report an error.

References

[1] Abdulaziz Alanazi and Jeremy Straub. Statistical analysis of cubesat mission failure. *32nd Annual AIAA/USU Conference on Small Satellites*, 2018.

[2] A. Avizienis, J. C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *Dependable and Secure Computing, IEEE Transactions on*, 1(1):11–33, Jan 2004.

[3] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: a versatile and industrial-strength SMT solver. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 415–442. Springer, 2022.

[4] Clark Barrett, Christopher L Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *International Conference on Computer Aided Verification*, pages 171–177. Springer, 2011.

[5] Jason Belt, John Hatcliff, Robby, John Shackleton, Jim Carciofini, Todd Carpenter, Eric Mercer, Isaac Amundson, Junaid Babar, Darren Cofer, David Hardin, Karl Hoech, Konrad Slind, Ihor Kuz, and Kent Mcleod. Model-driven development for the seL4 microkernel using the HAMR framework. *Journal of Systems Architecture*, 2022.

[6] Benjamin Bittner, Marco Bozzano, Roberto Cavada, Alessandro Cimatti, Marco Gario, Alberto Griggio, Cristian Mattarei, Andrea Micheli, and Gianni Zampedri. The xSAP safety analysis platform. In *TACAS*, 2016.

[7] Mishap Investigation Board. Mars climate orbiter mishap investigation board phase I report, November 10, 1999. https://llis.nasa.gov/llis_lib/pdf/1009464main1_0641-mr.pdf, 1999.

[8] Alex Boydston, Peter Feiler, Steve Vestal, and Bruce Lewis. Architecture centric virtual integration process (acvip): A key component of the dod digital engineering strategy. https://www.adventiumlabs.com/sites/adventiumlabs.com/files/publication/ACVIP_A_Key_Component_of_the_OSD_Digital_Engineering_Strategy_PA04421b.pdf, October 2019.

[9] Marco Bozzano, Alessandro Cimatti, Alberto Griggio, and Cristian Mattarei. Efficient anytime techniques for model-based safety analysis. In *Computer Aided Verification*, 2015.

[10] Darren D. Cofer, Andrew Gacek, Steven P. Miller, Michael W. Whalen, Brian LaValley, and Lui Sha. Compositional verification of architectural models. In Alwyn E. Goodloe and Suzette Person, editors, *Proceedings of the 4th NASA Formal Methods Symposium (NFM 2012)*, volume 7226, pages 126–140, Berlin, Heidelberg, April 2012. Springer-Verlag.

[11] Sylvain Conchon, Albin Coquereau, Mohamed Iguernlala, and Alain Mebsout. Alt-ergo 2.2. In *SMT Workshop: International Workshop on Satisfiability Modulo Theories*, 2018.

[12] Julien Delange and Peter Feiler. Architecture fault modeling with the aadl error-model annex. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 361–368. IEEE, 2014.

[13] Maeve Doyle, Rachel Dunwoody, Gabriel Finneran, David Murphy, Jack Reilly, Joseph Thompson, Sarah Walsh, Jessica Erkal, Gianluca Fontanesi, Joseph Mangan, et al. Mission testing for improved reliability of cubesats. In *International Conference on Space Optics—ICSO 2020*, volume 11852, pages 2699–2718. SPIE, 2021.

[14] Kevin Driscoll, Brendan Hall, Håkan Sivencrona, and Phil Zumsteg. Byzantine fault tolerance, from theory to reality. In *International Conference on Computer Safety, Reliability, and Security*, pages 235–248. Springer, 2003.

[15] Peter H Feiler and David P Gluch. *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison-Wesley, 2013.

[16] Kim Fowler. Mission-critical and safety-critical development. *IEEE Instrumentation & Measurement Magazine*, 7(4):52–59, 2004.

[17] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The jk ind model checker. In *Computer Aided Verification: 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part II 30*, pages 20–27. Springer, 2018.

[18] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[19] John Hatcliff, Jason Belt, Robby, and Todd Carpenter. HAMR: an AADL multi-platform code generation toolset. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation - 10th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2021, Rhodes, Greece, October 17-29, 2021, Proceedings*, volume 13036 of *Lecture Notes in Computer Science*, pages 274–295. Springer, 2021.

[20] John Hatcliff et al. Slang: The sireum programming language. In *International Symposium on Leveraging Applications of Formal Methods*, pages 253–273. Springer, 2021.

[21] John Hatcliff, Jerome Hugues, Danielle Stewart, and Lutz Wrage. Formalization of the AADL run-time services. In *Leveraging Applications of Formal Methods,*

Verification and Validation - 11th International Symposium on Leveraging Applications of Formal Methods, ISoLA 2022, Rhodes, Greece, 2022.

[22] John Hatcliff, Danielle Stewart, Jason Belt, Robby, and August Schwerdfeger. An AADL contract language supporting integrated model- and code-level verification. In *Supporting a Rigorous Approach to Software Development - 7th High Integrity Language Technology Workshop, HILT 2022*, 2022.

[23] Philipp Helle. Automatic SysML based safety analysis. In *Proceedings of the 5th International Workshop on Model Based Architecting and Construction of Embedded Systems*, pages 19–24, 2012.

[24] Evelyn Honoré-Livermore and Cecilia Haskins. Model-based systems engineering for cubesat fmeca. In *Recent Trends and Advances in Model Based Systems Engineering*, pages 529–540. Springer, 2022.

[25] Anjali Joshi and Mats P.E. Heimdahl. Model-based safety analysis of Simulink models using SCADE design verifier. In *SAFECOMP*, volume 3688 of *LNCS*, page 122, 2005.

[26] Anjali Joshi, Steven P. Miller, Michael Whalen, and Mats P.E. Heimdahl. A proposal for model-based safety analysis. In *Proceedings of 24th Digital Avionics Systems Conference*, 2005.

[27] SAnToS Laboratory. Sireum logika. https://logika.v3.sireum.org/index.html, 2022.

[28] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. Compcert-a formally verified optimizing compiler. In *ERTS 2016: Embedded Real Time Software and Systems, 8th European Congress*, 2016.

[29] O. Lisagor, T. Kelly, and R. Niu. Model-based safety assessment: review of the discipline and its challenges. In *The Proceedings of 2011 9th International Conference on Reliability, Maintainability and Safety*, 2011.

[30] Faida Mhenni, Nga Nguyen, and Jean-Yves Choley. Automatic fault tree generation from SysML system models. In *2014 IEEE/ASME International Conference on Advanced Intelligent Mechatronics*, pages 715–720. IEEE, 2014.

[31] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[32] NICTA and General Dynamics. sel4 microkernel, 2015. sel4.systems/.

[33] Society of Automotive Engineers. Architecture analysis & design language (aadl). Aerospace Standard AS5506C, 2022.

[34] Society of Automotive Engineers. Architecture Analysis & Design Language (AADL) Error Modeling version 2 (EMv2) Annex. Aerospace Standard AS5506/1, 2022.

[35] Danielle Stewart, Jing (Janet) Liu, Darren Cofer, Mats Heimdahl, Michael W. Whalen, and Michael Peterson. AADL-based safety analysis using formal methods applied to aircraft digital systems. *Reliability Engineering & System Safety*, 213:107649, 2021.

[36] Danielle Stewart, Jing (Janet) Liu, Michael Whalen, Darren Cofer, and Michael Peterson. Safety annex for architecture analysis design and analysis language. In *ERTS 2020: 10th European Conference Embedded Real Time Systems*, 2020.

[37] Danielle Stewart, Michael Whalen, Mats Heimdahl, Jing Janet Liu, and Darren Cofer. Composition of fault forests. In *International Conference on Computer Safety, Reliability, and Security*, pages 258–275. Springer, 2021.

[38] Donalt T. Ward and Steven B. Helton. Estimating Return on Investment for SAVI (a Model-Based Virtual Integration Process. In *SAE International Nournal of Aerospace*, November 2011.

[39] Adam West. Nasa study on flight software complexity. URL, March 2009. https://www.nasa.gov/pdf/418878main_FSWC_Final_Report.pdf.