Software Engineering Institute
Carnegie Mellon University

Improving Quality Using Architecture Fault Analysis with Confidence Arguments

Peter H. Feiler
Charles B. Weinstock
John B. Goodenough
Julien Delange
Ari Z. Klein
Neil Ernst

March 2015

TECHNICAL REPORT
CMU/SEI-2015-TR-006

Software Solutions Division

http://www.sei.cmu.edu

Table of Contents

Abstract **vii**

1 Introduction **1**

2 Approach, Concepts, and Notations **5**
- 2.1 Requirement Specification and Architecture Design 5
- 2.2 AADL Concepts Supporting Architecture Design 8
- 2.3 Architecture Fault Modeling and Analysis with EMV2 8
- 2.4 Confidence Map Concepts and Notation 11

3 Overview of the Stepper-Motor System **14**

4 An AADL Model of the SMS Architecture **16**
- 4.1 The Operational Environment and Interface Specification of the SMS 16
- 4.2 The SMS Architecture 19
  - 4.2.1 The Position Control System Specification 20
  - 4.2.2 The Actuator Specification 21
  - 4.2.3 The Stepper-Motor Specification 22
- 4.3 SMS Functional Design as State and Behavior 23
  - 4.3.1 The Position Control System State and Behavior 23
  - 4.3.2 The Actuator State and Behavior 25
  - 4.3.3 The Stepper-Motor State and Behavior 26
  - 4.3.4 SMS Design Constraints 26
- 4.4 Three Proposed Corrections 26
  - 4.4.1 The Fixed Command Send Time Solution 26
  - 4.4.2 The Buffered Position-Change Command Solution 27
  - 4.4.3 The Desired Position Actuator Commanding Solution 27

5 Fault Analysis of the SMS Architecture **29**
- 5.1 Hazards and Their Impact on the SMS Architecture 29
  - 5.1.1 The Position Control System SM_PCS Architecture Fault Model 30
  - 5.1.2 The Actuator SM_ACT Architecture Fault Model 31
  - 5.1.3 The Stepper-Motor SM_Motor Architecture Fault Model 33
  - 5.1.4 The Engine Control System ECS Architecture Fault Model 34
  - 5.1.5 The ECU Architecture Fault Model 35
  - 5.1.6 The Direct Access Memory Architecture Fault Model 35
  - 5.1.7 The Power Supply Architecture Fault Model 35
  - 5.1.8 The Fuel-Valve Architecture Fault Model 35
  - 5.1.9 The SMS Architecture Fault Model 36
- 5.2 Fault Impact Analysis of the Original SMS Architecture 37
- 5.3 Quantified Time-Sensitive Derived Safety Requirements 39
  - 5.3.1 Derived Requirement for Early Command Delivery 39
  - 5.3.2 Derived Requirement for Rate Mismatch 40
  - 5.3.3 Derived Requirement for Immediate Command Response 41
- 5.4 Analysis of the Fixed Command Send Time Solution 42
- 5.5 Analysis of the Buffered Position-Change Command Solution 42
- 5.6 Analysis of the Position-Commanded Actuator Design Alternative 42
- 5.7 Comparison of the SMS Designs 43

6 Establishing Confidence in the SMS **45**
- 6.1 Confidence Maps for SMS 45

6.1.1 The Stepper Motor 45
6.1.2 The Stepper-Motor Position Control System (SM_PCS) 46
6.1.3 Eliminating Defeater R2.3 48
6.1.4 Using the Confidence Map to Allocate Assurance Resources 52
6.2 Confidence in the Position-Commanded (“Alternate”) Design for the Stepper Motor 53
6.2.1 Top Level 53
6.2.2 Using the Confidence Map to Allocate Assurance Resources 54
6.3 Discussion 55
6.3.1 The Effort Needed to Obtain Needed Evidence 56

7 Conclusions 59

Appendix A The SMS Data Model 61

Appendix B Variation in Inter-Arrival Time 63

Appendix C Confidence Maps 64

C.1 Other Defeaters from the Original Design 64
C.1.1 Stepper-Motor Malfunction Defeater (Defeater R2.1) 64
C.1.2 Stepper-Motor Speed Defeater (R2.2) 65
C.1.3 The SM_PCS Does Not Issue Appropriate Commands (R2.4) 66

C.2 Other Defeaters from the Position-Commanded Design 70
C.2.1 Stepper-Motor Malfunction Defeater (Defeater R2.1) 70
C.2.2 Stepper-Motor Speed Defeater (R2.2) 71
C.2.3 The SM_PCS Does Not Issue Appropriate Commands (R2.3) 72

References 75

List of Figures

Figure 1: Architecture-Centric Requirements Decomposition 5
Figure 2: A System and Its Interface with Its Environment 6
Figure 3: AADL Graphical Symbols 8
Figure 4: Textual AADL Example 8
Figure 5: Textual AADL Error Model Example 9
Figure 6: Potential Hazard Sources in the Feedback Control Loop [Leveson 2012] 11
Figure 7: Confidence Map Example 13
Figure 8: SMS in Its Operational Environment 17
Figure 9: Textual SMS Interface Specification 17
Figure 10: Details of Logical SM_PCS Architecture 19
Figure 11: Textual SM_PCS Interface Specification 20
Figure 12: Actuator SM_ACT Interface Specification 21
Figure 13: Position Control System Behavior Specification 25
Figure 14: Stepper-Motor Actuator Behavior Specification 26
Figure 15: Position-Commanded Actuator Behavior 28
Figure 16: Fault Model Specification of the Position Control System SM_PCS 31
Figure 17: Fault Model Specification of the Actuator SM_ACT 33
Figure 18: Fault Model Specification of the Stepper Motor SM_Motor 34
Figure 19: Fault Model Specification of ECS 34
Figure 20: Fault Model Specification of the Direct Access Memory 35
Figure 21: Fault Model Specification of Fuel Valve 36
Figure 22: Abstracted Fault Model Specification for SMS 37
Figure 23: SMS Variant Specific Fault Information 37
Figure 24: Missing Error Containment Guarantee Specification 38
Figure 25: Fault Impact Report for SMS 38
Figure 26: End-to-End Flow Specification for SMS Commands 41
Figure 27: End-to-End Latency for the Original Design 42
Figure 28: The Top-Level Confidence Map for the Stepper Motor 46
Figure 29: Subdefeaters for Determining SM_PCS Position 48
Figure 30: Eliminating a Defeater Regarding SM Position at Startup 49
Figure 31: Eliminating a Defeater Regarding the Position of the SM at the Start of a Frame 50
Figure 32: Eliminating a Defeater Regarding the Overwriting of the Previous Command 51
Figure 33: Top-Level Claim of the Alternative Design 54

Figure 34: The Two Confidence Maps 55
Figure 35: S Steps Cannot Be Completed in Time F 56
Figure 36: Evidence Required for the Two Designs 57
Figure 37: SM_PCS Data Model in AADL 62
Figure 38: Eliminating a Defeater Regarding SM Malfunction 64
Figure 39: Eliminating a Defeater Regarding SM Speed 66
Figure 40: Eliminating a Defeater Regarding the SM_PCS Issuing Appropriate Commands to the SM 67
Figure 41: The Claim that the SM_PCS Calculates the Command Correctly 68
Figure 42: Eliminating a Defeater Regarding Data Corruption 69
Figure 43: Eliminating a Defeater Regarding SM Malfunction 70
Figure 44: Eliminating a Defeater Regarding Speed of the SM 71
Figure 45: Eliminating a Defeater Regarding the SM_PCS Issuing Appropriate Commands 72
Figure 46: Eliminating a Defeater Regarding the Position Calculation 73
Figure 47: Eliminating a Defeater Regarding Data Corruption 74

List of Tables

Table 1:	Defeater Types	12
Table 2:	Comparison of Architecture Design Alternatives	43

Abstract

This case study shows how an analytical architecture fault-modeling approach can be combined with confidence arguments to diagnose a time-sensitive design error in a control system and to provide evidence that proposed changes to the system address the problem. The analytical approach, based on the SAE Architecture Analysis and Design Language for its well-defined timing and fault behavior semantics, demonstrates that such hard-to-test errors can be discovered and corrected early in the lifecycle, thereby reducing rework cost. The case study shows that by combining the analytical approach with confidence maps, we can present a structured argument that system requirements have been met and problems in the design have been addressed adequately—increasing our confidence in the system quality. The case study analyzes an aircraft engine control system that manages fuel flow with a stepper motor. The original design was developed and verified in a commercial model-based development environment without discovering the potential for missed step commanding. During system tests, actual fuel flow did not correspond to the desired fuel flow under certain circumstances. The problem was traced to missed execution of commanded steps due to variation in execution time.

1 Introduction

The purpose of this case study is to show how architecture fault modeling and analysis can be used to diagnose a time-sensitive design error encountered in a control system and to investigate whether proposed changes to the system address the problem. The analytical approach demonstrates that such errors, which are notoriously hard to test for, can be discovered and corrected early in the lifecycle, thereby reducing rework cost. The case study shows that by combining the analytical approach with confidence maps, we can present a structured argument that system requirements have been met and problems in the design have been addressed adequately—increasing our confidence in the system quality.

In this case study, we investigated a stepper-motor system (SMS) that is part of an aircraft engine control system that manages fuel flow by adjusting a fuel valve. The baseline design for controlling the valve is from an actual system, but we have omitted or changed inessential application- and manufacturer-specific details. The original design was developed and verified in a model-based development environment called SCADE,[1] and an implementation was tested on actual equipment.

In some situations, actual fuel flow did not correspond to the desired fuel flow. The problem was traced to the fact that the stepper motor sometimes did not execute enough steps to achieve the commanded fuel-valve position. The failure was suspected to be due to execution time jitter in the stepper-motor control system, which resulted in commands being sent to the actuator at variable time intervals. The failure was not immediately detectable as there is no direct feedback from the stepper motor, and the feedback loop of the engine control system does not have sufficient fidelity to detect single missed steps during executions. For this reason, we focus on the SMS portion of the engine control system, that is, the stepper motor, its actuator, and the position control system for the stepper motor.

Two repairs were proposed to correct the problem, but there was limited evidence before testing that either proposed solution would address the problem of missed steps. Our analytical approach provides predictive evidence, completed with prototype testing evidence, that the problem is inherent in the architecture design of the SMS. The findings regarding the stepper-motor position control system are applicable not only to the engine control system but also to any system that uses a stepper motor in an open-loop setting, such as those that control other types of valves, flaps, rudders, and other mechanical parts.

SCADE is a notation for modeling system behavior using interacting and nested state machines. While the SCADE tool set is able to verify behavioral aspects of the system specification, it does not address the time-sensitive nature of this control system implementation. We will demonstrate that the SAE Architecture Analysis & Design Language (AADL) [SAE 2012] and its Error Model Annex are effective tools for

- identifying causes and effects of failures (particularly those of a time-sensitive nature
- verifying that proposed system changes address the problem

[1] See http://www.esterel-technologies.com/products/scade-suite/ for more information about SCADE.

SAE AADL is an architecture modeling language specifically designed for software-dependent systems with well-defined execution and interaction semantics. The Error Model Annex standard is designed to support fault analysis of AADL models.

We will also demonstrate that a particular architecture decision—namely, the translation of the desired target position of the stepper motor into a sequence of position-change command by the stepper-motor controller—becomes a source of avoidable faults. We compare the original architecture to an alternative architecture design with reduced complexity and present arguments for increased confidence in the analytical evidence over existing practice and reduced need for evidence for the alternative design.

The approach used in this case study draws on a framework for software assurance and software quality improvement [Feiler 2012] in support of software-dependent, safety-critical system qualification and certification. *Software assurance* is defined as "the level of confidence that software is free from vulnerabilities, either intentionally designed into the software or accidentally inserted at any time during its life cycle, and that the software functions in the intended manner" [CNSS 2010]. In other words, it assures that a system meets the mission requirements specified for it and that hazards resulting in non-nominal behavior have been addressed through elimination or by dependability requirements on a fault management architecture.

This definition connects to several key elements of the framework for software assurance and software quality improvement:

- *Analysis of mission and dependability requirements associated with the system architecture*: Studies of software-dependent, safety-critical systems have shown that 80% of all errors are not discovered until system integration, and 70% or more of those errors are related to requirements and architecture design. Complexity in safety-critical software due to limited architecture abstractions is a major contributor to high certification-related rework cost [Dvorak 2009]. Requirements errors fall into the following categories: 33% omission, 24% incorrect, 21% incomplete, 6.3% ambiguous, 6.1% over specified, 4.7% inconsistent [Hayes 2003]. At the same time, T text, diagram, and table-based requirements documentation and the use of Microsoft Word and Dynamic Object-Oriented Requirements System (DOORS) dominate the practice [FAA 2009a].

  We draw on a set of requirements engineering practices in the context of an architecture specification expressed in AADL [SAE 2012], as outlined in the FAA Requirement Engineering Management Handbook [FAA 2009], demonstrated in goal-oriented requirements engineering (KAOS) [Lamsweerde 2003], and reflected in the draft Requirements Definition and Analysis Language (RDAL) Annex standard [Blouin 2011]. We also draw on guidance from the Virtual Upgrade Validation (VUV) method developed by the Software Engineering Institute (SEI) for modeling and predictive analysis of the impact of system changes on key quality attributes through the use of AADL [DeNiz 2012].

- *Assessment and management of hazards on the safety, reliability, and security of the system*: Traditional statistical reliability engineering methods do not carry over well to software. Eliminating these software design defects and mitigating residual software defects as hazards has to be the focus [Feiler 2012a]. Safety analysis requires a systemic approach to identifying contributing hazards rather than a single failure as the root cause of an accident [Leveson 2012].

In this case study, we use version 2 (EMV2) of the Error Model Annex [SAE 2006],[2] which supports established safety assessment practices such as SAE ARP 4761 [Delange 2014a]. We draw on two hazard impact analysis methods—Fault Propagation and Transformation Calculus (FPTC) [Paige 2009] and System Theoretic Process Analysis (STPA) [Leveson 2012]—to systematically identify scenarios leading to contributing hazards from a safety perspective so they can be eliminated or managed.

- *Evidence in the form of architecture design analysis of operational qualities to complement testing*: AADL is a key technology in System Architecture Virtual Integration (SAVI), an aerospace industry initiative to achieve early discovery of system-level problems through analysis of integrated architecture models [Redman 2010]. Such architectural analysis can identify design errors before a system is implemented, thereby reducing high-cost rework needed when such errors are not discovered until system integration. The feasibility of this approach was demonstrated on a multi-tier aircraft model [Feiler 2010]. Other applications of this approach include the analysis of a reference architecture for autonomous space platforms and its instantiation for specific systems [Feiler 2009a] and the predictable integration of medical devices [Larson 2013].

  We use existing analysis supported by AADL, such as end-to-end flow latency analysis, and augment them with formalized specification of the particular timing-related problem.

- *Structured argumentation to assure confidence in the presented evidence*: Structured argumentation methods (such as assurance cases [Kelly 1998, GSN 2011, ISO 2011]) explain, in a reviewable form, how various items of evidence combine to support claims about system properties. In the form of structured argument used in this report (a *confidence map*), reasons for doubting the truth of claims, the validity of evidence, and the soundness of inferences used in the argument are represented explicitly. As reasons for doubt, called *defeaters*, are removed, confidence in system claims increases [Goodenough 2013, Weinstock 2013].

  We use the confidence map in developing an argument and in evaluating an argument's strengths and weaknesses.

Section 2 discusses the approach taken in this case study based on the above-mentioned framework. It introduces the AADL architecture modeling, the EMV2 fault modeling, and the confidence map notations. Section 3 is a brief overview of the SMS architecture and its essential functional requirements. Section 4 presents the SMS in its operational environment, the architecture of SMS, and the detailed design specification of each SMS subsystem as an AADL model. In this section derived requirements on these subsystems are clearly identified and traced to the SMS system requirements. The original architecture is discussed, as are the two corrective solutions and the alternative design. In Section 5 we elaborate the architecture model into an architecture fault model identifying hazards and analyze their system impact. The section also discusses quantified time-sensitive derived (safety) requirements that must be satisfied to eliminate the hazards—as well as analysis and prototype implementation results to diagnose the specific missed-step problem encountered in the SMS. Section 6 presents a confidence map for the original SMS architecture as well as for the design alternative. These maps show the evidence and inferences

[2] The SAE Error Model Annex EMV2 standard revision, led by one of the authors, will be published in spring 2015.

needed to eliminate doubts about the adequacy of the SMS design. Finally, Section 7 provides conclusions. Three appendices provides additional supporting material.

2 Approach, Concepts, and Notations

In this section we present a short introduction to the approach we are taking for this case study. The approach combines practices for requirements engineering, architecture modeling and analysis, hazard analysis from a safety-critical system perspective, and assurance cases for structured confidence arguments. Our approach is model based to support tool-based analysis and uses concepts and notations of SAE AADL, the Error Model Annex EMV2, and confidence maps.

2.1 Requirement Specification and Architecture Design

We use AADL to model the system and its operational environment to clearly identify the system boundary, environmental assumptions, and system requirements. Some requirements for the system may need to be refined into verifiable requirements. In the process of developing the next layer of architecture we decompose system requirements and allocate them to subsystems, illustrated in Figure 1. In some cases, a system-level requirement is met when all the sub-requirements are met. In other cases it is necessary to provide additional evidence at the system level. A collection of evidence in the form of review, analysis, and test results provides confidence that the requirement is met. Subsets of verification results may be considered sufficient evidence.

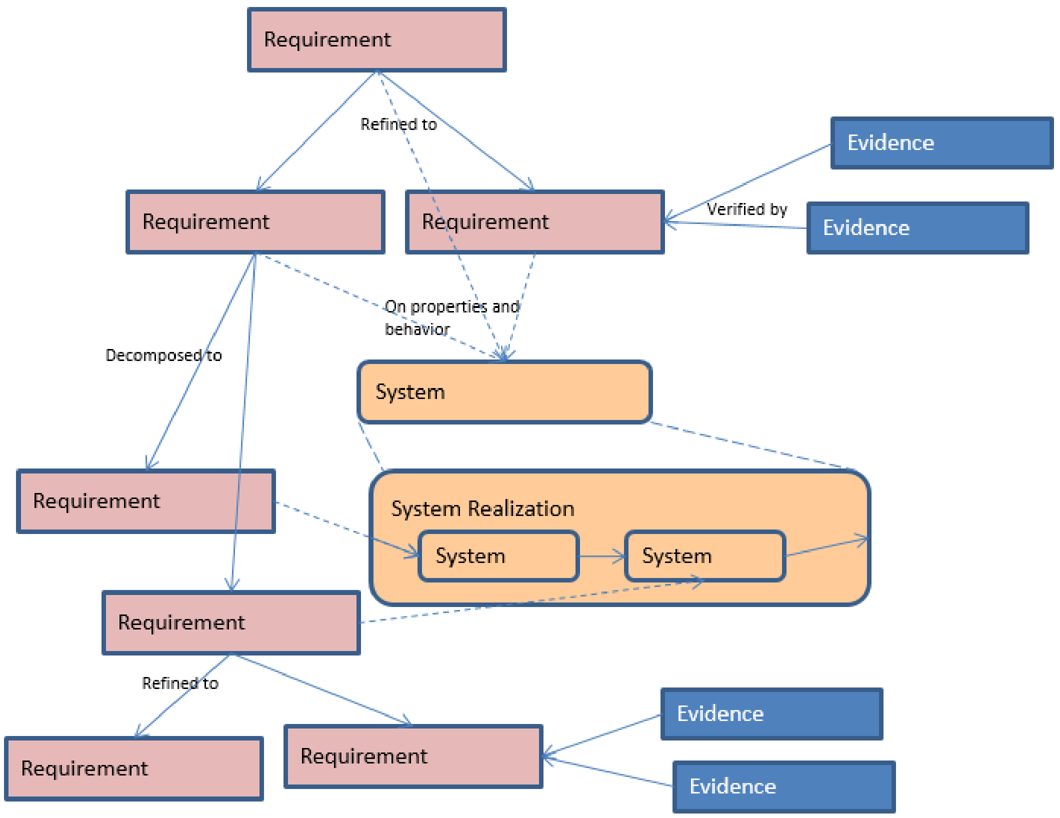


Figure 1: Architecture-Centric Requirements Decomposition

An operational system, visualized in Figure 2, is defined as a set of behaviors by execution of functions to transform input into output using state, respecting constraints/controls, and requiring resources to meet a defined mission in a given environment [AFIS 2010]. The requirements specification of a system consists of assumptions about the input, the incoming controls, the availability of resources, guarantees made by the system on the produced output, requirements on the functions executed in different states, and constraints on the state. Requirements are associated with a system in four forms:

- *requirements* on the state and behavior of the system
- *assumptions* about processing input, control input, and utilized resources
- *guarantees* about output, control feedback, and resource usage
- *constraints* on the implementation of the system (i.e., on the interaction between its parts)

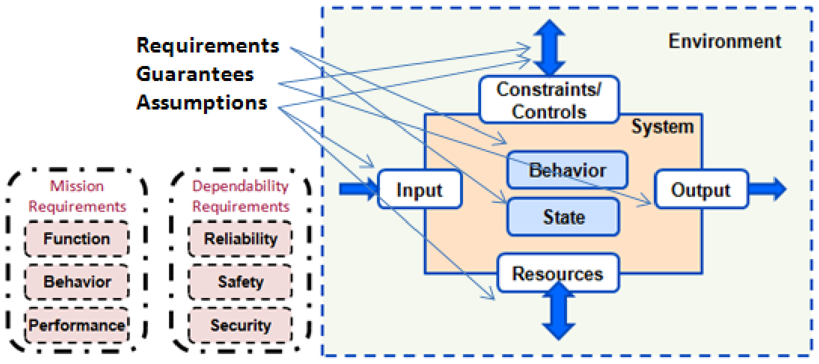


Figure 2: A System and Its Interface with Its Environment

Requirements fall into two major categories: mission requirements and dependability requirements [Feiler 2012a].

- *Mission requirements* focus on nominal system operation (i.e., functional, behavioral, and performance requirements).
- *Dependability requirements* focus on non-nominal system operation due to failures and other hazardous conditions (i.e., safety, reliability, and security requirements).

Requirement specification and architecture design of safety-critical systems is an iterative process in two ways. First, the architecture design is iteratively decomposed into subsystems, as mentioned earlier. Second, a nominal set of mission requirements and architecture design are examined by hazard and fault impact analysis for non-nominal conditions that result in safety and reliability requirements and fault management functionality [McDermid 2007]. The fault management architecture introduces its own non-nominal conditions that must be addressed iteratively.

In each phase of this approach we argue that the system requirement specification and design have sufficient quality. We provide rationale for

- sufficient coverage of operational use cases by system requirements and sufficient evidence of requirement specification completeness and consistency
- sufficient system requirement coverage by subsystem requirements
- sufficient hazard coverage in terms of a fault ontology and their mitigation through safety, reliability, and security requirements on the fault management architecture
- sufficient evidence of assumption/guarantee contract satisfaction by the system design

- sufficient evidence of requirement and constraint satisfaction by the next-level design and implementation

This approach allows for compositional verification evidence through a combination of type and consistency checking of AADL models and Error Model specifications to meet standard consistency rules, such as assumptions being met by guarantees, and verification of system specific requirements and constraints.

To diagnose the problem of missed steps due to the time-sensitive nature of the SMS, we use fault impact analysis to determine all possible contributors to the step loss and identify those that are due to SMS design decisions that can be corrected. We then make the proposed corrections and proposed alternative design changes to the architecture model and update any affected requirements and hazard analysis to assess whether they have addressed the problem.

We take advantage of the well-defined execution and interaction semantics in AADL to abstractly, but precisely, specify the runtime behavior of the SMS to diagnose the time-sensitive failure behavior and evaluate how well proposed solutions address the problem.

The VUV method [DeNiz 2012] provides guidance on how to model a system in AADL by mapping particular application system categories (e.g., a control system), into application patterns (e.g., a feedback loop), representing those patterns in AADL, and identifying relevant operational quality attributes, and expressing them as AADL properties on software and hardware components, connections and deployment bindings, and end-to-end flows. VUV also provides guidance on how to focus modeling on those aspects of the system that are relevant to the proposed system change and its effect on quality attributes of interest.

Some requirements are directly reflected in the model, while other requirements represent constraints or invariants on system properties or state. We use the RDAL notation to provide a requirement identifier, description, and rationale, as well as traceability to stakeholders, and requirement refinement and decomposition for each requirement. In addition, the RDAL requirement declaration identifies the system or system element to which the requirement applies, as well as the property in the model that reflects the requirement. For example, guarantees and assumptions on exchanged data are specified as properties on data types—and requirements on the timing of the interaction are specified as properties on periodic and aperiodic threads with sampling and queuing ports. State behavior is modeled by modes, persistent data components, and detailed behavior specifications. Invariants and constraints are expressed in the RDAL requirement declaration using a constraint notation (Lute[3]), which is then used to verify the design. Note that RDAL supports the association of verification activities with requirement declarations to represent an assurance plan.

We model non-nominal conditions by annotating the architecture model with fault behavior using EMV2. Impact analysis of this architecture fault model leads to a set of assumptions that reflect the absence of hazards and a set of derived requirement declarations to reflect mitigation of the presence of hazards.

[3] Lute and its extension Resolute are constraint specification notations developed by Rockwell Collins for supporting contract based assurance cases on AADL models.

2.2 AADL Concepts Supporting Architecture Design

The AADL standard defines a number of component categories with specific semantics: *system*, *device*, *bus*, *processor*, *memory*, *virtual bus*, *virtual processor*, *process*, *thread*, *thread group*, *data*, *subprogram*, and *subprogram group*. AADL also defines a number of component *features*, that is, interaction points with other components. These are points for any kind of interaction: *bus access* for physical connections between hardware components; ports in the form of a sampling *data port*, a queuing *event port*, and an *event data port* for queued message communication; *data access* for references to shared (global) variables; and *feature group* for representing collections of interaction points. Data and event data ports include a specification of the type of data that are communicated. The interactions between the interaction points are represented by connections.

An AADL thread has properties that indicate whether it has a *periodic* dispatch based on the clock or an *aperiodic* dispatch based on the arrival of events or messages, the *period* at which it executes, the *deadline* for completion of one execution dispatch, and the best- and worst-case *execution time*. The execution and communication timing semantics are key to identifying the potential for timing errors in the SMS design.

The specification of an AADL component may also include a *mode* state machine to represent operational modes and *flow specifications* to model *end-to-end flows* across a system. These flow specifications are used in latency analysis to determine the responsiveness of the SMS to commands. Since flow specification is part of the AADL, flow representations are integrated with the architecture representation and do not require separate models or modeling languages. For complete coverage of AADL, see the AADL book by Feiler and Gluch [Feiler 2012].

AADL has a graphical and textual notation. The graphical symbols for the AADL components and features used in the SMS model are shown in Figure 3. Their use in specifying the SMS and its operational environment can be seen in Figure 8. Connections between the features of different components are shown as lines. The textual representation for the declaration of an AADL device called *Sensor* is shown in Figure 4. It specifies that the sensor has a single outgoing data port of data type *Position* providing a sampled sensor reading.

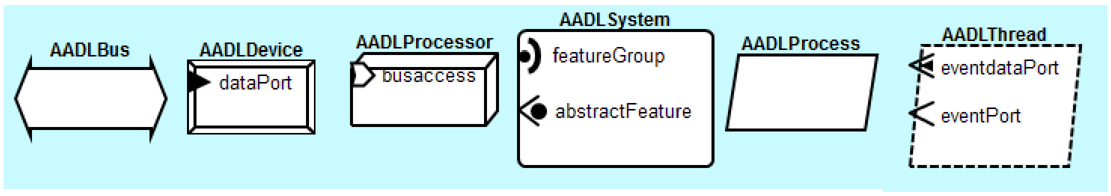


Figure 3: AADL Graphical Symbols

```
device Sensor
features
  SampledPosition: out data port Position;
end Sensor;
```

Figure 4: Textual AADL Example

2.3 Architecture Fault Modeling and Analysis with EMV2

We use EMV2 to represent and analyze the architecture fault model. EMV2 supports automated Failure Modes and Effects Analysis (FMEA) and reliability predictions [Hecht 2011] and the fault propagation and transformation calculus (FPTC) for automated safety analysis [Paige 2009]. EMV2 also includes a fault propagation ontology that draws on research in failure mode assumptions and assumption coverage [Powell 1992], and on a fault model for redundant distributed systems [Walter 2003]. For further information on architecture fault modeling with EMV2 see Delange [Delange 2014, Delange 2014a].

EMV2 allows fault information to be attached to each component specification and implementation. EMV2 supports architecture fault modeling at three levels of abstraction:

- *error propagation*: focus on fault sources in a system and their impact on other components or the operational environment through propagation. It allows for safety analysis in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
- *component error behavior*: focus on a system or component fault model identifying faults in a system (component), their manifestation as failure, the effect of incoming propagations, and conditions for outgoing propagation. It allows for fault tree analysis of a system stochastic reliability and availability analysis of systems in terms of its components and their interactions.
- *composite error behavior*: focus on relating the fault model of system components to the abstracted fault model of the system. It allows for scalable compositional fault analysis.

In this case study we will primarily make use of the error propagation specification, in particular error source, incoming error propagation, and containment specifications reflecting assumptions, and outgoing error propagation and containment specifications reflecting guarantees.

EMV2 introduces the concept of error type to characterize faults, failures, and propagations. Sets of error types are organized into error type libraries and are used to annotate error events, error states, and error propagations.

For example, a valve can have faults of the types *Leakage*, *StuckOpen*, and *StuckClosed*. Error types can be grouped into type sets. For example the type set *ValveErrors* may be defined as *{Leakage*, *StuckOpen*, *StuckClosed}*. Error types can also be placed into a type hierarchy indicating that the subtypes cannot occur at the same time. For example, *EarlyDelivery* and *LateDelivery* are declared as subtypes of *TimingError*. By referring to the type set *ValveErrors* or the type *TimingError* we indicate that any of the element types can occur.

Figure 5 illustrates the specification of an outgoing error propagation of type *ValueError*. The propagation paths to other components are determined by the connections and software to hardware binding declarations in the AADL model.

```
device Sensor
features
  SensedPosition: out data port Position;
  annex EMV2 {**
  use types ErrorLibrary;
  error propagations
    SensedPosition : out propagation {ValueError};
  end propagations;
  **};
end Sensor;
```

Figure 5: Textual AADL Error Model Example

EMV2 includes a set of predefined error types as starting point for systematic identification of different types of fault propagations – providing an error propagation ontology based on Powell, Walter, and Paige [Powell 1992, Walter 2003, Paige 2009]. They fall into four categories: *service* related, *timing* related, *value* related, and *replication* related. They are expressed in terms of an individual *service item*, a *sequence* of service items, or the whole *service* provided by a component.

Service-related error types include *ServiceOmission* (representing failure to provide service items, such as in a power loss), *ServiceCommission* (representing service items when not expected, such as unexpected acceleration), *ItemOmission* (representing a missing item, such as a lost message), and *ItemCommission* (representing an extra item, such as a spurious message). Users can define aliases for error types (e.g., we will use the alias *NoPower* for *ServiceOmission* to characterize the propagation resulting from a failed power supply).

Value-related errors for individual items are *OutOfRange*, *OutOfBounds*, and *UndetectableValueError*. The set of these types is referred to as *ValueError*. Examples of sequence and service-related value errors are *StuckValue* and *OutOfCalibration*.

TimingError for an individual item can be *EarlyDelivery* and *LateDelivery*. *RateError* for a service item sequence are *LowRate*, *HighRate*, and *RateJitter*.

ReplicationError occurs in redundant systems and can be *AsymmetricTiming*, *AsymmetricOmission*, and *AsymmetricValue*. They allow for characterization of errors due to a Byzantine fault or errors occurring independently on replicated channels.

We make use of the ability to define aliases for these error types to give them more meaningful names in the context of a specific component. For example, we define the alias *MissingCommand* for *ItemOmission*.

We will use these error propagation types to specify the potential presence or absence of a hazard for SMS and for each of the SMS components by error propagation and containment declarations associated with each incoming and outgoing port and other interaction points to other components —as described in Section 5.1. The ontology and the explicit specification of error propagations and containments allow us to ensure that we have considered all possible hazards in the analysis.

Finally, we draw on elements of the STPA by Leveson [Leveson 2012] to systematically identify scenarios leading to contributing hazards from a safety perspective so they can be eliminated or managed. She uses a feedback loop, shown in Figure 6, as a primary pattern.

The hazards identified in the pattern are represented in EMV2 by error sources and incoming and outgoing propagations on ports. For example, in the context of the sensor *no information* maps into *service* or *item omission*, *incorrect information* maps into *value error*, and *feedback delay* maps into *late delivery*. The EMV2 ontology suggests a distinction of different types of value errors, as well as consideration for early delivery, sequence, and rate errors.

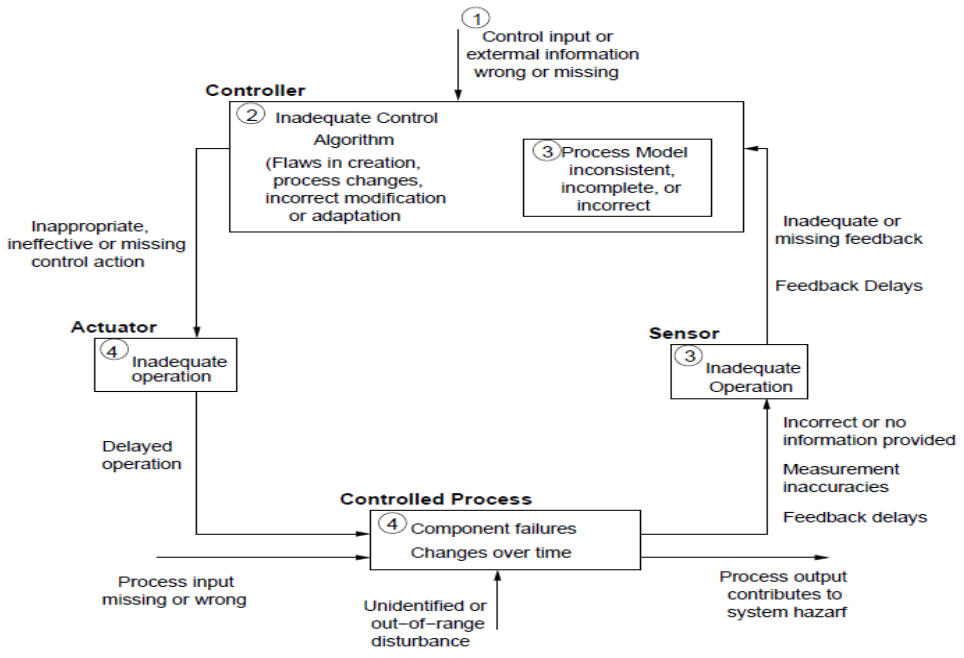

1
Control input or external information wrong or missing
Controller
2 Inadequate Control Algorithm
(Flaws in creation, process changes, incorrect modification or adaptation
3 Process Model inconsistent, incomplete, or incorrect
Inappropriate, ineffective or missing control action
Inadequate or missing feedback
Feedback Delays
Actuator
4 Inadequate operation
Sensor
3 Inadequate Operation
Delayed operation
Incorrect or no information provided
Measurement inaccuracies
Feedback delays
Controlled Process
4 Component failures
Changes over time
Process input missing or wrong
Unidentified or out–of–range disturbance
Process output contributes to system hazarf

Figure 6: Potential Hazard Sources in the Feedback Control Loop [Leveson 2012]

2.4 Confidence Map Concepts and Notation

A confidence map is a structure that explicitly shows the reasons for doubt relevant to a particular argument. A confidence map consists of claims, evidence (data), reasons for doubt (defeaters), and inference rules explaining

- why evidence or a valid claim serves to eliminate a defeater
- why the elimination of a doubt about the validity of a claim, evidence, or inference rule is regarded as justifying confidence in the validity of the claim, evidence, or rule

A confidence map is useful both in developing an argument and in evaluating an argument's strengths and weaknesses.

A confidence map consists of a connected set of inferences having the form

if P then Q unless R, S, T, …

The inference if P then Q is defeasible, meaning that the conclusion Q is subject to doubt based on additional information [MRL 2009]. In particular, in the above formulation, if any of R, S, or T are true, Q is either invalid or its validity is unknown. Because R, S, and T cast doubt on the validity of conclusion Q, they are called defeaters. The "…" is significant because in principle, additional defeaters can be identified at any time. In a confidence map, inference rules exist between claims, evidence, and defeaters.

A confidence map is developed by identifying defeaters and then arguing (via inference rules) that the defeaters are false and that the falsity of the defeaters implies the validity of some otherwise defeasible claim, evidence, or inference rule.

In the defeasible reasoning literature [Pollock 2008, Prakken 2010] there are only three types of defeaters: rebutting, undermining, and undercutting. A rebutting defeater provides a counter-example to a conclusion. An undermining defeater raises doubts about the validity of evidence. An undercutting defeater raises doubt about the sufficiency of an inference rule by specifying circumstances under which its conclusion is in doubt even when its premises are true.

Table 1: Defeater Types

Defeater	**Attacks**	**Form**	**Mitigation**
Rebutting	Claim	R, so claim Q is false	Look for counter-examples and why they can't occur
Undercutting	Inference rule	U, so conclusion Q can be true or false	Look for conditions under which the rule is insufficient and why those conditions don't hold or what additional information is needed to make the rule sufficient
Undermining	Evidence	M, so premise P is invalid	Look for reasons the premise might be invalid and show those conditions don't hold

As a simple example, we might argue "Tweety can fly because Tweety is a bird." A confidence map for this argument will make explicit possible reasons for doubting that Tweety can fly (see Figure 7). The confidence map starts by identifying the top-level claim, "Tweety can fly" and then identifies characteristics of counter-examples to the claim, namely, "Unless Tweety is heavier than air, not capable of producing lift, and not being propelled by an external agent." We seek evidence eliminating this doubt about the claim. The map shows that someone has examined Tweety and determined that Tweety is a bird. The inference rule (IR3.2), "If X is a bird, X is capable of producing lift" is used to show how the evidence eliminates the counter-example. We also look for possible ways the evidence could be invalid (undermining defeaters). The map mentions the possibility that the examiner is incompetent; perhaps Tweety is actually a bat, in which case, our inference rule does not apply and we have no basis for concluding whether Tweety can fly. We would need to eliminate this defeater to have complete confidence in the conclusion about Tweety's ability to fly. We also consider weaknesses in the inference rule, noting that if Tweety is a juvenile the rule is insufficient—Tweety might be able to fly and might not. Finally, there is an inference rule (IR2.2) that links the rebutting defeater (R2.1) to the claim that Tweety can fly.

As these defeaters are eliminated (i.e., shown to be false), our confidence in the conclusion increases. When all doubts have been eliminated, we say we have total confidence in the claim.

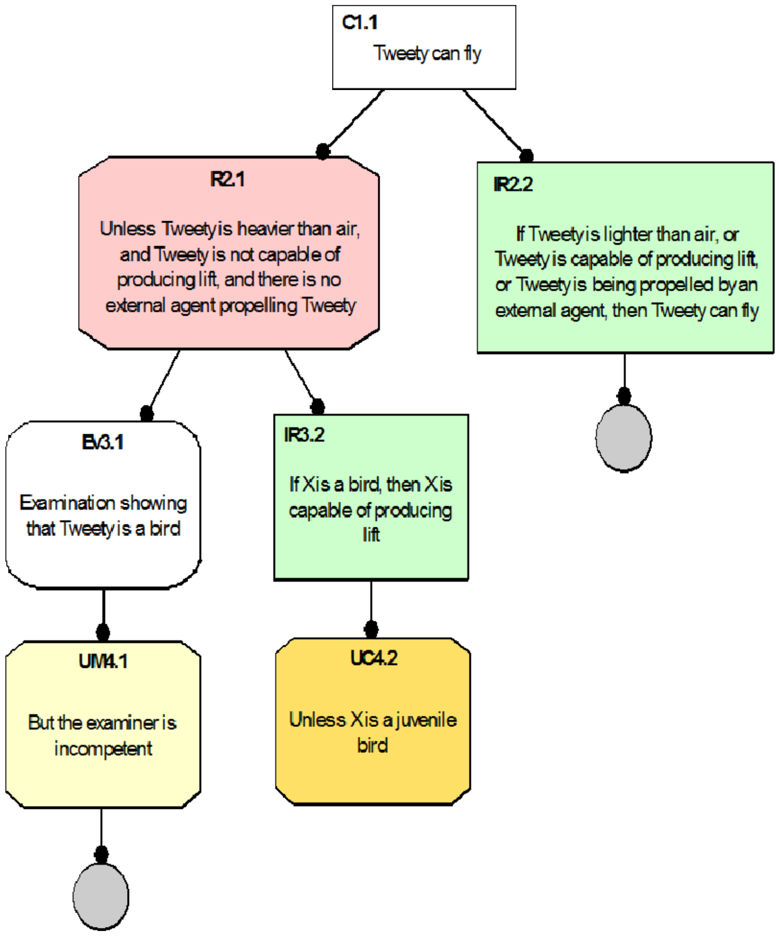

C1.1
Tweety can fly
R2.1
Unless Tweety is heavier than air, and Tweety is not capable of producing lift, and there is no external agent propelling Tweety
IR2.2
If Tweety is lighter than air, or Tweety is capable of producing lift, or Tweety is being propelled by an external agent, then Tweety can fly
EV3.1
Examination showing that Tweety is a bird
IR3.2
If X is a bird, then X is capable of producing lift
UM4.1
But the examiner is incompetent
UC4.2
Unless X is a juvenile bird

Figure 7: Confidence Map Example

We use a graphical notation for confidence maps. In this map, the claim "Tweety can fly" is shown in a clear rectangular box. Claims are always stated as predicates, that is, they are either true or false. Defeaters are shown in rectangles with chopped-off corners; the color of the rectangle indicates the type of defeater. Rebutting defeaters are shown in red. Evidence that Tweety is a bird serves to eliminate this rebutting defeater. The evidence is shown in a rounded clear rectangle. An undermining defeater casting doubt on the validity of the evidence is shown in a light yellow. Inference rules are shown in a green rectangle. Undercutting defeaters (casting doubt on the sufficiency of the rule) are colored yellow-orange. In general, when an inference rule is indefeasible (has no undercutting defeaters) we do not show it. IR2.2 is an example of an indefeasible rule.

The grey circle indicates there are no doubts about the element to which it is attached. For example, the use of this element indicates we have decided that IR2.2 has no undercutting defeaters and similarly, that UM4.1 is assumed to be false without having to present any further evidence or argument. Such decisions can, of course, be challenged by reviewers of a map.

3 Overview of the Stepper-Motor System

The subject of the case study is an engine control system (*ECS*) for an aircraft engine. It is a feedback control system that manages the thrust of an engine. One of its functions is to adjust the fuel flow to the engine through a fuel valve. A stepper motor is used to change the position of the fuel valve. The stepper-motor control system operates open loop (i.e., there is no direct feedback on the successful execution of a step by the motor). The enclosing ECS feedback control loop can detect a misalignment of the actual fuel-valve position with the expected position, but not at the granularity of individual steps. Since the functionality of the stepper-motor controller has been modeled and validated with SCADE without taking time into account, indications are that the problem of missed step execution is due to the time-sensitive nature of processing of the control system, specifically in the stepper-motor system (*SMS*), which consists of the stepper motor (*SM_Motor*), the actuator (*SM_ACT*), and the position control system (*SM_PCS*).

The SMS is commanded to open the fuel valve in terms of a percentage with zero being closed and 100 being completely open. The stepper motor takes a known number of steps to move the fuel valve from a completely closed to a completely open position. The SMS is expected to reach the commanded position within a bounded time that is proportional to the distance between the current position and the desired position. At command completion the stepper motor is expected to have reached the commanded position closest to the requested opening percentage.

The SMS may receive a new command from ECS before the previous command has been completed. SMS is expected to immediately respond to the new command (i.e., immediately moves the fuel valve to the most recent commanded position without first continuing to the previously commanded position).

SM_PCS operates periodically at a rate of 25ms, converts the percentage into the desired position in terms of stepper motor steps (*Steps*), and commands the actuator to move the stepper motor a specified number of steps in the *open* or *close* direction. The maximum number of steps (*MaxStepCount*) to be commanded per frame is bounded to a maximum of 15 steps (i.e., the maximum number of steps the motor is able to perform within a frame of 25ms). To move the fuel valve as quickly as possible to the new position—that is, in a time roughly proportional[4] to the number of steps required to move from the current position to the desired position—the position-change command sequence passed to the actuator consists of a sequence of maximum step count commands followed by a single command with the remaining steps less or equal to the maximum step count.

SM_PCS maintains a record of the *desired position* and the position to be reached through the most recent position-change command (*commanded position*). On completion of the position command, the desired position, commanded position, and actual position of the motor are ex-

[4] We say “roughly proportional” because of possible delays in starting to move the valve and possible variations in the time it takes to move small distances rather than large distances. “Roughly proportional” is one way of capturing the informal specification “as quickly as possible.”

pected to be the same. A *Homing* command (to a fully closed position) is executed during initialization to synchronize the actual position with the initial desired and commanded position assumed by the SM_PCS.

4 An AADL Model of the SMS Architecture

In this section, we present an architecture specification of the SMS in three levels of abstraction using AADL and the Behavior Annex as modeling notation:

- SMS as the system of interest in its operational context
- the runtime architecture of the SMS as a set of interacting tasks
- the detailed functionality of each task and its interaction with the other tasks

We proceed by first presenting a model of SMS in its operational environment. This allows us to present requirements on SMS in the context of assumptions made by the SMS about the operational environment, and to examine the impact of potential failures in the operational environment on the SMS and vice versa. Next, we discuss the architecture model of the SMS implementation in terms of specifications of its components and their interactions. Emphasis is placed on capturing the execution and communication semantics between the components in order to address time sensitivity issues within SMS. This is followed by a specification of SMS component states and functional behavior. This specification provides the basis for quantifying the conditions under which some commanded steps may be missed (see Section 5.3). Finally, we present the two proposed fixes to SMS, and an alternative architecture design that greatly reduces the complexity of the position control system without significantly increasing the logic of the actuator.

Requirements and assumptions for the SMS and its components become claims in the confidence map or may be recorded as contextual assumptions.

4.1 The Operational Environment and Interface Specification of the SMS

The operational environment of the SMS is illustrated in Figure 8. SMS is shown to be part of a larger engine control feedback loop consisting of an ECS, the SMS as the actuator for the fuel valve of the engine, and the engine with a built-in thrust sensor. In addition, the operational environment consists of the computer hardware of the Electronic Control Unit (ECU) that executes the SMS software and a direct access memory that physically connects the ECU to the stepper motor interface unit, which resides within the SMS. Finally, the operational environment includes a power supply. For simplicity we use the AADL bus rather than an AADL memory component to represent direct access memory since it acts as a data transfer mechanism. For simplicity we assume a single power supply provides power to the ECU and the stepper motor inside the SMS.

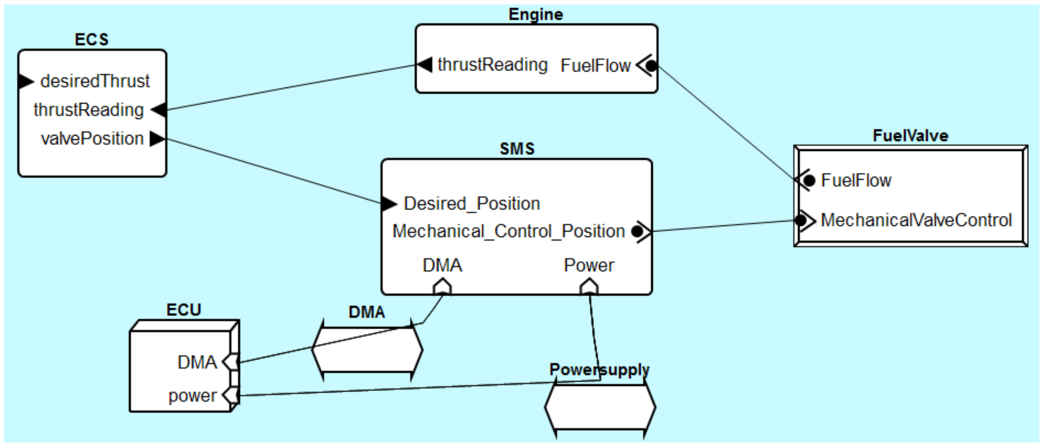


Figure 8: SMS *in Its Operational Environment*

The SMS and ECS are represented as AADL system components. This allows us to decompose the SMS as necessary to elaborate its architecture. The ECU is represented as an AADL processor, and the device bus for transferring data between any sensor, the actuator, and the processor as an AADL bus. The power supply is also modeled as an AADL bus, in this case transferring electricity. The fuel valve is represented an AADL device.

```
system SMS
features
  Desired_Position: in data port SM_Position.PercentOpen;
  Mechanical_Control_Position: out feature;
  DMA: requires bus access DirectAccessMemory;
  Power: requires bus access Power_Supply.Volt28;
end SMS;
```

Figure 9: Textual SMS *Interface Specification*

The interface specification of the SMS is shown textually in Figure 9. Incoming desired position commands are represented by a data port labeled *Desired_Position* with the data type *SM_Position.PercentOpen*. It specifies the requirement SMS-Req-1:

> The desired fuel-valve opening (i.e., the position of the stepper motor commanded by the ECS) shall be in terms of percent open with a value range of zero to *MaxPercentOpen*.

MaxPercentOpen is defined as a property constant with value 100 in the property set *PCSProperties*.

In a contract between the SMS and the ECS, this specification represents an assumption. This assumption is verified against the guarantee made by ECS (i.e., the data type specification of its outgoing port). Details of the data types used in the data model for the SMS can be found in Appendix A. It uses the Data Model Annex of AADL and specifies constraints on the data values as well as the measurement units to be used.

The mechanical interface between the stepper motor in SMS and the fuel valve is represented as an *abstract feature* called *Mechanical_Control_Position*. Its data type represents requirement SMS-Req2:

The actual position of the stepper motor driving the valve shall be expressed in units of *PercentOpen*.

SMS must ensure that the actual position is consistent with the commanded position. This leads to requirement SMS-Req-3:

At startup completion and at command completion the actual position of the stepper motor must be the same as the position commanded by the ECS.

This is expressed more formally as an invariant on the *Desired_Position* and the *Mechanical_Control_Position* using the Lute notation. This invariant must be verified on the model.

SMS must complete commands in a timely fashion expressed as requirement SMS-Req-4:

A desired position command shall be completed within *T = MaxPosition* * max(*StepDuration*).

This requirement is expressed by a Latency property value associated with a flow specification from the SMS *Desired_Position* input port to the *Mechanical_Control_Position* output feature. The property constant *MaxPosition* specifies the maximum stepper motor position in units of *Steps* when fully opened, with zero as fully closed. *StepDuration* is specified as property constant with a time range according to a data sheet. End-to-end latency analysis will verify whether the SMS system implementation meets this response time requirement.

SMS is expected to complete commands in proportion to the distance the stepper motor has to move from its current position to the desired position—expressed as requirement SMS-Req-5:

The command duration shall be proportional to the distance between the current and desired position (i.e., not exceed)
*roundup (| Desired_Position – Mechanical_Control_Position | * MaxStepCount/100*) * *Frame-Duration*).

The *Desired_Position* and *Mechanical_Control_Position* values are in terms of *PercentOpen*, thus, must be converted into number of steps. *MaxStepCount* has a value 15 steps per frame and *FrameDuration* has a value 25ms—specified as property constants in the property set *PCSProperties*. These two design decisions become derived requirements on the subcomponents of SMS.

SMS is expected to immediately respond to a new desired position command from ECS—requirement SMS-Req-6:

There shall be a delay of no more than *one frame* before responding to the newly received command.

This specific delay bound accommodates sampling delay of command input by SMS.

SMS uses a device bus to transfer data from SM_PCS to SM_ACT, that is, it requires access to a bus of a particular type (SMS-Req-7):

SMS shall access a bus of type *DirectAccessMemory*.

The *requires bus access* feature called *DMA* specified this requirement. Type matching along the bus access connection ensures that the correct bus type is physically connected to the SMS.

SMS also has a requirement for externally supplied power (SMS-Req-8):

SMS shall be supplied with 28-volt power.

This requirement is specified by the *requires bus access* feature called *Power* indicating that 28-volt power needs to be supplied by identifying the bus type *Power_Supply.volt28*. Type matching along the bus access connection ensures that the correct power supply is connected to the SMS.

Note that these requirements are recorded through a combination of RDAL requirement declarations and specification directly in the AADL model.

4.2 The SMS Architecture

The SMS consists of three components: digital position control software for the stepper motor SM_PCS, an actuator SM_ACT that translates commands from the position control software into electrical signals to a stepper motor, and the stepper motor SM_Motor.

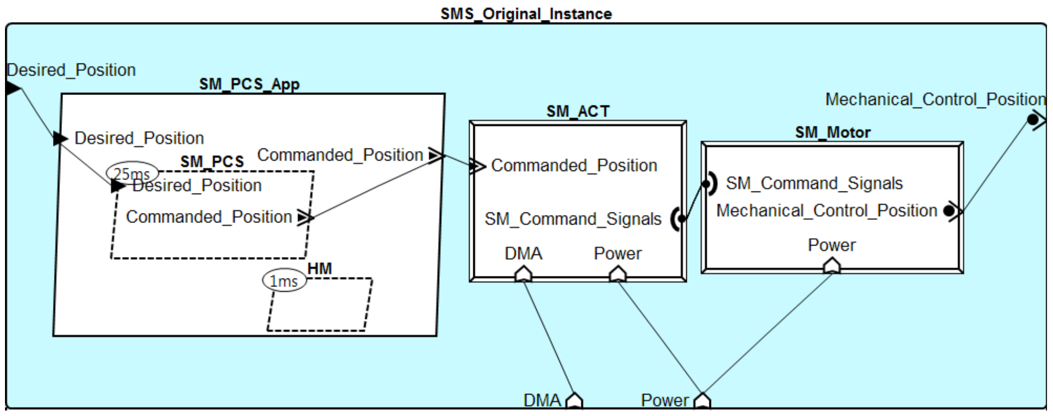


Figure 10: Details of Logical SM_PCS *Architecture*

Figure 10 shows the SMS architecture as a graphical view.

The position control system software SM_PCS is an AADL thread with a period of 25ms that resides in an AADL process called SM_PCS_App. The figure also shows a health monitor (SM_HM) thread with a period of 1ms in the same process that has no logical interaction with SM_PCS. This indicates that the two threads share the same address space; thus, a coding error in one can potentially affect the other.

A binding property in the operational environment of SMS, which contains the ECU, indicates that SM_PCS and SM_HM execute on the ECU. A *Priority* property indicates that SM_HM takes precedence over SM_PCS, thus, can affect the completion time of SM_PCS due to preemption. A *Scheduling_Protocol* property on the ECU indicates that preemptive scheduling is used.

The SM_ACT and SM_Motor are modeled as AADL devices to reflect that they are separate pieces of hardware. When developing the confidence map in Section 6, we will treat the two as a single assembly, referred to as the fuel-valve stepper motor or as SM.

We proceed by elaborating the AADL specification of each of the subsystems of SMS, annotating them with properties and constraints to represent derived requirements that must be met to satisfy SMS requirements.

4.2.1 The Position Control System Specification

We elaborate the specification of SM_PCS as shown in Figure 11.

```
thread SM_PCS
features
  Desired_Position: in data port SM_Position.Percentage;
  Commanded_Position: out event data port SM_Position_Change {
    Output_time => ([Time => Completion; Offset => 0 ns .. 0 ns;]);
    Output_Rate => [Value_Range => 0.40 .. 40.0; Rate_Unit => PerSecond;];
       };
flows
  flowpath: flow path Desired_Position -> Commanded_Position;
properties
  Dispatch_Protocol => Periodic;
  Period => PCSProperties::FrameDuration;
end SM_PCS;
```

Figure 11: Textual SM_PCS Interface Specification

SM_PCS inherits SMS-Req-1 on the incoming command from ECS on port Desired_Position (SM_PCS-Req-1). This is specified as a decomposition relationship between the requirement declarations and is recorded in the AADL model by a connection declaration from the incoming ECS port to the appropriate incoming SM_PCS port.

The data port Desired_Position for SM_PCS specifies that desired position commands are to be sampled with a period of 25ms (FrameDuration) (SM_PCS_Req-2). This derived requirement for SM_PCS contributes to meeting requirement SMS-Req-5.

SM_PCS has a requirement to convert the *Desired_Position*, specified in *PercentOpen* into units of *Steps* by rounding to the nearest step (SM_PCS-Req-3)

Desired_Position [Steps] = *round(MaxPosition* * *Desired_Position [PercentOpen*]/ 100).

SM_PCS has a requirement to provide actuator commands in units of *Steps* within the range of 0 to *MaxStepCount*, a direction of *Open* or *Close*, and a *StepRate* of 15 steps (*MaxStepCount*) (SM_PCS-Req-4).

The data type *SM_Position_Change* on the outgoing event data port *Commanded_Position* specifies the required data format for the position-change command (see Figure 37).

The SM_PCS has the requirement SM_PCS-Req-5 to command the actuator to the desired position. This derived requirement supports requirement SMS-Req-2. SM_PCS-Req-5 is expressed as the invariant that the sum of step counts in a position-change command sequence must be equal to the difference between the desired position and the commanded position at the time SM_PCS received the desired position command. In addition, the step direction must be consistent (i.e., it must be *Open* if *Desired_Position* > *Commanded_Position* otherwise it must be *Close*).

The commanded position (*Commanded_Position*) is a SM_PCS internal state that represents its understanding of the stepper-motor position and reflects the position-change commands that have been issued (i.e., SM_PCS' understanding of the current stepper-motor position) (see also Section 4.3.1). It needs to be initialized correctly to meet SMS-Req-3.

SM_PCS has requirement SM_PCS-Req-6 that all step counts in a position-change command sequence—except for the last non-zero one—must be equal to *MaxStepCount*. This requirement supports SMS-Req-4.

SM_PCS has requirement SM_PCS-Req-7 that the command stream has a rate of 40 commands per second. This is specified by the *Output_Time* property value of *Completion_Time* for the outgoing *Commanded_Position* port.

SM_PCS has requirement SM_PCS-Req-8 that once the desired position has been reached the position-change command will be issued with a step count of zero. This requirement is due to the fact that SM_ACT expects a command every period. This is reflected by the lower bound of *StepCount* of zero. In addition we have a Lute theorem identified in the requirement declaration that is used to validate this requirement on the behavior specification for SM-PCS.

4.2.2 The Actuator Specification

```
device SM_ACT
features
-- logical interface
  Commanded_Position: in event data port SM_Position_Change {
    Queue_Size => 0;
    Overflow_Handling_Protocol => Error;
  };
  SM_Command_Signals: feature group inverse of SM_Command_Signals;
-- physical interface
  DMA: requires bus access DirectAccessMemory;
  Power: requires bus access Power_Supply.Volt28;
flows
  flowpath : flow path Commanded_Position -> SM_Command_Signals;
properties
  Dispatch_Protocol => Aperiodic;
end SM_ACT;
```

Figure 12: Actuator SM_ACT Interface Specification

Figure 12 shows the interface specification for SM_ACT.

SM_ACT has requirement SM_ACT-Req-1 to process position-change commands with a specified step count, direction, and a step rate. This requirement is specified by the data type on the incoming Commanded_Position event data port.

SM_ACT has the requirement SM_ACT-Req-2 to immediately respond to commands from SM_PCS. This is specified in the AADL model by a Dispatch_Protocol property with the value Aperiodic for SM_ACT and by Commanded_Position as event data port to trigger the dispatch. The specification SM_ACT-Req-1 affects how end-to-end latency for the flow through SMS is calculated (SMS-Req-4).

An assumption that SM_ACT is always ready to accept position-change commands from SM_PCS leads to requirement SM_ACT-Req-3 that Commanded_Position port does not buffer incoming commands. This is specified by properties that the port has queue size of zero and an overflow handling protocol of Error. The overflow handling protocol of Error specifies that the previous command execution is aborted if it has not completed when the new command arrives. This requirement specification of not queuing incoming commands potentially affects the execution of all steps (SMS-Req-2). This specification must be verified to reflect the detailed behavior specification of SM_ACT (see Section 4.3).

SM_ACT is responsible for translating the commanded number of steps into electrical signals to SM to perform one step at a time. The interface between the actuator and the stepper motor is modeled by a feature group called SM_Step_Command_Signals with separate signal ports for Increment_Step, Decrement_Step, and Goto_Home. SM_ACT has the requirement to correctly support this interface specification (SM_ACT-Req-4).

SM_ACT has the requirement SM_ACT-Req-5 that within a frame the number of Increment_Step or Decrement_Step command signals must be equal to the commanded step count and must be consistent with the commanded direction. This requirement is expressed as a Lute theorem and contributes to the satisfaction of SMS-Req-2.

SM_ACT includes a requirement that the commanded number of steps can be completed within one frame, that is, MaxStepCount * max(StepDuration) <= FrameDuration (SM_ACT-Req-6). This value is specified as Latency value on the flow specification and will be used to calculate end-to-end latency for commands through SMS (i.e., to determine whether SMS-Req-4 and SMS-Req-5 are met).

The SM_ACT interface specification also includes required bus access declarations for the direct access memory of a specific bus type (SM_ACT-Req-7) and the power supply with a specific voltage (SM_ACT-Req-8). AADL model type checking ensures use of the correct bus and power supply.

4.2.3 The Stepper-Motor Specification

The stepper motor SM_Motor is specified as a device with dispatch protocol of Aperiodic indicating that it reacts to commanding signals from the actuator.

The interface with the actuator includes a signal from the stepper motor to indicate the completion of a step execution (Step_Completion event port in the feature group SM_Command_Signals). The detailed specification of the interface can be found in Figure 37 and is expected to be supported by SM_Motor (SM_Motor_Req-1). These command and response signals must match those of the outgoing feature group of SM_ACT. Type checking ensures matching assumption and guarantee.

SM_Motor also includes the specification of the external interface to the fuel value (SM_Motor-Req-2), which is inherited from SMS-Req-2.

The execution time of the stepper motor is specified to be the time range StepDuration, that is, the length of time it takes for the motor to move one step at the specified rate of 15 steps per frame (SM_Motor-Req-3). This time range corresponds to the range of steps per second documented in the stepper-motor specification sheet. This requirement assumes that the actuator has been instructed to operate at the MaxStep step rate.

4.3 SMS Functional Design as State and Behavior

In this section we elaborate the three components of SMS with a detailed functional design specification. Our focus is on state-based behavior and maintenance of state of the stepper motor by the control system (the combination of the position control system and the actuator).

Within SMS we deal with three states:

- the desired position of the stepper motor (i.e., the position requested by the ECS)
- the commanded position (i.e., the position that the position control system and actuator have asked the stepper motor to be at)
- the actual position of the stepper motor

As we will show, the interaction timing between the position control system and the actuator can lead to a discrepancy in the desired position and the commanded position when execution of the desired position command has completed. We do this by identifying the assumption in the actuator logic that all step commands have been issued to the stepper motor before the next command arrives. We are led to this potential issue by considering early command arrival as a potential hazard contributor.

4.3.1 The Position Control System State and Behavior

The position controller SM_PCS maintains two persistent state variables: the *DesiredPositionState* holding the most recently received *DesiredPosition* to be reached and the *CommandedPositionState* representing the position it has commanded the actuator to reach (i.e., the understanding by SM_PCS of the current stepper motor position). They are represented in the AADL model as persistent data subcomponents of the SM_PCS.*impl* thread implementation, as shown in Figure 13. We have the requirement that when the position-change command sequence has been completed the desired position and commanded position must be the same (SM_PCS-Req-5). SM_PCS must correctly translate the desired position command into the position-change command sequence (i.e., meet SM_ACT-Req-4, SM_ACT-Req-5, and SM_ACT-Req-6).

We use the Behavior Annex of AADL [SAE 2011] to specify the details of the functional behavior of the *Position_Controller* (see Figure 13). This specification has a single state that triggers a transition to itself on every dispatch. The transition action indicates that at every dispatch, if a new *Desired_Position* (command) value has been received, its range is checked. If the value is within range, it is converted to units of steps and recorded as *DesiredPositionState*. If it is not within range, the command is ignored. This provides runtime assurance that the desired position is in range (inherited requirement SMS-Req-1) (i.e., resilience to out-of-range values from the sender or any value corruption during transfer).

The stepper motor can execute steps at a specified rate with a maximum rate determined by the physical characteristics of the stepper motor. A fixed rate of 15 steps per frame (SPF) was chosen in the baseline design to minimize functional complexity and mechanical acceleration lag. Since the interface between SM_PCS and SM_ACT includes step rate as a parameter (see data type *SM_Position_Change*), SM_PCS is expected to set the step rate to 15 SPF—reflected in the property constant *MaxStepCount*. This design decision is specified by limiting the step rate field values in *SM_Position_Change* to *MaxStepCount*. Hence, when the motor is requested to move fewer than *MaxStepCount* steps (e.g., four steps), it executes the steps at the 15 SPF rate and then is idle until the end of the frame.

SM_PCS compares the desired position and commanded position states to determine whether the stepper motor needs to *Open* or *Close*, and how many steps must be performed. The *CommandedPositionState* is then updated to reflect the commanded change in position toward the desired position. By doing so, SM_PCS assumes that the commanded number of steps will actually be executed by the stepper motor (i.e., it relies on SM_ACT and SM_Motor to not introduce a missed step).

The behavior specification indicates that the maximum step count is used to reach the desired position until the position difference is fewer than 15 steps. At that point the actual difference value is used as the step count (necessary to meet SM_PCS-Req-6).

The SM_PCS responds immediately to a new desired target position by updating the *DesiredPositionState* with any new incoming *Desired_Position* value every 25ms (necessary to meet SM_PCS-Req-2 contributing to SMS-Req-5).

```
thread implementation SM_PCS.impl
subcomponents
  DesiredPositionState: data SM_Position.Steps;
  CommandedPositionState: data SM_Position.Steps;
annex Behavior_Specification {**
  variables
    distance: Base_Types::Integer;
    stepcount: Base_Types::Integer;
  states
   Ready: initial complete state;
  transitions
    Ready -[on dispatch]-> Ready { -- on every 25ms dispatch begin action
    -- check for out of range if a new command has been received
    if ((Desired_Position'fresh = true) and (Desired_Position >= 0 )
        and ( Desired_Position <= PCSProperties::MaxPercent)){
    -- convert from PercentOpen to Steps
      DesirePositionState := PCSProperties::MaxPosition*Desired_Position/100
    } end if;
    distance := DesiredPositionState - CommandedPositionState ;
    if (abs(distance)> PCSProperties::MaxStepCount)
      stepcount := PCSProperties::MaxStepCount
    else
      stepcount := abs(distance)
    end if;
    Commanded_Position.Step_Rate := PCSProperties::MaxStepCount;
    if (distance>0){
      Commanded_Position.Step_Direction := Open;
      Commanded_Position.Step_Count := stepcount;
      CommandedPositionState := CommandedPositionState + stepcount
    } else {
    -- this case handles steps in the close direction as well as zero steps
    -- note that zero step commands are expected to be issued
      Commanded_Position.Step_Direction = Close;
      Commanded_Position.Step_Count = stepcount;
      CommandedPositionState = CommandedPositionState - stepcount
    } end if;
    Commanded_Position!;
  }; -- end action
**};
```

```
end SM_PCS.impl;
```

Figure 13: Position Control System Behavior Specification

4.3.2 The Actuator State and Behavior

The stepper motor actuator also maintains system state in the form of a persistent *StepsToDo* state represented by a persistent data subcomponent of type *SM_Position_Change*.

Figure 14 shows the behavior specification for SM_ACT. The behavior is characterized by three states:

- *Ready*, indicating that it is waiting for a command from SM_PCS
- *WaitOnStep* to indicate that the execution of a step by *SM* is in progress
- *Decide* as an intermediate state dealing with the decision of whether there are steps left to be executed by *SM*.

Arrival of a *Commanded_Position* is handled by the *Ready* state and the *WaitOnStep* state (first two transitions in Figure 14). *StepsToDo* is set to the newly arrived value. An assumption is made that the count value at that time is zero, that is, all steps from the previous command have been issued (SM_ACT-Req-6). It is a necessary condition for SM_ACT-Req-2 and SM_ACT-Req- 4.

The first transition out of the *Decide* state determines that no step has to be taken. The other transition out of the *Decide* state specifies whether an *Increment_Step* or *Decrement_Step* signal is to be issued according to the specified Direction and updates the step count.

The transition out of the *WaitOnStep* state triggered by the arrival of the step completion signal leads to the *Decide* state, which determines whether additional steps are to be performed. A timeout may compensate for a missing completion signal from the motor.

```
device implementation SM_Act.impl
subcomponents
  StepsToDo: data SM_Position_Change.DataRecord;
annex Behavior_Specification {**
  states
    Ready: initial state;
    WaitOnStep: complete state;
    Decide: state;
  transitions
    Ready -[on dispatch Commanded_Position]-> Decide {
      StepsToDo := Commanded_Position
    };
    WaitOnStep -[on dispatch Commanded_Position]-> WaitOnStep {
      StepsToDo := Commanded_Position
    };
    WaitOnStep -[on dispatch DoStempCmd.StepDone]-> Decide ;
    Decide -[StepsToDo.Step_Count = 0]-> Ready ;
    Decide -[StepsToDo.Step_Count > 0]-> WaitOnStep {
      StepsToDo.Step_Count :=  StepsToDo.Step_Count - 1;
      if (StepsToDo.Step_Direction = Open)
        SM_Command_Signals.SM_Cmd.DoIncrement!(StepsToDo.Step_Rate);
      else
        SM_Command_Signals.SM_Cmd.DoDecrement!(StepsToDo.Step_Rate);
      end if
    };
**};
```

```
end SM_Act.impl;
```

Figure 14: Stepper-Motor Actuator Behavior Specification

4.3.3 The Stepper-Motor State and Behavior

The stepper motor *SM* has a persistent state as well, represented by the data subcomponent *ActualPositionState* in the *SM* device implementation. It represents the mechanical position of the stepper motor. The state is incremented or decremented according to the arriving command signal and the mechanical execution of the motor step.

4.3.4 SMS Design Constraints

In order for the SMS to operate correctly, the *DesiredPositionState* of SM_PCS, *CommandedPositionState* of SM_PCS, and *ActualPositionState* of *SM* must be the same when the SMS is not actively executing steps. This design constraint is declared as part of an RDAL requirement declaration, and supports the requirement SMS-Req-3 using SM_PCS-Req-5 and SM_ACT-Req-4.

During initialization of the SMS, the *Homing* command brings the actual state of the stepper motor to a known state, namely the zero position.

4.4 Three Proposed Corrections

In order to correct the missed step problem in the stepper motor control system, two corrections have been proposed by the developers. The first correction proposes to minimize variation in the time SM_PCS send the position-change command to SM_ACT. The second correction proposes to introduce buffering of position-change commands as they arrive at SM_ACT. In this section we describe the changes to the AADL model to reflect each of these corrections. We will also examine a third correction in the form of an architecture design change, in which SM_PCS does not issue position-change commands to the actuator, but passes on the desired position instead after validating that the command received from ECS is acceptable. We will show that this design alternative eliminates several design hazards and results in a more robust design without significantly increasing the functionality of the actuator.

4.4.1 The Fixed Command Send Time Solution

In this proposal SM_PCS sends the sequence of position-change commands at a fixed offset of 13ms from the beginning of a 25ms time frame—a new requirement on SM_PCS (SM_PCS-Req-9). The rationale is that this will minimize variation in inter-arrival time of the commands at the actuator.

We document this change in the SMS architecture model by changing the *Output_Time* property on the *Commanded_Position* port of SM_PCS to be the deadline of SM_PCS. Furthermore, we set the deadline for SM_PCS to 13ms, since a delay until the end of the frame was considered to be too long.

This specified communication time behavior can be implemented in two ways:

- by the runtime system—In addition to dispatching tasks at specified times the runtime system can initiate communication based on the specification in the AADL model. This solution avoids context switching overhead and results in negligible variation in send time. No further changes to the model are necessary.

- by an application I/O thread—The application thread is scheduled at the same 25ms period as SM_PCS, but with an offset dispatch time of 13ms. Note that in this case we have context switching overhead, and this thread is immediately preempted by the SM_HM thread running at a period of 1ms.

4.4.2 The Buffered Position-Change Command Solution

In this proposal the actuator will buffer incoming position-change commands until it has completed the execution of the previous command—a revision to requirement (SM_ACT-Req-2-rev). A buffer size of one was deemed sufficient since SM_PCS only sends one command per 25ms frame—an assumption that must be verified.

We document this change in the SMS architecture model by changing the queue size of the *Commanded_Position* port for SM_ACT to be of size 1.

The implementation specification of SM_ACT changes as follows. SM_ACT does not respond to the arrival of a command until it has reached the *Ready* state (i.e., we remove the transition out of *WaitOnStep* triggered by the port *Commanded_Position*).

4.4.3 The Desired Position Actuator Commanding Solution

In this solution the desired position is immediately passed on to the actuator after it has been converted from *PrecentOpen* units to *Steps* units and validated as an acceptable position to be commanded.

This requires a change in the interface specification of SM_PCS *Commanded_Position* and SM_ACT *Commanded_Position* to the data type *SM_Position*.

The behavior specification of SM_PCS is simplified to perform range checking of the incoming desired position value and conversion to *Steps*.

The complexity of the actuator functional behavior does not increase significantly compared to the original behavior logic (see Figure 14) and is shown in Figure 15.

The *StepsToDo* state variable is replaced by the *DesiredPositionState* and the *CommandedPositionState* variables. Instead of comparing the step count against zero and then issuing an increment or decrement command signal to the motor depending on the direction flag, SM_ACT now compares the most recently received desired position against the last commanded position and issues an increment or decrement command depending on whether the desired position is greater or less than the commanded position.

As we will see in the architecture fault analysis, this design is able to tolerate transient message corruption or loss by the direct access memory. The original design was sensitive to message corruption or loss.

```
device implementation SM_Act_SMPos.impl
subcomponents
  DesiredPositionState: data SM_Position.Steps;
  CommandedPositionState: data SM_Position.Steps;
```

```
annex Behavior_Specification {**
states
  Ready: initial complete state;
    WaitOnStep: complete state;
    Decide: state;
transitions
  Ready -[on dispatch Commanded_Position]-> Decide {
    DesiredPositionState := Commanded_Position
  };
  WaitOnStep -[on dispatch Commanded_Position]-> WaitOnStep {
    DesiredPositionState := Commanded_Position
  };
  WaitOnStep -[on dispatch DoStempCmd.StepDone]-> Decide ;
  Decide -[DesiredPositionState = CommandedPositionState]-> Ready ;
  Decide -[ DesiredPositionState != CommandedPositionState]-> WaitOnStep {
    if (CommandedPositionState > DesiredPositionState) {
      CommandedPositionState :=  CommandedPositionState - 1;
      SM_Command_Signals.SM_Cmd.Decrement_Step!(PCSProperties::MaxStepCount)
    } elsif (CommandedPositionState < DesiredPositionState) {
      CommandedPositionState :=  CommandedPositionState + 1;
      SM_Command_Signals.SM_Cmd.Increment_Step!(PCSProperties::MaxStepCount)
    } end if
  };
**};
end SM_Act_SMPos.impl;
```

Figure 15: Position-Commanded Actuator Behavior

5 Fault Analysis of the SMS Architecture

In this section we use architecture fault modeling to analyze the SMS architecture for software-induced hazards. In particular, we use fault impact analysis to identify all possible error sources in the original SMS design that can result in a missed step—a violation of SMS-Req-3. Using the fault propagation ontology of EMV2 we systematically identify potential sources of hazards and determine whether they can be assumed to be eliminated (e.g., satisfaction of SMS-Req-6) or whether they lead to additional (safety) requirements with respect to timing.

We annotate the SMS components and the components of the operational environment with error source and propagation information. We annotate the original design, the two proposed corrections, and the design alternative. We then analyze the resulting architecture fault models for fault impact and unhandled faults. The analysis will determine contributors to missed steps in the SMS and identify those that are avoidable in the design of the SMS. The analysis also gives us insight into the resilience of the SMS design to propagations from the operational environment. Finally, we quantify the failure conditions that can result in the missed step or delayed response to new commands, and derive safety requirements that must be met in order for the failure condition not to occur.

5.1 Hazards and Their Impact on the SMS Architecture

We annotate the AADL architecture with potential error sources and error propagations for each of the SMS components and components in the operational environment. We utilize the fault propagation ontology of EMV2, a library of pre-declared error propagation types, as a checklist to ensure we have considered all possible effects of failure conditions on other components.

In a first step, we focus on malfunctions of each component due to a defect in the component. The malfunction can lead to a violation of a requirement or a non-functional component. The effects of such a malfunction are observable as error propagations. These become possible defeaters in the confidence map.

We examine each of the outgoing ports and determine whether the component is the source of an error propagation of the error type's timing, rate, value, and omission. We use error type aliases for the error types to provide more meaningful names in the context of the component for which error sources and propagations are specified. For example, *MissingCommand* is used as an alias for *ItemOmission*.

We specify potential error sources and assumptions about their absence in SMS, its components, and the elements of the operational environment. We record error sources for each component within SMS by declaring out propagations for specific error types, and by identifying them as error sources via *error source* declarations.

In a second step, we specify how each system component deals with incoming error propagations—that is, whether a particular type of incoming error propagation is passed on as propagation of the same type, as propagation of a different type, and whether the component becomes a sink for the propagated error type. In this case study we infer these error flow specifications from the detailed design. In a new development these error flow specifications would become additional requirements on the system design.

Since the design of the SMS components has been specified and verified in SCADE, we assume that the functionality is correctly implemented (i.e., that value errors will not occur). We document this by an appropriate error containment declaration using the *not out propagation* construct of EMV2.

We also specify the fault model for SMS as a whole. This specification represents an abstraction of the error propagation behavior of the SMS implementation in terms of SM_PCS, SM_ACT, and SM_Motor. Such an abstracted fault model specification has several benefits. It allows various forms of safety analysis to be performed compositionally one layer at a time. For example, the error propagation guarantees and assumptions of SMS as a whole can be checked against those of the components in the operational environment separately from checking that the SMS implementation is consistent with the abstracted fault model of SMS. It also reflects a fault management strategy of a system (e.g., fail silent/fail stop) and a consistency check between the abstraction and the implementation ensures that the strategy is realized correctly.

5.1.1 The Position Control System SM_PCS Architecture Fault Model

SM_PCS is a potential source of timing and rate errors. Since SM_PCS sends the position-change command at completion time and completion time is variable, it can be a source of timing errors (i.e., early and late delivery). This is documented in the AADL model by declaring an outgoing error propagation of type *TimingError*, whose subtypes are *EarlyDelivery* and *LateDelivery* on the *Commanded_Position* port, and by identifying it as an error source (see the *timingsrc* error source declaration in Figure 16).

We also consider SM_PCS to be a source of *RateError*. First, the SM_PCS thread may get dispatched at a rate slower than the specified periodic rate, for example, due to a slow-running clock in the processor. Second, the receiving task (SM_ACT) operates asynchronously from the sender thread (SM_PCS) and is driven by the arrival of commands. Its maximum command execution rate is determined by the time it takes to execute the commanded steps by the stepper motor. This rate can potentially be lower than the rate of SM_PCS.

To address requirement SMS-Req-6 to immediately respond to a new command we use the error type *DelayedService* with the alias *DelayedResponse*. We specify that we do not expect SMS and any of its components to delay execution of a new command by declaring that SM_PCS will not be propagating this error type.

We assume that SM_PCS is not a source of value errors. In particular we assume that the step count value for the commanded position change is not out of range (expressed by the alias *StepCountOutOfRange* for the error type *OutOfRange*) and the resulting commanded position is not out of bounds (expressed by the alias *ResultingPositionOutOfRange* for the error type *OutOfBounds*). This is expressed by the *not out propagation* declaration on *Commanded_Position*.

```
thread SM_PCS
features
  Desired_Position: in data port SM_Position.PercentOpen;
  Commanded_Position: out event data port SM_Position_Change;
```

```
annex EMV2 {**
use types SMErrorTypes;
error propagations
-- outgoing errors
  Commanded_Position : out propagation {TimingError, RateError,
                           MissingCommand, NoCommandSequence};
-- errors not propagated
  Commanded_Position: not out propagation {StepCountOutOfRange,
                  ResultingPositionOutOfRange, DelayedReponse};
-- incoming errors
  Desired_Position : in propagation {MissingCommand, NoCommandSequence,
    OutOfRange};
-- errors not propagated in
  Desired_Position : not in propagation {DelayedReponse};
-- impact of processor
  Processor: in propagation {NoService, CompletionTiming};
flows
-- error sources
  timingsrc: error source Commanded_Position {TimingError};
  ratesrc: error source Commanded_Position {RateError};
-- error pass through
  omissionpPassthrough: error path Desired_Position{MissingCommand,
    NoCommandSequence} -> Commanded_Position;
-- map out of range into a missing command
  outOfRangeHandling: error path Desired_Position{OutOfRange} ->
    Commanded_Position {MissingCommand};
-- map ECU error propagation into no commands
  resourceserviceimpact: error path processor{NoService} ->
    Commanded_Position{NoCommandSequence};
end propagations;
**};
end SM_PCS;
```

Figure 16: Fault Model Specification of the Position Control System SM_PCS

We specify the following SM_PCS fault model behavior for dealing with incoming error propagations:

- SM_PCS does not assume incoming desired position commands are always within range. The incoming error propagation of type *OutOfRange* is mapped into an outgoing *MissingCommand* (alias for *ItemOmission*) by the error path declaration *outOfRangeHandling*. This specification results in a requirement that SM_PCS checks the range of the incoming desired position and ignores the command if out of range. This makes SM_PCS robust to out-of-range errors even though ECS not expected to intentionally propagate such errors.
- Any incoming *MissingCommand* or *NoCommandSequence* (alias for *ServiceOmission*) from the ECS is passed through as outgoing propagations of the same type (see error path *omissionPassthrough*).
- Incoming *NoService* error propagations from the processor SM_PCS is bound to are mapped into *NoCommandSequence* by the error path *resourceserviceimpact*

5.1.2 The Actuator SM_ACT Architecture Fault Model

The actuator SM_ACT fault model specifies the following fault behavior (Figure 17):

- SM_ACT assumes *StepCountOutOfRange* and *ResultingPositionOutOfRange* errors will not occur as incoming propagation on the *CommandedPosition* port.

- The functional logic of SM_ACT is assumed to be current under nominal conditions (i.e., has been verified by SCADE). In other words, SM_ACT is an error source declaration for the *MissingStepCommand* out propagations.
- SM_ACT can have a mechanical failure—it can be the source of a *NoCommandSequence* out propagation (alias for *ServiceOmission*)—expressed by the error source declaration *mechanicalFailure*.
- *EarlyDelivery* beyond a certain time limit will result in SM_ACT aborting the previous position-change command by overriding a non-zero step count. This is expressed by the error path declaration *EarlyDeliveryImpact* mapping *EarlyDelivery* into *MissingStepCommand*. As we will see from the analysis, this is the case if the new command arrives before the last step command has been issued by the actuator.
- Incoming *LateDelivery* results in a minor delay in stepper-motor response, indicated by the error path *LateDeliveryImpact* to outgoing error propagation *SlowResponse* (alias for *LateDelivery*).
- SM_ACT will immediately respond to a new command (i.e., we assume there is no *DelayedResponse* error source and there is no incoming *DelayedResponse* propagation).
- An incoming *HighRate* error results in an outgoing *MissingStepCommand* error propagation —expressed by the error path *Rateimpacthi*. This can occur when commands arrive faster than SM_ACT can process them, which has the same effect as early delivery.
- An incoming *LowRate* error results in a minor delay in stepper-motor response, indicated by the outgoing error propagation *SlowResponse*—expressed by the error path *Rateimpactlo*.
- SM_ACT can fail to provide a command sequence to the stepper motor when power fails to be supplied. This is expressed by the error path declaration *nopowerflow* from the incoming error propagation *Power*.

```
device SM_Actuator
features
features
-- logical interface
  CommandedPosition: in event data port SM_Position_Change{
       Queue_Size => 0;
       Overflow_Handling_Protocol => Error;
  };
  SM_Command_Signals: feature group inverse of SM_Command_Signals;
-- physical interface
  DMA: requires bus access DirectAccessMemory;
  Power: requires bus access Power_Supply.Volt28;
flows
       flowpath : flow path CommandedPosition -> SM_Command_Signals;
properties
       Dispatch_Protocol => Aperiodic;
annex EMV2 {**
use types SMErrorTypes;
error propagations
  Commanded_Position : in propagation { MissingCommand, NoCommandSequence,
    TimingError, RateError};
  Commanded_Position : not in propagation { StepCountOutOfRange,
             ResultingPositionOutOfRange, DelayedResponse};

  SM_Command_Signals.SM_Cmd : out propagation { MissingStepCommand,
             NoCommandSequence, SlowResponse};
  SM_Command_Signals.SM_Cmd : not out propagation {DelayedResponse};
  SM_Command_Signals.Step_Completion: not in propagation {CompletionOmission};
  Power : in propagation {NoService};
flows
  omissionPath1: error path CommandedPosition{ MissingCommand, NoCommandSequence} -
>  SM_Command_Signals.SM_Cmd(MissedStep);

  LateDeliveryImpact: error path CommandedPosition{ LateDelivery} ->
SM_Command_Signals.SM_Cmd(SlowResponse);

  EarlyDeliveryImpact: error path CommandedPosition{ EarlyDelivery} ->
SM_Command_Signals.SM_Cmd(MissingStepCommand);

  Rateimpacthi: error path CommandedPosition{ HighRate} ->
SM_Command_Signals.SM_Cmd(MissingStepCommand);

  Rateimpactlo: error path CommandedPosition{ LowRate} ->
SM_Command_Signals.SM_Cmd(SlowResponse);

  MechanicalFailure: error source
    SM_Command_Signals.SM_Cmd {NoCommandSequence} when {ActuatorFailure};
  nopowerflow: error path Power ->
      SM_Command_Signals.SM_Cmd {NoCommandSequence};
end propagations;
**};
end SM_Actuator;
```

Figure 17: Fault Model Specification of the Actuator SM_ACT

5.1.3 The Stepper-Motor SM_Motor Architecture Fault Model

The actuator SM_Motor fault model specifies its fault behavior as shown in Figure 18:

- SM_Motor can be the source of a mechanical stepper-motor failure—expressed by the error source declaration *SMFailure*. This failure becomes visible as *NoSteps* through the *Mechanical_Control_Position* feature.
- SM_Motor passes on an incoming *MissingStepCommand* as *MissedStep* (both aliases for *ItemOmission*).
- SM_Motor passes on *NoCommandSequence* as *NoSteps* (both aliases for *ServiceOmission*).
- SM_Motor passes on *SlowResponse* as *SlowResponse* and assumes *DelayedResponse* to a new command will not occur.
- Loss of power to SM_Motor turns into *NoSteps*.

```
device Stepper_Motor
features
  SM_Command_Signals: feature group SM_Command_Signals;
  Mechanical_Control_Position: out feature ;
  Power: requires bus access Power_Supply.Volt28;
flows
  flowsink: flow sink SM_Command_Signals{Latency => 1 ms .. 1 ms;};
properties
  Dispatch_Protocol => Aperiodic;
  Compute_Execution_Time => PCSProperties::StepDuration;
annex EMV2 {**
use types SMErrorTypes;
error propagations
  SM_Command_Signals.SM_Cmd : in propagation {MissingStepCommand,
     NoCommandSequence, SlowResponse};
  SM_Command_Signals.SM_Cmd : not in propagation {DelayedResponse};
  SM_Command_Signals.Step_Completion: not out propagation {CompletionOmission};
  Power: in propagation{NoService};
  Mechanical_Control_Position:  out propagation
    {MissedStep, NoSteps, SlowResponse};
  Mechanical_Control_Position:  not out propagation {DelayedResponse};
flows
  SMfailure: error source Mechanical_Control_Position{NoSteps}
    when {StepperMotorFailure};
  cmdimpact1: error path SM_Command_Signals.SM_Cmd{MissingStepCommand}
    -> Mechanical_Control_Position{MissedStep};
  cmdimpact2: error path SM_Command_Signals.SM_Cmd{NoCommandSequence}
    -> Mechanical_Control_Position{NoSteps};
 cmdlate1: error path SM_Command_Signals.SM_Cmd{SlowResponse}
    -> Mechanical_Control_Position{SlowResponse};
 nopower: error path Power{NoService} -> Mechanical_Control_Position{NoSteps};
end propagations;
**};
end Stepper_Motor;
```

Figure 18: Fault Model Specification of the Stepper Motor SM_Motor

5.1.4 The Engine Control System ECS Architecture Fault Model

The engine control system ECS fault model specifies the following fault behavior (Figure 19):

- ECS can be the source of *NoCommandSequence* and *MissingCommand* propagations due to ECS failure.
- ECS is expected to provide desired position commands within range position values (i.e., does not propagate *OutOfRange* position values).

The ECS determines the desired percentage of fuel-valve opening to achieve desired fuel flow and thrust of an engine. We specify that the desired position will be in range, and that an ECS failure will result in lack of desired position commands (*MissingCommand* and *NoCommandSequence).*

```
system EngineControlSystem
features
  desiredThrust: in data port ;
  thrustReading: in data port ;
  valvePosition: out data port SM_Position;
flows
  flowsource : flow source valvePosition{Latency => 1 ms .. 1 ms;};
annex EMV2{**
use types SMErrorTypes;
error propagations
  valvePosition: out propagation {NoCommandSequence, MissingCommand};
  valvePosition: not out propagation {OutOfRange};
flows
  ecsfailure: error source valvePosition{NoCommandSequence, MissingCommand}
    when {ECSFailure};
end propagations;
**};
end EngineControlSystem;
```

Figure 19: Fault Model Specification of ECS

5.1.5 The ECU Architecture Fault Model

The ECU can fail, that is, it is an error source of type *NoService* (alias for *ServiceOmission*). The ECU can propagate errors to any software component bound to it, specified by an outgoing propagation declaration for **bindings**. *NoService* is also propagated to the direct access memory connected to the ECU. The ECU is affected by *PowerLoss* from the power supply, which is passed on as a *NoService* error propagation to the software units bound to the ECU.

5.1.6 The Direct Access Memory Architecture Fault Model

The architecture fault model of the device bus is shown in Figure 20. The direct access memory can be the source of *MessageLoss* (alias of *ItemOmission*) and *MessageCorruption* (alias for *ValueError*) to any connection bound to the bus. This is declared by the error source declaration *Commerror* for the *bindings* propagation point.

In addition, the device bus passes on incoming *NoService* propagations from the ECU—expressed by the error path declaration *NoECUService*.

```
bus DirectAccessMemory
annex EMV2 {**
use types SMErrorTypes;
error propagations
  bindings: out propagation {NoService, MessageLoss, MessageCorruption};
  access: in propagation {NoService};
flows
  NoECUService: error path access {NoService} -> Bindings(NoService);
  Commerror: error source bindings{MessageLoss, MessageCorruption}
    when {DeviceBusFailure};
end propagations;
**};
end DirectAccessMemory;
```

Figure 20: Fault Model Specification of the Direct Access Memory

Propagations from the direct access memory affect the connection between SM_PCS and SM_ACT. Type transformation rules associated with the connection specify that *NoService* and *MessageLoss* from the bus result in *NoCommandSequence* and *MissingStepCommand*. *MessageCorruption* results in a *ValueError* for the commanded position arriving at SM_ACT. Note that a value error can be subtle or detectable out-of-range value.

5.1.7 The Power Supply Architecture Fault Model

The power supply is the error source for PowerLoss, which is propagated to all components connected to this power source—in our example, to the ECU, actuator, and stepper motor.

5.1.8 The Fuel-Valve Architecture Fault Model

The fuel valve can fail (ValveFailure). In addition, incoming MissedStep errors are mapped to valve errors in the form of IncorrectFlow. Similarly, an incoming NoSteps error has the effect of a StuckValve. A slow response by the stepper motor propagates as SlowResponse. We assume that delayed response to a new command will not occur—expressed by not in/out propagation for DelayedResponse.

```
device Valve
```

```
features
  MechanicalValveControl: in feature;
  FuelFlow: out feature;
annex EMV2 {**
use types SMErrorTypes;
error propagations
  MechanicalValveControl: in propagation { NoSteps, MissedStep,
    SlowResponse};
  MechanicalValveControl: not in propagation {DelayedResponse};
  FuelFlow: out propagation {StuckValve, IncorrectFlow, SlowResponse};
  FuelFlow: not out propagation {DelayedResponse};
flows
  FailedValve: error source FuelFlow{StuckValve} when {ValveFailure};
  MapToStuckValve: error path MechanicalValveControl{NoSteps}
    -> FuelFlow{StuckValve};
  MapToIncorrectFlow: error path MechanicalValveControl{MissedStep}
    -> FuelFlow{IncorrectFlow};
  MapToSluggishResponse: error path MechanicalValveControl{SlowResponse}
    -> FuelFlow{SlowResponse};
end propagations;
**};
end Valve;
```

Figure 21: Fault Model Specification of Fuel Valve

5.1.9 The SMS Architecture Fault Model

For the SMS we specify fault model information for the SMS implementation and an abstracted fault model for the SMS. The fault information for the SMS implementation relates to the interaction between the SMS components (i.e., error behavior associated with connections). The abstracted fault model for SMS allows us to understand the fault behavior of the SMS in the context of its operational environment without requiring access to the SMS implementation—it facilitates compositional fault model analysis.

The abstracted fault model will be checked for consistency with the fault models of the system implementation components. The incoming and outgoing error propagation declarations must be consistent with those in the SMS implementation subcomponents that are connected to each of these interaction points (port, bus access, and feature). Similarly, error source, error path, and error sink declarations must be consistent with the fault behavior of the SMS subcomponent error models.

A set of type transformation rules specifies how the error propagations from the direct access memory affect the commanded position command being transmitted between SM_PCS and SM_ACT. *NoService* and *MessageLoss* from the bus result in *NoCommandSequence* and *MissingStepCommand*. *MessageCorruption* results in a *ValueError* for the commanded position arriving at SM_ACT. This type transformation set is associated with the *SendPositionChangeCommand* connection in the *connection error* section of the EMV2 subclause in the SMS.Original system implementation.

The abstracted fault model for the SMS is specified in the system type (shown in Figure 22). It documents assumptions made about incoming propagations to SMS and how SMS handles them, about outgoing propagations from SMS, and whether SMS is the source. This specification is common to all implementations of SMS.

For the SMS variant, we document that it is the error source for *MissedStep* in the system implementation declaration *SMS.Original* as shown in Figure 23. This specification inherits the EMV2 declarations from the system type SMS.

```
System SMS
features
  Desired_Position: in data port SM_Position;
  Mechanical_Control_Position: out feature;
  DMA: requires bus access DirectAccessMemory;
  Power: requires bus access Power_Supply.Volt28;
annex EMV2 {**
use types SMErrorTypes;
error propagations
  DesiredPosition: in propagation {NoCommandSequence, MissingCommand,
     OutOfRange};
  Mechanical_Control_Position: out propagation {MissedStep, NoSteps,
    SlowResponse};
  Mechanical_Control_Position: not out propagation {DelayedResponse};
  Power: in propagation {PowerLoss};
  Processor: in propagation {NoService, CompletionTiming};
  Connection: in propagation {NoService, MessageLoss, MessageCorruption};
flows
  DPpath1: error path DesiredPosition{NoCommandSequence}
    -> Mechanical_Control_Position{NoService};
  DPpath2: error path DesiredPosition{MissingCommand}
    -> Mechanical_Control_Position {MissedStep};
  DPpath3: error path DesiredPosition{NoCommandSequence}
    -> Mechanical_Control_Position {NoService};
  -- mapping of power errors
  powerpath: error path Power{PowerLoss}
    -> Mechanical_Control_Position (NoService);
  ECUpath: error path Processor{NoService}
    -> Mechanical_Control_Position {NoService};
  Connectionpath: error path Connection {MessageCorruption}
    -> Mechanical_Control_Position {MissedStep};
end propagations;
**};
end SMS;
```

Figure 22: Abstracted Fault Model Specification for SMS

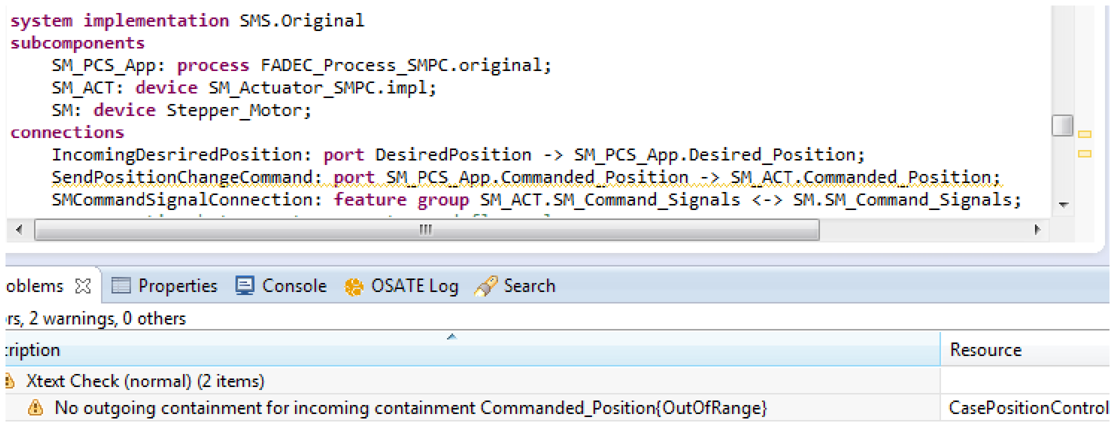
```
System implementation SMS.Original
-- subcomponents and connections
annex EMV2 {**
use types SMErrorTypes;
error propagations
flows
  MissedStepSource: error source Mechanical_Control_Position{MissedStep};
end propagations;
**};
end SMS.Original;
```

Figure 23: SMS Variant Specific Fault Information

5.2 Fault Impact Analysis of the Original SMS Architecture

In this section we use fault impact analysis on the architecture fault model to determine potential contributors to the missed step problem, to identify potentially unhandled faults, and to assess the resilience of SMS to fault propagations from the operational environment. In the next section we will quantify timing-related error propagation types as timing conditions that must be satisfied for the error propagation not to occur. These conditions become derived safety requirements on the SMS components.

We create an instance model of the system implementation *SMS.Original* in its operational environment and perform fault impact analysis on it. In the process the EMV2 consistency checker will ensure that all incoming error propagation assumptions are met by outgoing propagation guarantees and that the propagation specifications in the abstracted SMS fault model are consistent with those of the SMS components. For example, the consistency checker would give a warning if we had forgotten to specify that SM_PCS will not propagate *OutOfRange* errors, an assumption made by SM_ACT (shown in Figure 24).



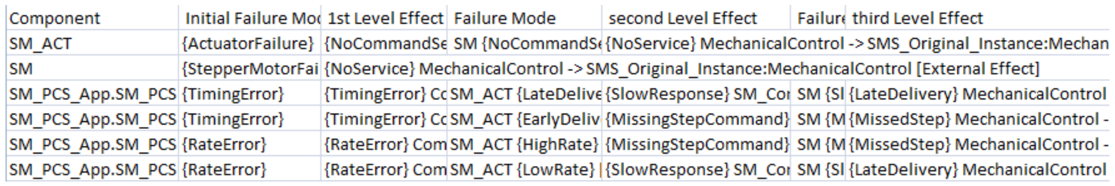
```
system implementation SMS.Original
subcomponents
    SM_PCS_App: process FADEC_Process_SMPC.original;
    SM_ACT: device SM_Actuator_SMPC.impl;
    SM: device Stepper_Motor;
connections
    IncomingDesriredPosition: port DesiredPosition -> SM_PCS_App.Desired_Position;
    SendPositionChangeCommand: port SM_PCS_App.Commanded_Position -> SM_ACT.Commanded_Position;
    SMCommandSignalConnection: feature group SM_ACT.SM_Command_Signals <-> SM.SM_Command_Signals;
```

```
oblems ✕   Properties   Console   OSATE Log   Search
rs, 2 warnings, 0 others
ription                                                                              Resource
  Xtext Check (normal) (2 items)
    No outgoing containment for incoming containment Commanded_Position{OutOfRange}  CasePositionControl
```

Figure 24: Missing Error Containment Guarantee Specification

Fault impact analysis traces propagation of every error source, representing a failure mode, to components impacted by the effect of the propagation, and does so multiple levels deep. A sample of the resulting report is shown in Figure 25.

From this report we can identify all the error sources that result in a missed step (i.e., a violation of SMS-Req-3). The report shows that timing errors in the form of *EarlyDelivery* and rate errors in the form of *HighRate* can result in missed steps, while *LateDelivery* and *LowRate* result in *SlowResponse*. In the next section we will quantify the amount of time for early delivery that results in a missed step as well as the deviation from the expected rate at which commands are supplied. Slow response within bounds by the SMS is considered an acceptable risk for the ECS.



Component	Initial Failure Mo	1st Level Effect	Failure Mode	second Level Effect	Failur	third Level Effect
SM_ACT	{ActuatorFailure}	{NoCommandSe	SM {NoCommandS	{NoService} MechanicalControl -> SMS_Original_Instance:Mechan		
SM	{StepperMotorFai	{NoService} MechanicalControl -> SMS_Original_Instance:MechanicalControl [External Effect]				
SM_PCS_App.SM_PCS	{TimingError}	{TimingError} Cc	SM_ACT {LateDelive	{SlowResponse} SM_Co	SM {Sl	{LateDelivery} MechanicalControl
SM_PCS_App.SM_PCS	{TimingError}	{TimingError} Cc	SM_ACT {EarlyDeliv	{MissingStepCommand}	SM {M	{MissedStep} MechanicalControl -
SM_PCS_App.SM_PCS	{RateError}	{RateError} Com	SM_ACT {HighRate}	{MissingStepCommand}	SM {M	{MissedStep} MechanicalControl -
SM_PCS_App.SM_PCS	{RateError}	{RateError} Com	SM_ACT {LowRate} |	{SlowResponse} SM_Co	SM {Sl	{LateDelivery} MechanicalControl

Figure 25: Fault Impact Report for SMS

The report also shows the impact of mechanical failures, of power loss, and of data corruption by the direct access memory. Mechanical failures and power loss are typically identified during a system safety analysis and the impact is as expected unless addressed by redundancy.

Data corruption is a more interesting case. Transient data corruption on the direct access memory can be due to factors such as vibration, heat, or radiation. Such corruption leads to value errors, both detectable (in that the step count value is out of range), and subtle (where the step count is incorrect but within range or where the direction is flipped, resulting in potentially commanding outside the acceptable range of positions and in inconsistency between the actual position and the commanded position (SMS-Req-3)).

When performing fault impact analysis the consistency checker will flag the fact that value errors are being propagated from the direct access memory into the connection between SM_PCS and SM_ACT. However, SM_ACT assumes that it is not receiving any out-of-range values. In other words, we have unhandled faults originating in the direct access memory. A possible correction of this problem is for SM_ACT to check for out-of-range values and ignore them.

5.3 Quantified Time-Sensitive Derived Safety Requirements

In this section we quantify the fault condition for missed steps due to early delivery and mismatched command rate (SMS-Req-3), as well as the condition that must be met in order to assure immediate response to new commands (SMS-Req-6). These requirements apply to any stepper motor or other control system with a command recipient that does not buffer commands and assumes a constant command stream.

5.3.1 Derived Requirement for Early Command Delivery

Early delivery of a position-change command to SM_ACT can result in a missed step if the command arrives while the step count for the previous command is non-zero. This is due to the fact that the newly arriving command is responded to immediately by aborting the previous command as expressed in the interface specification in Figure 12—implemented by resetting the step count to the new command value even though the old command execution is still in progress as expressed abstractly in Figure 14.

This leads to the following condition that must be satisfied in order to avoid a missed step. The worst-case inter-arrival time variation for commands arriving at SM_ACT must be less than the time difference between the next frame and the latest time for a non-zero step count value, that is, the derived requirement for the SMS implementation
SMS-Req-D1: `Delta(InterarrivalTime) < StepMissBound.`

The inter-arrival time variation is determined by the variation in the time at which the command is sent by SM_PCS and variation in communication time, that is
`Delta(Interarrival) = Delta(Send`$_{SM\_PCS}$`)+ Delta(Comm)`

Appendix B provides a formula for the inter-arrival time variation for the specific task set executing on the ECU using the worst-case send time assumption of sending at the end of task execution. For the general case, a scheduling analyzer can provide best-case and worst-case task completion times for SM_PCS.

The value of step count is non-zero until the last step of a position-change command has been issued (i.e., until (Step_Count -1) * Step_Duration). This results in a worst-case step miss bound for the maximum acceptable variation of inter-arrival time of
`StepMissBound = 25ms – ((MaxStepCount -1) * max(Step_Duration))`

According to specifications for the nominal rate of 600 steps per second, the step duration varies between 578 (1.730ms step duration) and a maximum of 621 steps (1.61ms step duration). Using the maximum step duration as worst case, this results in a step miss bound of 0.78ms. Notice that this bound is less than the minimum duration of one step.

When we compare the step miss bound against the maximum inter-arrival variation, we determine whether the sum of communication variation, execution time variation in SM_PCS, maximum execution time of HM, and multiples of execution time variation in HM can possibly exceed 0.78ms.

This verification constraint has been specified as a Lute theorem that interprets the relevant model properties to determine whether the maximum inter-arrival time variation exceeds the step miss bound. Evaluation of this theorem provides analytical evidence as to whether a missed step can occur.

We have also generated a prototype implementation of the SMS architecture in Java. This implementation of SMS has been exercised with execution time variations as specified in the AADL model and has resulted in missed steps. This provides further evidence that the problem of missed steps is inherent in the original SMS design.

5.3.2 Derived Requirement for Rate Mismatch

A rate mismatch between the sender SM_PCS and the receiver SM_ACT can occur for two reasons:

- SM_PCS executes at a rate faster than the specified 25ms frame rate
- SM_ACT requires more than 25ms to complete the execution of the position-change command.

SM_PCS can execute faster than 25ms if the hardware clock of the ECU operates faster. For our case study we assume that this is not the case.

As mentioned earlier, the data sheet for the stepper motor indicates that the stepper motor's step duration varies between 578 and 621 steps per second when executing at the rate of 15 steps per frame. This results in an SM_ACT completion time variation between 24.15ms and 25.95ms for performing 15 steps.

In other words, SM_ACT potentially may execute at a lower rate than the rate at which SM_PCS sends commands.

In a worst-case scenario, the stepper motor could continuously operate at the longest step duration if it is caused by mechanical conditions, such as increased friction due to high temperature, falling behind 0.95ms for every frame that executes a position-change command of 15 steps.

We observe that the completion delay is cumulative for a sequence of consecutive maximum step count commands. Note that SM_PCS sends the maximum step count only until the desired position is reached. A step count less than the maximum allows the stepper motor to catch up with the S*M_ACT* commands and make up for the time delay.

This leads to the derived requirement for the SMS implementation
SMS-Req-D2: `max(StepDuration) * MaxStepCount * max(MaxStepCountCommandSequenceLength) < StepMissBound.`

It takes 16 commands (250/15) to go from a completely closed to a completely open position with a cumulative delay of 15.2ms. This figure is still less than one frame length (25ms) but larger than the step miss bound, leading to missed steps.

To make matter worse, the maximum step count sequence potentially can be longer. A new desired position may be issued before the previous one has been reached. The new position may be in the opposite direction from the current position. As a result, the sequence of maximum step commands is extended. If the controller shows oscillating behavior we may face a potentially unbounded sequence of maximum step commands.

The condition for a rate mismatch between SM_PCS and SM_ACT has been specified as a Lute theorem. When applied to the SMS model it will inform us whether such a rate mismatch is possible for a given range of step duration values.

5.3.3 Derived Requirement for Immediate Command Response

SM-Req-6 specifies that SMS immediately respond to a new desired-position command, even when the previously commanded desired position has not been reached. This requirement can be violated if the new command is queued and processed only after completion of the previous command. In the architecture fault model we have specified by the error type alias *DelayedResponse* that this is assumed not to occur.

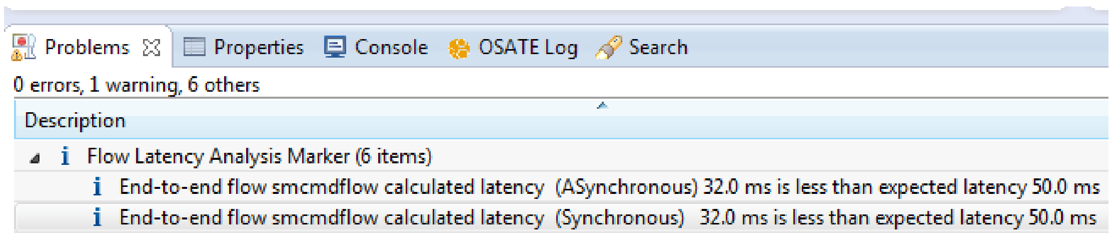
We can validate that such behavior does not occur in two ways. First, we specify a derived requirement *SMS-Req-D3* reflecting acceptable delay in immediate processing is caused by queuing ports in communication or by buffering of the command in the SM_PCS internal program logic. We have specified a Lute theorem to determine whether the *DesiredPosition* command is queued in its processing path. When applied to the original SMS model it evaluates to the condition being satisfied.

Second, we specify a derived requirement *SMS-Req-D4* of maximum the end-to-end latency for initiating the first step of a new desired-position command. An acceptable delay is less than two frames: one frame to accommodate sampling delay by SM_PCS and one frame to push the execution of the first step through to the stepper motor. For that purpose, we have specified an end-to-end flow in the architecture model with a maximum expected latency (see Figure 26). End-to-end latency analysis is used to determine whether the requirement is satisfied. End-to-end latency analysis takes into account latency contributions by periodically sampling tasks and by aperiodic tasks with possible queuing delays as well as worst-case task completion time and communication time.

```
flows
  smcmdflow: end to end flow
    CS.flowsource -> valvecmd -> SMS.flowsink {Latency => 50 ms .. 50 ms;};
```

Figure 26: End-to-End Flow Specification for SMS *Commands*

The results of an end-to-end latency analysis on our example show that the end-to-end latency numbers are within the expected processing delay of 50ms (see Figure 27).


Problems Properties Console OSATE Log Search

```
0 errors, 1 warning, 6 others
Description
  i  Flow Latency Analysis Marker (6 items)
       i  End-to-end flow smcmdflow calculated latency  (ASynchronous) 32.0 ms is less than expected latency 50.0 ms
       i  End-to-end flow smcmdflow calculated latency  (Synchronous)  32.0 ms is less than expected latency 50.0 ms
```

Figure 27: End-to-End Latency for the Original Design

5.4 Analysis of the Fixed Command Send Time Solution

The fixed command send time solution uses a fixed offset from the frame for SM_PCS to send the position-change command. This affects the architecture fault model in two ways:

- It reduces the inter-arrival time variation considerably, possibly guaranteeing a value below the *StepMissBound*.
- It introduces a *DelayedDelivery* if SM_PCS normally completes below the specified offset. This additional delay results in a *SlowResponse* behavior by the control system. The quantified increase in control lag must be assessed at the ECS for acceptability.

The fixed command send time solution does not change the derived requirement SMS-Req-D2 addressing rate mismatch. In other words, the condition for a potential missed step due to rate mismatch is the same as for the original design.

Although *DelayedDelivery* introduces a delay in response time to a command, the delay is within a frame and is accounted for by SMS-Req-D4.

5.5 Analysis of the Buffered Position-Change Command Solution

The buffered position-change command solution uses a buffer for position-change commands arriving at SM_ACT. This affects the architecture fault model in the following way:

- It allows SM_ACT to accommodate for *EarlyDelivery* (i.e., SM_ACT becomes a sink of this error propagation—eliminating the need for SMS-Req-D1). A queue size of one is sufficient because only one command is sent per frame assuming no rate error.

The buffered position-change command solution changes the derived requirement SMS-Req-D2 addressing rate mismatch by increasing the threshold from `StepMissBound` to `StepMissBound + 25ms`. However, given the potential for unbounded maximum step count command sequences the mismatched rate problem still exists.

The buffered position-change command solution introduces a queue for commands to SM_ACT. This queue applies only to position changes delaying its early arrival until the expected time. We reflect this in SMS-Req-D3. From an end-to-end latency perspective the delay in response time to a command is less than a frame and is accounted for by SMS-Req-D4.

5.6 Analysis of the Position-Commanded Actuator Design Alternative

In the position-commanded actuator design solution SM_PCS is reduced to checking the range of the incoming desired position and to sending the desired position to the actuator every 25ms. This affects the architecture fault model in two ways:

- *EarlyDelivery* propagation to SM_ACT results in a *FastResponse* propagation instead of a *MissingStepCommand*.
- Similarly, *HighRate* propagation to SM_ACT results in a *FastResponse* propagation instead of a *MissingStepCommand*.

This eliminates the error sources within SMS and derived requirements SMS-Req-D1 and SMS-Req-D2 to address missed steps.

The position-commanded actuator design solution does not introduce unnecessary buffering of desired position commands (i.e., SMS-Req-D3 and SMS-Req-D4 will be satisfied).

The original solution was sensitive to value corruption by the direct access memory. A corrupted step count would result in SM_ACT commanding the stepper motor to an incorrect actual position without the opportunity to detect and correct the error. By sending the desired position repeatedly, SM_PCS addresses transient corruption of the commanded position. SM_ACT may temporarily move towards an incorrect position (subtle value error) and, by checking the range, avoid out-of-range positioning. However, assuming transient corruption, uncorrupted transfer of the desired position in succeeding periods self-corrects the transient error. In other words, this design is resilient to transient data corruption by the direct access memory.

5.7 Comparison of the SMS Designs

Table 2 presents a comparison of the four architecture design alternatives in terms of their fault models. The first row focuses on logical failures in the SMS design, the second row describes mechanical failures within the SMS, the third row captures the effects of computer hardware on the SMS, and the last row represents mechanical failures in the operational environment.

The comparison shows that the position-commanded actuator design is not sensitive to early delivery or high rate errors, nor is it sensitive to transient message corruption or loss, while the original design and the two corrections are sensitive to transient data corruption. This is due to the design choice of commanding the actuator by desired position rather than by a sequence of position-change commands. The two corrections to the missed step problem, fixed send time and buffered command, address early delivery as a cause of missed steps, but do not address rate mismatch.

We can also see that mechanical failures affect the SMS the same way in both designs and must be addressed at the enclosing system level (e.g., by replication of the engine control system and the engine).

Table 2: Comparison of Architecture Design Alternatives

Missed Step	**Original Design**	**Fixed Send Time**	**Buffered Command**	**Position Command**
SMS logical failures	EarlyDelivery HighRate	HighRate	HighRate	
SMS mechanical failures	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure	ActuatorFailure StepperMotorFailure
Transient comm failures	MessageCorruption MessageLoss	MessageCorruption MessageLoss	MessageCorruption MessageLoss	
Mechanical failures in Op Environment	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure	ECUFailure PowerLoss ValveFailure

The architecture fault model analysis is a good source for developing a confidence map for each design alternative. The top-level claim of the SMS confidence map reflects a safety requirement that the actual stepper-motor position and the position known to SMS must be consistent.

Error source specifications of system components can become potential defeaters if they impact system properties that reflect the safety-related requirements. In the case of the stepper motor, one example is the safety requirement stating that the fuel valve must reach the desired position. This requirement can be violated if the SMS position control function can result in missed step commanding. Fault impact analysis tells us which error sources can impact the derived requirement to command the correct number of steps.

Finally, the architecture fault model includes assumptions about the absence of certain error propagations. In particular, the fault model interface specification of SM_PCS states that it is not a source of computational errors. This claim is addressed in the confidence map.

6 Establishing Confidence in the SMS

In this section we develop confidence maps for the baseline architecture design as well as for one of the alternative designs (the position-commanded design). We draw on the insights gained from the architecture fault model analysis to help us identify error sources as defeaters, in particular those that impact the operation in an undesirable manner.

6.1 Confidence Maps for SMS

The purpose of this section is twofold:

1. to show how developing a confidence map can help identify specific areas where verification and validation (V&V) efforts should be focused for a given design
2. to show how developing confidence maps for alternative designs can help with choosing the one that will lead to higher levels of justified confidence at a lower expenditure of scarce assurance resources

This section will discuss two of the candidate designs (the original design and the position-commanded design analyzed in the previous section) and what the confidence map for each tells us about the system. Since the problem area being investigated is primarily in the interaction between the position control system and the actuator that drives the stepper motor for the fuel valve, we consider only two subsystems as an architecture abstraction: the position control system for the fuel-valve stepper motor (referred to as SM_PCS), and the stepper motor that is mechanically connected to the fuel valve (*SM*). SM_PCS includes the conversion from a fuel-valve position (represented as a percent open *Desired_Position* as requested by the ECS) to the position in terms of stepper-motor steps *Mechanical_Control_Position*. *SM* includes the stepper-motor actuator (*SM_ACT* in the AADL model), the stepper motor (*SM_Motor*), and the fuel valve. In the AADL representation of the system these are separate components, a distinction not necessary here.

When creating confidence maps we use requirements associated with the architecture model to establish claims. We also use the specification of faults that manifest themselves as violation of requirements or nonfunctioning components, or the assumed absence of specific faults to identify defeaters and assertion of their absence. The decomposition of SMS requirements into requirements on its subsystems as well as the derivation of requirements is reflected in the hierarchical structuring of claims and defeaters.

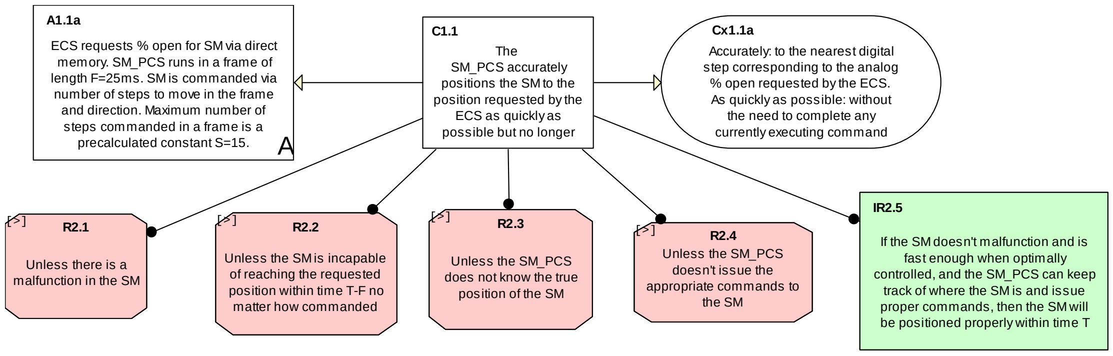
The confidence map not only considers defects in the system design, but also potential hazards in the development and verification process that may lead to reduced confidence in the presented evidence that requirements have been met and defeaters have been eliminated.

6.1.1 The Stepper Motor

The components of the *SM* have been adequately described earlier in this report. For purposes of the confidence map, we treat the *SM* as a black box consisting of those components.

6.1.2 The Stepper-Motor Position Control System (SM_PCS)

We recap the salient aspects of the original design: The stepper-motor position control system (SM_PCS) is an interface between the system at large and the stepper motor itself. Its purpose is to take a command such as "open the valve 80%" and translate it into a sequence of commands to the stepper motor to accomplish that request. The SM_PCS receives a number between 0 (closed) and 100 (fully opened) via a direct memory interface. The SM_PCS only "knows" the position of the *SM* by keeping track of how the *SM* has been commanded to change position, assuming these commands have been executed completely.[5] The system that makes requests of the SM_PCS (the ECS) has indirect knowledge of the actual position of the *SM* through observing related parameters (e.g., engine speed, torque, and the like).



A1.1a
ECS requests % open for SM via direct memory. SM_PCS runs in a frame of length F=25ms. SM is commanded via number of steps to move in the frame and direction. Maximum number of steps commanded in a frame is a precalculated constant S=15.
A
C1.1
The SM_PCS accurately positions the SM to the position requested by the ECS as quickly as possible but no longer
Cx1.1a
Accurately: to the nearest digital step corresponding to the analog % open requested by the ECS. As quickly as possible: without the need to complete any currently executing command
R2.1
Unless there is a malfunction in the SM
R2.2
Unless the SM is incapable of reaching the requested position within time T-F no matter how commanded
R2.3
Unless the SM_PCS does not know the true position of the SM
R2.4
Unless the SM_PCS doesn't issue the appropriate commands to the SM
IR2.5
If the SM doesn't malfunction and is fast enough when optimally controlled, and the SM_PCS can keep track of where the SM is and issue proper commands, then the SM will be positioned properly within time T

Figure 28: The Top-Level Confidence Map for the Stepper Motor

Figure 28 shows the top level of a stepper-motor confidence map for the architecture described above. The claim C1.1 reflects SMS-Req-3, SMS-Req-5, and SMS-Req-6 (all on Page 18, others mentioned below are on subsequent pages), and the corresponding derived requirements on SM_PCS. The box (A1.1a) with an "A" states what is assumed about the architecture while the oval (Cx1.1a) provides context information about the claim C1.1. Relevant assumptions include:

- The input to the SM_PCS is how far to open the valve as a percentage (SMS-Req-1 a.k.a. SM_PCS-Req-1)
- The input to the SM_PCS is provided via a direct memory access
- The SM_PCS is scheduled and runs in a frame of length F=25ms (SM_PCS-Req-2)
- The *SM* receives commands consisting of the number of steps to move and the direction to move in. The commanded rate is a constant, S=15 steps per frame, calculated at the time the SM_PCS is designed, and based upon the specifications of the *SM* (SM_PCS-Req-4)

A further consideration of the design is that when a new position for the stepper motor is requested by the ECS, the SM_PCS should command the *SM* to the new position in as direct a manner as possible without the need for it to go to the previously requested position first (SMS-Req-6). If, when the SM_PCS considers the direction the *SM* is moving in, the new position is behind the current position, then the SM_PCS issues commands that cause the *SM* to move in the opposite direction. If the new position is ahead of the current position, but before the previously desired position, then the SM_PCS should stop issuing commands once the *SM* has reached the

[5] The larger system "knows" the current position of the fuel valve, but as mentioned before, we are only considering this particular subsystem.

newly desired position without regard to the previously desired position. Note that this idea is not captured by the claim C1.1, which only requires that the *SM* be able to reach the new position within time T, but it is captured by the context Cx1.1a. A poorly designed SM_PCS could issue sub-optimal commands to the *SM*, causing it to take time T to move to the new position even when it was adjacent to the old one.

In addition, the context (Cx1.1a) provides a definition of what it means to satisfy the word "accurately" in claim C1.1.

There are four rebutting defeaters, which say that claim C1.1 would not be true if

1. There is a malfunction in the stepper motor. This corresponds to the error source declarations of type *NoCommandSequence* and *NoSteps* reflecting actuator failure and stepper-motor failure.
2. The stepper motor is physically incapable of reaching the position commanded within time T-F.[6] This corresponds to the *SlowResponse* out propagation from *SM*, which can be traced back to several error sources.
3. The SM_PCS loses track of the position of the stepper motor. This corresponds to the *MissedStep* out propagation of *SM*, which can be traced back to originating in the actuator due to an incoming *EarlyDelivery* or *RateError*.
4. The SM_PCS does not issue the appropriate commands to the stepper motor. This corresponds to *StepCountOutOfRange*, *ResultingPositionOutOfRange*, and *SubtleValueError* in the position-change command sent to the actuator.

The inference rule (IR2.5) says (essentially) that if all of these defeaters can be eliminated then the claim C1.1 is valid (but of course it is still necessary to consider any possible undercutting defeaters that apply to the inference rule). In the following section we'll discuss R2.3 and how it can be eliminated. A discussion of other portions of the confidence map can be found in the Appendix.

[6] T-F rather than just T because the current frame might have to complete its execution before the SM can begin to respond to a new ECS request.

6.1.3 Eliminating Defeater R2.3

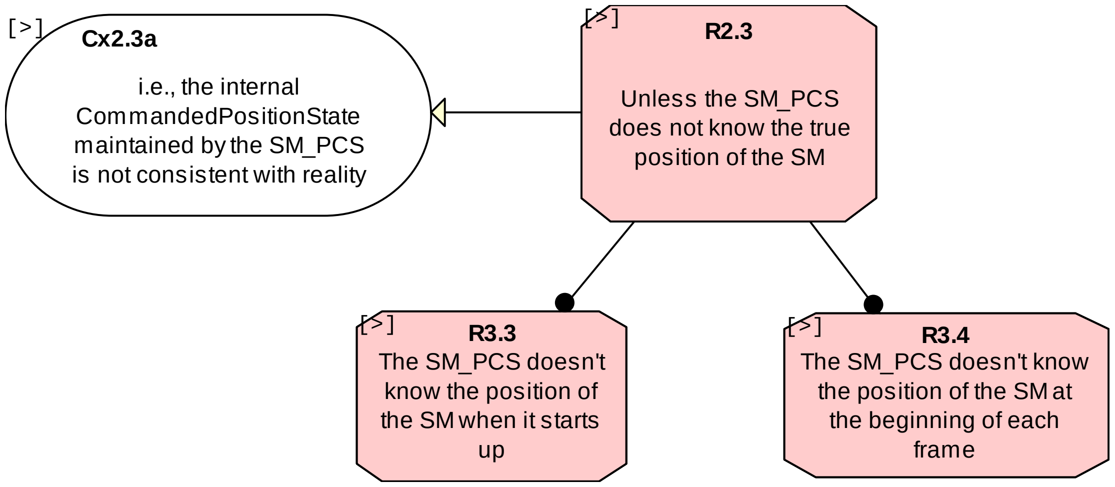
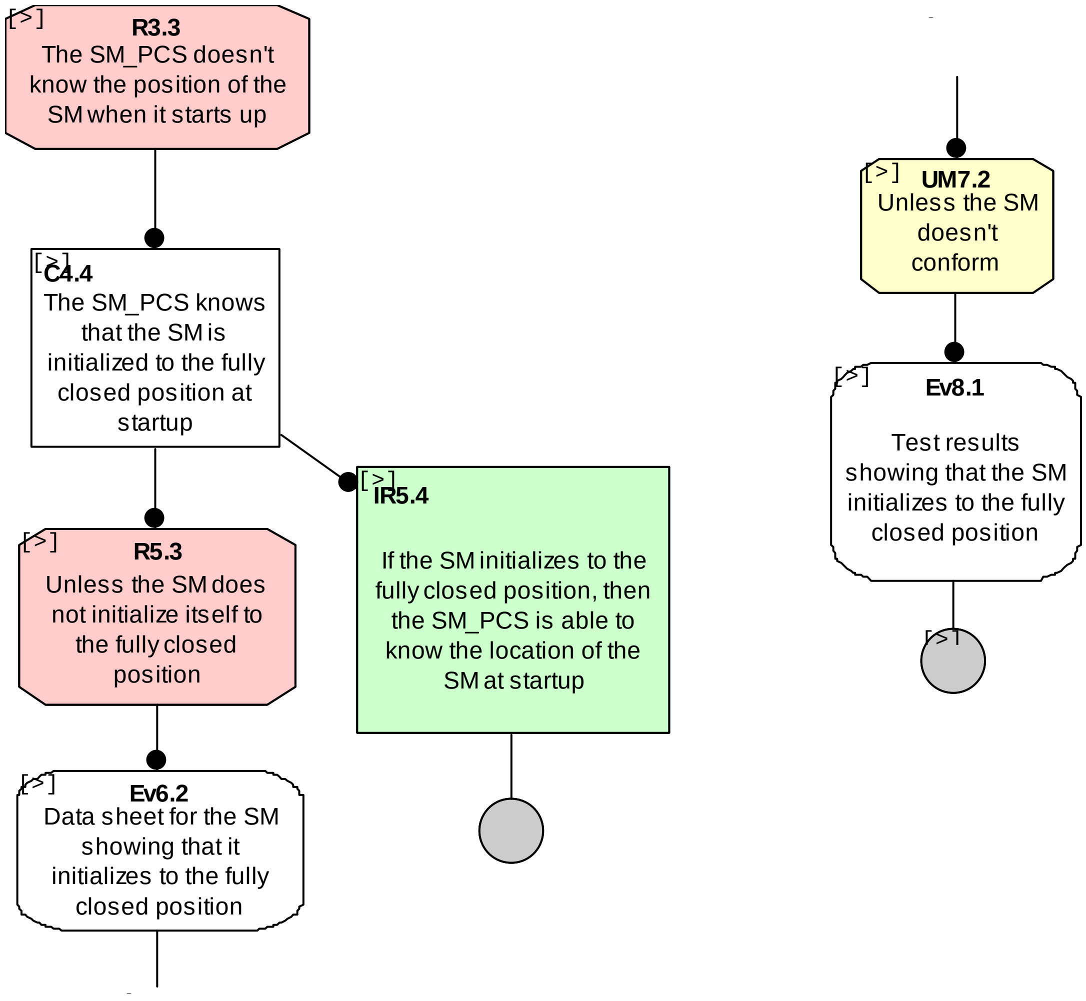

[>]
Cx2.3a
i.e., the internal CommandedPositionState maintained by the SM_PCS is not consistent with reality
[>]
R2.3
Unless the SM_PCS does not know the true position of the SM
[>]
R3.3
The SM_PCS doesn't know the position of the SM when it starts up
[>]
R3.4
The SM_PCS doesn't know the position of the SM at the beginning of each frame

Figure 29: Subdefeaters for Determining SM_PCS *Position*

The third of the top level defeaters, R2.3, considers the possibility that the SM_PCS's view of where the stepper motor is at any particular time is not consistent with reality. The SM_PCS's view of the position of the stepper motor is represented by the internal state *CommandedPositionState*. There are two cases to consider, shown by the sub-defeaters R3.3 and R3.4. The first is that it the SM_PCS's view of the position is inconsistent when the system starts up, and the second is that the system's view is inconsistent when it is invoked at the start of each subsequent frame. Note that the subdefeater notation is a way of making the argument more readable. The parent defeater (R2.3) summarizes the two subdefeaters (R3.3 and R3.4). Both subdefeaters must be eliminated in order to eliminate R2.3, and the argument would be just as valid without R2.3 (that is with R3.3 and R3.4) directly connecting to the parent claim (C1.1).

The defeater 2.3 is reflected in invariants on the desired position, commanded position, and actual position discussed in Section 4.3.

SM_PCS doesn't know the position of the SM when it starts up (R3.3)

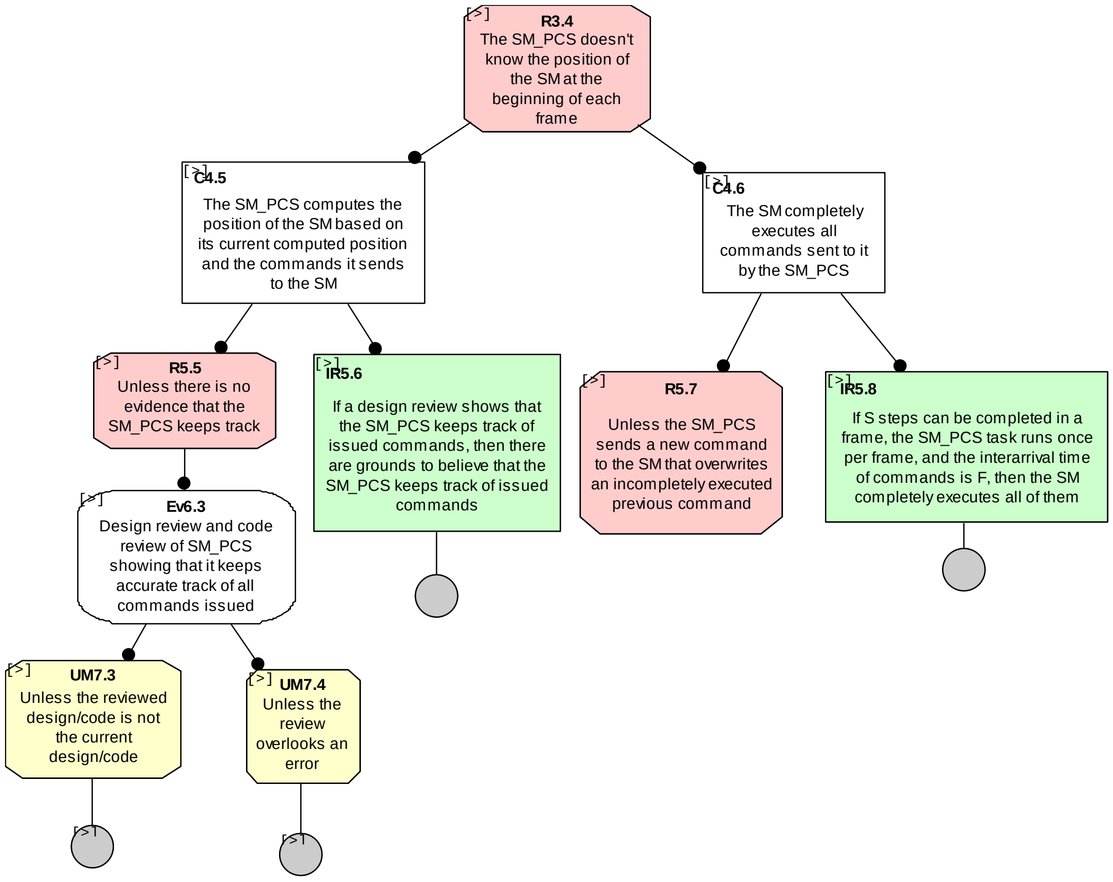


[>] R3.3
The SM_PCS doesn't know the position of the SM when it starts up
[>] C4.4
The SM_PCS knows that the SM is initialized to the fully closed position at startup
[>] IR5.4
If the SM initializes to the fully closed position, then the SM_PCS is able to know the location of the SM at startup
[>] R5.3
Unless the SM does not initialize itself to the fully closed position
[>] Ev6.2
Data sheet for the SM showing that it initializes to the fully closed position
[>] UM7.2
Unless the SM doesn't conform
[>] Ev8.1
Test results showing that the SM initializes to the fully closed position
[>]

Figure 30: Eliminating a Defeater Regarding SM Position at Startup

We consider R3.3 (the SM_PCS doesn't know the position of the *SM* when it starts up) first. This defeater is eliminated by the claim that the stepper motor initializes to the fully closed position (*Home*) when it starts up. This claim is, in turn, rebutted by defeater R5.3 concerning the possibility that there might be no evidence that this behavior is true. The argument eliminating this defeater is similar to that eliminating R2.2 above. This is a common argument pattern in this confidence map.

SM_PCS doesn't know the position of the SM at the beginning of subsequent frames



[>] R3.4 The SM_PCS doesn't know the position of the SM at the beginning of each frame
[>] C4.5 The SM_PCS computes the position of the SM based on its current computed position and the commands it sends to the SM
[>] C4.6 The SM completely executes all commands sent to it by the SM_PCS
[>] R5.5 Unless there is no evidence that the SM_PCS keeps track
[>] IR5.6 If a design review shows that the SM_PCS keeps track of issued commands, then there are grounds to believe that the SM_PCS keeps track of issued commands
[>] R5.7 Unless the SM_PCS sends a new command to the SM that overwrites an incompletely executed previous command
[>] IR5.8 If S steps can be completed in a frame, the SM_PCS task runs once per frame, and the interarrival time of commands is F, then the SM completely executes all of them
[>] Ev6.3 Design review and code review of SM_PCS showing that it keeps accurate track of all commands issued
[>] UM7.3 Unless the reviewed design/code is not the current design/code
[>] UM7.4 Unless the review overlooks an error
[>]
[>]

Figure 31: Eliminating a Defeater Regarding the Position of the SM at the Start of a Frame

The argument that the SM_PCS can tell the position at startup is quite straightforward. The argument that it knows the position at the start of subsequent frames is somewhat more complex. It requires that the SM_PCS keep track of commands that it has issued (number of steps, direction), and that the *SM* be able to successfully and completely execute all of the commands issued by the SM_PCS.

The first of these requirements is easily shown via a design/code review. Showing that the *SM* completely executes commands before receiving a subsequent command is the subject of the argument that begins with a rebutting defeater that worries about a command from the SM_PCS overwriting a previous command before the *SM* is able to complete it.

Note that defeater 5.7 corresponds to the *EarlyDeliveryImpact* error path specification for an incoming *EarlyDelivery* of a position-change command, which is mapped into a *MissingStepCommand* effect as outgoing propagation.

The SM_PCS overwrites a previously sent command

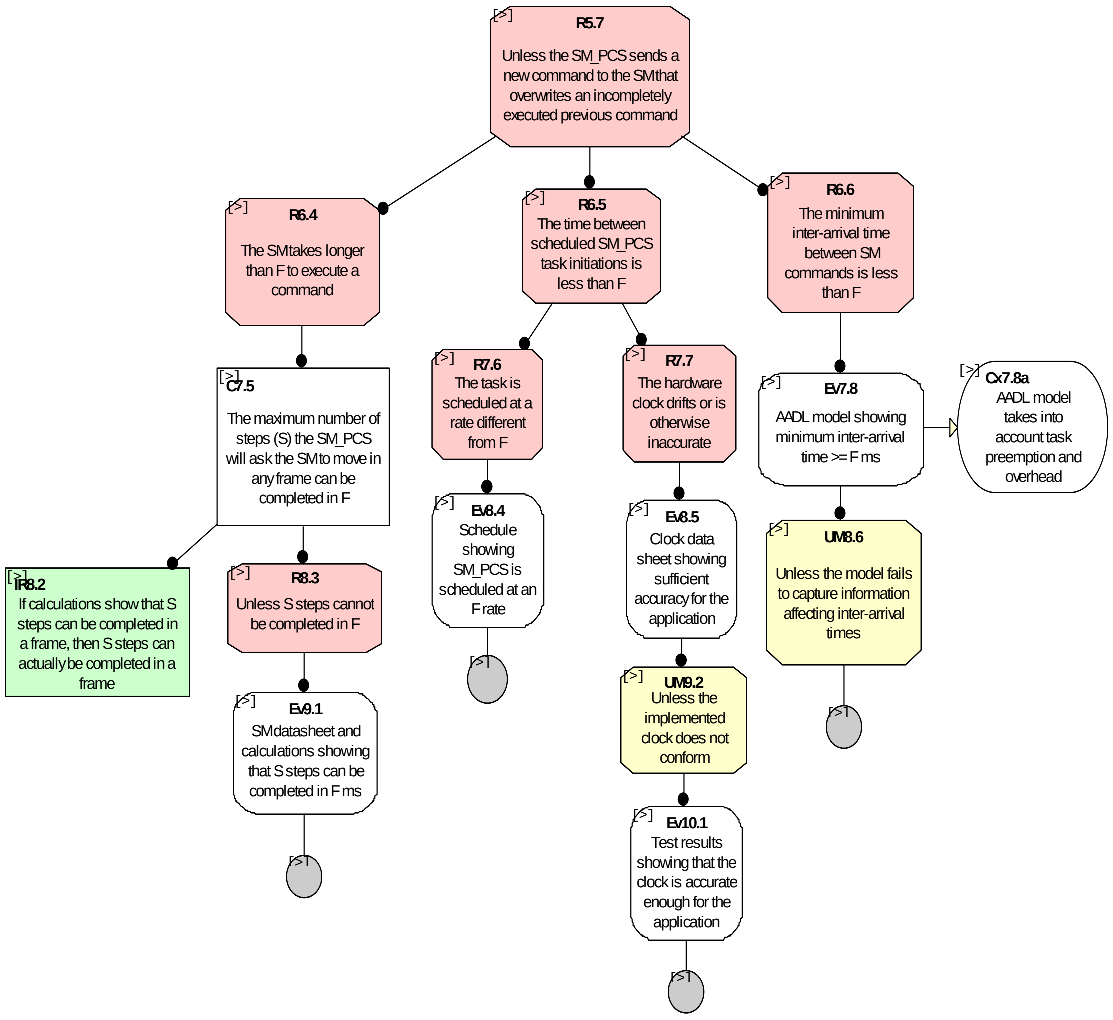

[>] R5.7
Unless the SM_PCS sends a new command to the SM that overwrites an incompletely executed previous command
[>] R6.4
The SM takes longer than F to execute a command
[>] R6.5
The time between scheduled SM_PCS task initiations is less than F
[>] R6.6
The minimum inter-arrival time between SM commands is less than F
[>] C7.5
The maximum number of steps (S) the SM_PCS will ask the SM to move in any frame can be completed in F
[>] R7.6
The task is scheduled at a rate different from F
[>] R7.7
The hardware clock drifts or is otherwise inaccurate
[>] Ev7.8
AADL model showing minimum inter-arrival time >= F ms
[>] Cx7.8a
AADL model takes into account task preemption and overhead
[>] IR8.2
If calculations show that S steps can be completed in a frame, then S steps can actually be completed in a frame
[>] R8.3
Unless S steps cannot be completed in F
[>] Ev8.4
Schedule showing SM_PCS is scheduled at an F rate
[>] Ev8.5
Clock data sheet showing sufficient accuracy for the application
[>] UM8.6
Unless the model fails to capture information affecting inter-arrival times
[>] Ev9.1
SM datasheet and calculations showing that S steps can be completed in F ms
[>] UM9.2
Unless the implemented clock does not conform
[>] Ev10.1
Test results showing that the clock is accurate enough for the application

Figure 32: Eliminating a Defeater Regarding the Overwriting of the Previous Command

There are three reasons why the *SM* might not have finished executing the previous command by the time the SM_PCS is ready to send a new one. These are indicated by rebutting defeaters R6.4, R6.5, and R6.6. The first of these is that the *SM* may take longer than the frame length to execute the number of steps previously commanded by the *SM*. The argument eliminating this defeater says that the SM_PCS will never ask for more than S steps in a frame and that the value of S was calculated (using information from the datasheet for the *SM*) to ensure that that number of steps can be completed.

The second defeater (R6.5) concerns the SM_PCS running faster than every F ms. This could be for one of two reasons. Either it is scheduled more than every F ms (R7.6) or there is a clock inaccuracy (R7.7). An inspection of the schedule should be sufficient to eliminate R7.6, and the datasheet and tests should be sufficient to eliminate R7.7.

Note that defeaters R6.4 and 6.5 correspond to rate errors in the form of *HighRate* and *LowRate*.

The third reason why the *SM* might not be finished executing the previous command by the time the SM_PCS is ready to issue a new one is that it (the SM_PCS) did not give the *SM* at least F ms to execute the previous command. This would occur if the SM_PCS did not run at the same point in every frame (for instance late in one frame and on time in the next), resulting in an inter-arrival time of commands to the *SM* of less than F. This requires more analysis than a simple inspection of the schedule and the argument above suggests analysis using an AADL model to take into account preemption effects of other computations sharing the processor on which the SM_PCS task is running.

Note that defeater R6.6 corresponds to the potential of *MissingStepCommand* due to timing or rate errors quantified by the inter-arrival time variation and step miss bound discussed in Section 5.3 and Appendix B.

6.1.4 Using the Confidence Map to Allocate Assurance Resources

The best way to use the confidence map to determine how to allocate assurance resources is to begin at the leaves of the confidence map. The leaves show what ultimately has to be true for the top-level claim to be true. Looking at the details of the complete confidence map (including those parts discussed in the Appendix), under this design, the items that must be checked to have confidence in the claim that "the SM_PCS will accurately position the *SM* to the position most recently requested by the ECS within time T" include

- Does the *SM* have sufficient reliability for the application?
- Is the *SM* physically capable of moving from an arbitrary position to another arbitrary position within time T?
- Does the *SM* initialize to the fully closed position at startup?
- Does the SM_PCS keep track of all commands it issues to the *SM*?
- Is the *SM* always able to move the maximum number of steps requested by the SM_PCS in a single frame within that frame?
- Is the SM_PCS scheduled at a frame rate of F?
- Is the system clock accurate enough for the application?
- Is the minimum inter-arrival time for commands to the SM less than F?
- Will the SM_PCS always issue the correct command (direction and number of steps)?
- Does the compiler compile the code correctly? Does the hardware execute the compiled code correctly? Is it certain that the execution environment will not interfere with correct execution of the code?
- Does the link between the SM_PCS and the *SM* have sufficient reliability for the application?

If all of the above points can be shown, then there should be high confidence that the SM_PCS and *SM* will be able to meet the claim C1.1.

6.2 Confidence in the Position-Commanded ("Alternate") Design for the Stepper Motor

During the course of this work we considered the complexity of the above design of the SM_PCS and wondered if it could be simplified to make achieving a high level of confidence an "easier" undertaking. "Easier" can mean several different things such as

- requiring less evidence to eliminate the set of defeaters
- requiring less time to eliminate a set of defeaters
- requiring less money to eliminate a set of defeaters

At this point in our work, determining this is somewhat subjective. In future work we hope to be able to provide a more objective measure.

As detailed in Section 4.4.3 the alternative design that we considered in some detail requires that the *SM* be implemented somewhat differently than it is currently.[7] Here is a description of that alternative *SM*. Aspects of the original design that are no longer necessary have been struck out. Additions are shown in red.

> *For our purposes the stepper motor is a black box that controls a valve. The valve can be set at N discrete positions from 0 (fully closed) to N (fully opened.) ~~The control system for the stepper motor takes three parameters set via a data link: s (number of steps to take), d (direction to move), and r (rate—how fast to move). Note that the stepper motor is not commanded to move to a particular position—it is only commanded to move a certain number of steps from its current position.~~ The stepper motor takes a single parameter, the desired position, via a data link and immediately begins to move to that position. If the position requested is changed then the stepper motor begins to move to that position instead.*
>
> *The SM does not provide feedback on the actual position of the stepper motor to the software using it. The software must infer it, either from observation of the results of opening or closing the valve, or by keeping track of the cumulative effect of commands it issued to the stepper motor. ~~When the stepper motor is initialized it homes to the fully closed position.~~*

As you can see, this is a relatively small change to the logic of the stepper motor.[8] It is interesting to see how this small change affects the confidence map.

6.2.1 Top Level

The claim C1.1 is identical in this map to that of the original map. The context Cx1.1a is also identical. The assumption (A1.1a) reflects the change in *SM* design.

[7] The SM is being treated as an OEM product in this report. In such a case, it may not be possible to get the manufacturer to make such a design change. Nevertheless, this is an interesting exercise.

[8] In the original design, the stepper motor would decrement a counter after each step and when it reached zero, stop moving. In the revised design the stepper motor would decrement or increment a position variable after each step and compare it to the desired position. When the comparison was equal it would stop moving.

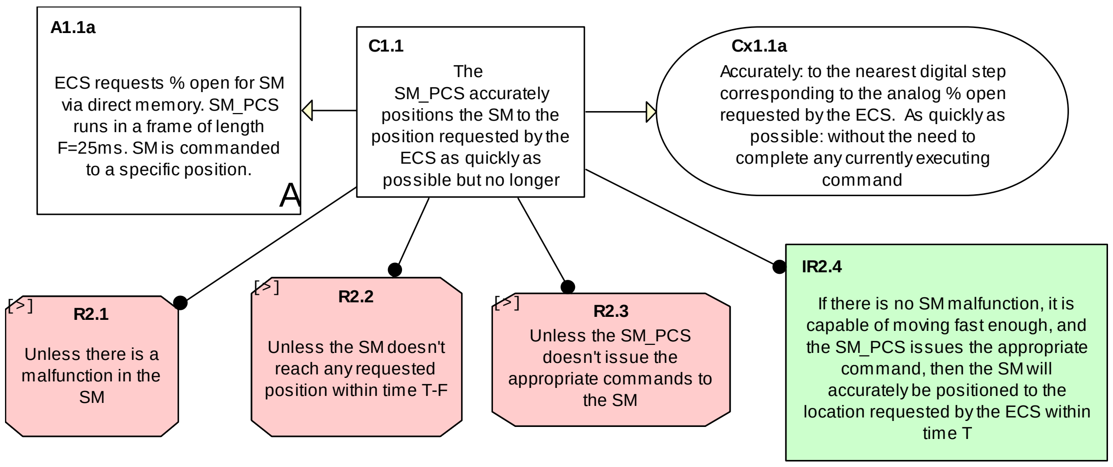

A1.1a
ECS requests % open for SM via direct memory. SM_PCS runs in a frame of length F=25ms. SM is commanded to a specific position.
A
C1.1
The SM_PCS accurately positions the SM to the position requested by the ECS as quickly as possible but no longer
Cx1.1a
Accurately: to the nearest digital step corresponding to the analog % open requested by the ECS. As quickly as possible: without the need to complete any currently executing command
[>]
R2.1
Unless there is a malfunction in the SM
[>]
R2.2
Unless the SM doesn't reach any requested position within time T-F
[>]
R2.3
Unless the SM_PCS doesn't issue the appropriate commands to the SM
IR2.4
If there is no SM malfunction, it is capable of moving fast enough, and the SM_PCS issues the appropriate command, then the SM will accurately be positioned to the location requested by the ECS within time T

Figure 33: Top-Level Claim of the Alternative Design

The set of defeaters has also changed. Notably missing is the top-level defeater discussed in detail above that requires the SM_PCS to keep track of the position of the *SM* (the *SM* knows its current position and how to get to the desired position). The remaining defeaters are similar to those in the original map and are shown in the Appendix.

6.2.2 Using the Confidence Map to Allocate Assurance Resources

Again beginning at the leaves of the confidence map under this design, the issues that must be checked to generate confidence in the claim that "the SM_PCS will accurately position the *SM* to the position most recently requested by the ECS within time T" include

- Does the *SM* have sufficient reliability for the application?
- Is the *SM* physically capable of moving from an arbitrary position to another arbitrary position within time T-F?
- Does the SM_PCS issue a command to the *SM* at the start of each frame?
- Will the SM_PCS always issue a command requesting that the *SM* move to the correct position?
- Does the compiler compile the code correctly? Does the hardware execute the compiled code correctly? Is it certain that the execution environment will not interfere with correct execution of the code?
- Does the link between the SM_PCS and the *SM* have sufficient reliability for the application?

If all of the above questions can be answered positively, then there should be high confidence that the SM_PCS and *SM* will be able to meet the claim C1.1. This is a *much* shorter list than for the original design alternative.

6.3 Discussion

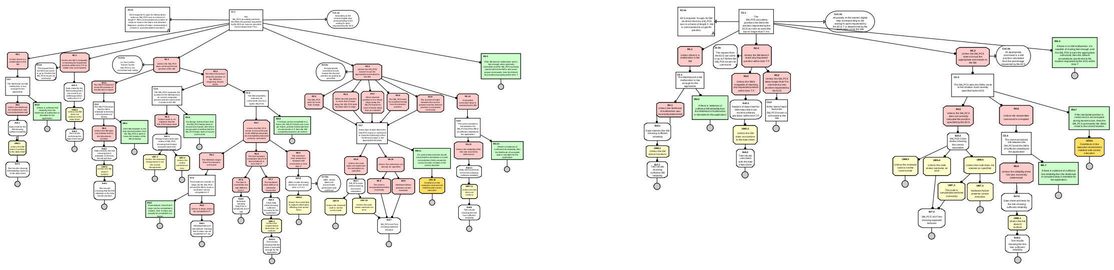


Figure 34: The Two Confidence Maps

Figure 34 shows the two complete confidence maps side by side and at approximately the same scale. They are not meant to be read, but instead are presented to give a gestalt of the difference in complexity of achieving confidence in each of the designs. The one for the alternate design has been purposely laid out with a "hole" in the middle to show where the original design has an original, complex, defeater.

The evidence nodes near the leaves of the confidence map identify V&V activities that can help eliminate key defeaters. Any defeaters that undermine the evidence nodes suggest necessary characteristics of the evidence (e.g., that the specification sheet being used as evidence be for the actual device.)

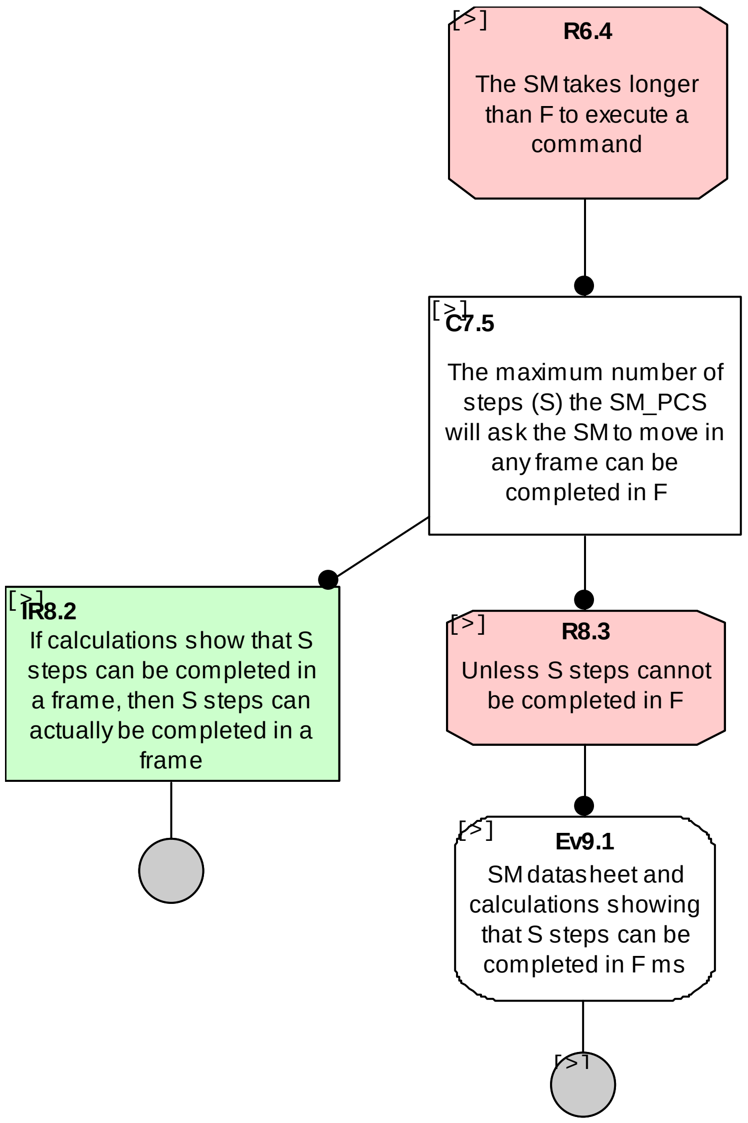

[>]
R6.4
The SM takes longer than F to execute a command
[>]
C7.5
The maximum number of steps (S) the SM_PCS will ask the SM to move in any frame can be completed in F
[>]
IR8.2
If calculations show that S steps can be completed in a frame, then S steps can actually be completed in a frame
[>]
R8.3
Unless S steps cannot be completed in F
[>]
Ev9.1
SM datasheet and calculations showing that S steps can be completed in F ms
[>]

Figure 35: S Steps Cannot Be Completed in Time F

For example consider Figure 35, a small portion of Figure 32. The circled defeater R8.3 is concerned with the ability of the *SM* to execute the maximum number of steps S within the frame length F. The defeater would be eliminated by the datasheet for the *SM* that supplies information used for calculating the value of S. Note that this calculation is crucial and should take into account worst-case performance of the *SM* or there is a risk that the defeater will not be eliminated. In analysis of the actual system design, R8.3 could not be eliminated, since, under worst-case conditions, execution of 15 steps could take 0.98ms longer than the frame length. The uneliminated defeater represents a residual doubt that the SMS works correctly.

6.3.1 The Effort Needed to Obtain Needed Evidence

The design of the SMS can have a big impact on the difficulty of showing that its defeaters have been eliminated. Figure 36 summarizes the evidence needed to eliminate key defeaters in each of the two designs considered in this section. The color coding in this figure is subjective. Red represents evidence that seems as if it would be relatively difficult to obtain. Green represents evidence that it seems relatively easy to obtain. Eliminating some of the defeaters will take more work than eliminating others, but the fact that the confidence map highlights them is important. It allows the designer to identify points at which things can possibly go wrong and to intelligently make decisions as to which to expend the most resources on elimination.

The alternate (position-commanded) design (at least from the point of view of the SM_PCS) has a significantly shorter list of defeaters to eliminate, and some of the defeaters in the original design will (subjectively) take more effort to remove than any in the alternative design. In no case should it take more work to eliminate a defeater in the position-commanded design than in the original design.

Evidence	**Original**	**Alternate**
	Subjective Difficulty	
Needed Evidence	**Original**	**Alternate**
The SM is reliable enough for the application	UM6.1	UM6.1
The SM can move between arbitrary points within T-F ms.	UM4.3	UM5.2
The SM homes to the closed position at start up	UM7.2	
The SM_PCS accurately knows where the SM is at the start of each frame.	R5.5	
The SM is capable of always moving S steps within a frame	R8.3	
The SM_PCS is scheduled at rate F	R7.6	
The SM_PCS commands a position requested by the ECS within F		R3.3
The actual clock accuracy conforms to calculated clock accuracy	UM9.2	
The AADL model shows the minimal interarrival time of the execution of SM_PCS to be >= F.	UM8.6	
The SM_PCS always calculates the proper number of steps and direction to move at each invocation	R5.9	
The SM_PCS properly transforms the ECS commanded position to a position for the SM		R4.5
The SM_PCS code executes properly	R5.10	UM6.5
The SM runs in an appropriate execution environment	UC6.10	UC5.5
The data link between the SM_PCS and the SM is of sufficient reliability for the application	UM7.11	UM8.1

Figure 36: Evidence Required for the Two Designs

As mentioned before, at this point in our research we treat the measurement of the relative effort necessary to show confidence between alternate designs subjectively. Effort can be affected either by having to collect more evidence for one design than for the other or by having to work harder to collect similar or identical evidence in one design than for the other.

In our two example designs, the original requires that 12 pieces of evidence be collected while the alternative only requires seven. Furthermore the seven in the alterative design all have counterparts (identical or similar) in the original design. So, all else being equal, this would lead to the conclusion that achieving confidence in the original design will require significantly more effort than achieving it in the alternate design. The only way this conclusion would not follow would be if obtaining the identical (or similar) evidence for the alternate design was significantly more difficult than obtaining it for the original design. From reviewing the lists in the previous section it seems clear that this would not be the case.

7 Conclusions

The purpose of this case study was to show how architecture fault modeling and analysis can be used to diagnose a time-sensitive design error encountered in a control system and to investigate whether proposed changes to the system address the problem. The analytical approach demonstrates that such errors that are hard to test for can be discovered and corrected early in the life-cycle, thereby reducing rework cost. The case study shows that by combining the analytical approach with confidence maps we can present a structured argument that system requirements have been met and problems in the design have been addressed adequately—increasing our confidence in the system quality.

The case study system is an SMS that is part of an engine control system. Its design had been verified with SCADE without discovering—until system integration and operational testing—the potential for missed step commanding due to variation in command inter-arrival time. Two possible solutions to the problem had been proposed.

In this case study we have combined architecture and fault modeling and analysis with confidence arguments—drawing on a framework for software assurance and software quality improvement [Feiler 2012]. AADL, an SAE architecture description language standard with well-defined execution and communication semantics suited for embedded software systems, has been used to iteratively specify the SMS architecture and its requirements. We have captured the original SMS architecture design, both proposed corrections to the design, and a design alternative. This design alternative eliminates error sources contributed by the position control system component SMS without significantly increasing the complexity of the actuator by commanding the actuator with the desired position instead of a sequence of position-change commands in the original design.

The resulting architecture model was then annotated with fault model specifications expressed in the EMV2 annex standard to AADL. We diagnosed the timing-related problem by following a safety analysis approach that uses the EMV2 fault propagation ontology to systematically identify hazards and their impact on the system. This systematic analysis allowed us to identify not one, but two time-sensitive sources for the missed step problem. The analysis also allowed us to confirm that an SMS requirement is to immediately respond to new position commands when previous commands are in progress. It also allowed us to assess the resilience of the different SMS designs to transient data corruption by a direct access memory. It illustrates the consequences of the architecture design decision to command the actuator by a sequence of position-change commands in terms of time sensitivity and the assumption of guaranteed delivery and execution of all commands. The time-sensitive nature inherent in the original design was not only shown analytically, but also by an auto-generated prototype implementation in Java.

The confidence map notation was used to present structured confidence arguments for the SMS architectures. Requirements become clearly traceable claims, organized to reflect requirements decomposition. Error sources representing potential hazards in the architecture fault model become defeaters with derived safety requirements as claims, whose evidence eliminates the defeater. Assumptions in a contract model between components, including the absence of fault propagation, are explicitly recorded. Arguments for the sufficiency of satisfying subclaims and eliminating defeaters to satisfy a claim are captured in inference rules. Eliminative induction systematically establishes confidence that presented evidence addresses defeaters and claims. This evidence can be in the form of traditional activities such as design review, code review, and testing, as well as analytical results based on predictive analysis of the architecture model. Development of such a confidence map for both the original SMS architecture and the alternative design allowed us to illustrate the difference in effort necessary to establish confidence in qualification and certification evidence.

Appendix A The SMS Data Model

An instance of the AADL data component type *SM_Position* represents the position of the stepper motor. The data type has been defined as two variants, expressed as the data component implementation *SM_Position.PercentOpen* and *SM_Position.Steps*. The declarations, shown in Figure 37, include properties to indicate the data representation of the value (*Fixed* and *Integer*), acceptable range of values, and the measurement unit as stepper-motor steps (*PercentOpen* and *Steps*).

The specification uses the Data Model Annex standard to characterize details of the data type. The role of this annex is to provide properties and guidance on how to map relevant information from data models expressed in other modeling notations, for example, UML or source code, into an architecture model expressed in AADL.

The SM_PCS commands SM_ACT to have the stepper motor execute a specified number of steps. The command takes three parameters: the direction in which to move the stepper motor (Direction: Open, Close), the number of steps to be executed (StepCount), and the rate (StepRate) at which to execute the steps in terms of steps per frame (SPF). The design assumes that the stepper motor can completely execute the requested number of steps in one frame duration; consequently, the requested step count cannot exceed the number of steps that can be executed in one frame (MaxStepCount). The Direction parameter indicates whether to move towards the maximum position (Open) or towards zero (Close).

An instance of the AADL data component type *SM_Position_Change* represents the position change being commanded, that is, the step count, step rate, and direction, and their acceptable value ranges. As shown in Figure 37, we use the subcomponent declarations in the AADL data component implementation to explicitly represent the fields of the position-change data record.

```
data SM_Position
end SM_Position;

data implementation SM_Position.PercentOpen
properties
  Data_Model::Data_Representation => Fixed;
  Data_Model::Measurement_Unit => "Percent";
  Data_Model::Integer_Range => 0 .. PCSProperties::MaxPercent;
end SM_Position.PercentOpen;

data implementation SM_Position.Steps
properties
  Data_Model::Data_Representation => Integer;
  Data_Model::Measurement_Unit => "Steps";
  Data_Model::Integer_Range => 0 .. PCSProperties::MaxPosition;
end SM_Position.Steps;

data SM_Position_Change
end SM_Position_Change;

data implementation SM_Position_Change.DataRecord
subcomponents
  StepCount: data {
    Data_Model::Data_Representation => Integer;
    Data_Model::Measurement_Unit => "Steps";
    Data_Model::Integer_Range => 0 .. PCSProperties::MaxStepCount;
  };
  StepDirection: data {
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("Open","Close");
  };
  StepRate: data {
    Data_Model::Data_Representation => Integer;
    Data_Model::Measurement_Unit => "Steps";
    Data_Model::Integer_Range =>
      PCSProperties::MaxStepCount .. PCSProperties::MaxStepCount;
  };
end SM_Position_Change.DataRecord;
```

Figure 37: SM_PCS *Data Model in AADL*

Appendix B Variation in Inter-Arrival Time

The inter-arrival time variation is determined by the variation in the time at which the command is sent by SM_PCS and variation in communication time, that is,

$\texttt{Delta(Interarrival) = Delta(Send}_{SM\_PCS}\texttt{)+ Delta(Comm)}$

For this analysis we assume that the command is sent at completion time of SM_PCS, that is, `Delta(Send) = Delta(CT)`. Completion time for SM_PCS is determined by its execution time on the processor, by preemption time from higher priority threads—in our example the *HM* thread, and any blocking time on shared logical resources—none in our example. Variation of completion time is

$\texttt{Delta(CT}_{SM\_PCS}\texttt{) = Max(CT}_{SM\_PCS}\texttt{) – Min(CT}_{SM\_PCS}\texttt{).}$

Scheduling analysis such as Rate Monotonic Analysis (RMA) determines whether a task meets its deadline and in the process calculates worst-case completion time. In our example the task set is simple, which leads to a simple formula for calculating the completion time variation from the variation of execution time for the two tasks sharing the ECU.

Maximum completion time variation for SM_PCS is calculated as follows, using ET as execution time and PT as preemption time:

$\texttt{Delta(CT}_{SM\_PCS}\texttt{) = Delta(ET}_{SM\_PCS}\texttt{) + Delta(PT}_{SM\_HM}\texttt{)}$

The maximum preemption time variation is calculated as follows:

$\texttt{Delta(PT}_{SM\_HM}\texttt{) = k * Delta(ET}_{HM}\texttt{) + Max(ET}_{SM\_HM}\texttt{)}$

$\texttt{with k = Ceiling(Max(ET}_{PCS}\texttt{)/P}_{SM\_HM}\texttt{)}$

As the formula shows, SM_PCS is preempted by SM_HM at least once and can be preempted multiple times. Furthermore, SM_HM may preempt SM_PCS one additional time under maximum ET conditions compared to minimum ET conditions. Thus, the preemption time variation is not just a multiple of SM_HM execution time variation but also includes the maximum execution time.

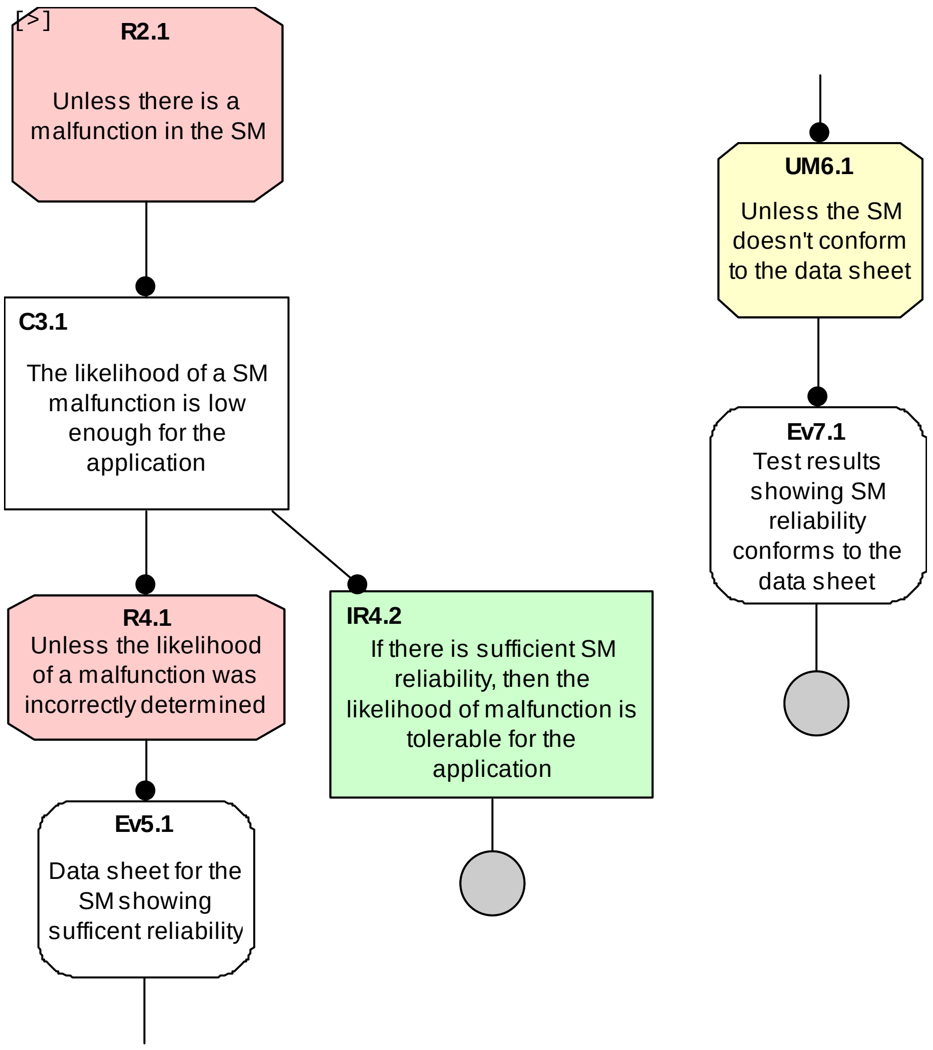
Based on actual or hypothetical data for the execution time variation and communication time variation, we can explore under what conditions this variation exceeds the maximum acceptable inter-arrival time variation. Note that the logical SM_PCS thread may execute other control functions. Thus, the worst-case variation in Min(ET_{SM_PCS}) and Max(ET_{SM_PCS}) is determined by the sum of execution times of all these functions.

Appendix C Confidence Maps

C.1 Other Defeaters from the Original Design

C.1.1 Stepper-Motor Malfunction Defeater (Defeater R2.1)

Clearly the SM_PCS cannot be sure of accurately positioning the stepper motor if the motor is malfunctioning. If defeater R2.1 is not eliminated then we have less confidence in claim C1.1. If, in fact, defeater R2.1 is shown to be true, then claim C1.1 is false. So our goal here is to build up confidence that R2.1 has been eliminated. Figure 38 shows the confidence map arguing the elimination. (It is split for display purposes; think of UM6.1 as being under Ev5.1.)



[>]
R2.1
Unless there is a malfunction in the SM
UM6.1
Unless the SM doesn't conform to the data sheet
C3.1
The likelihood of a SM malfunction is low enough for the application
Ev7.1
Test results showing SM reliability conforms to the data sheet
R4.1
Unless the likelihood of a malfunction was incorrectly determined
IR4.2
If there is sufficient SM reliability, then the likelihood of malfunction is tolerable for the application
Ev5.1
Data sheet for the SM showing sufficent reliability

Figure 38: Eliminating a Defeater Regarding SM Malfunction

Defeater R2.1 is eliminated by claim C3.1, which says that, while there may be a malfunction in the *SM*, the likelihood of this happening is tolerable. This is true unless the likelihood (reliability number if you will) has been incorrectly determined (R4.1). The manufacturer's data sheet was used as evidence of *SM* reliability (Ev5.1). That evidence can be undermined (UM6.1) if the actual *SM* doesn't conform to the *SM* that was used to create the data sheet. Of course independent tests (Ev7.1) can eliminate that defeater. We choose not to go any further (indicated by the grey circle) but we could: for instance, we could undermine Ev7.1 by suggesting that the testing was too limited to justify the reliability conclusion and therefore that the test results are irrelevant.

Note that the justification for eliminating the defeater R2.1 is probabilistic in nature. The Error Model annotations can be augmented with such probabilistic data to document the data sheet assumptions and to support reliability and availability predictions of ECS based on the failure probabilities of its parts and external error sources.

C.1.2 Stepper-Motor Speed Defeater (R2.2)

The ability for the *SM* to reach an arbitrary point from an arbitrary point within the requested time depends on its design and the values of T and F. If T and F (and S, the rate at which the stepper motor moves, in the more general case) are inappropriately chosen with respect to this design, then it may not be physically possible for the *SM* to reach the desired position no matter how efficiently the SM_PCS issues commands to move it. (F comes into the calculation because the ECS can request a new position at any time, even immediately after the SM_PCS issues a command, and the SM_PCS won't run again until the next frame.)

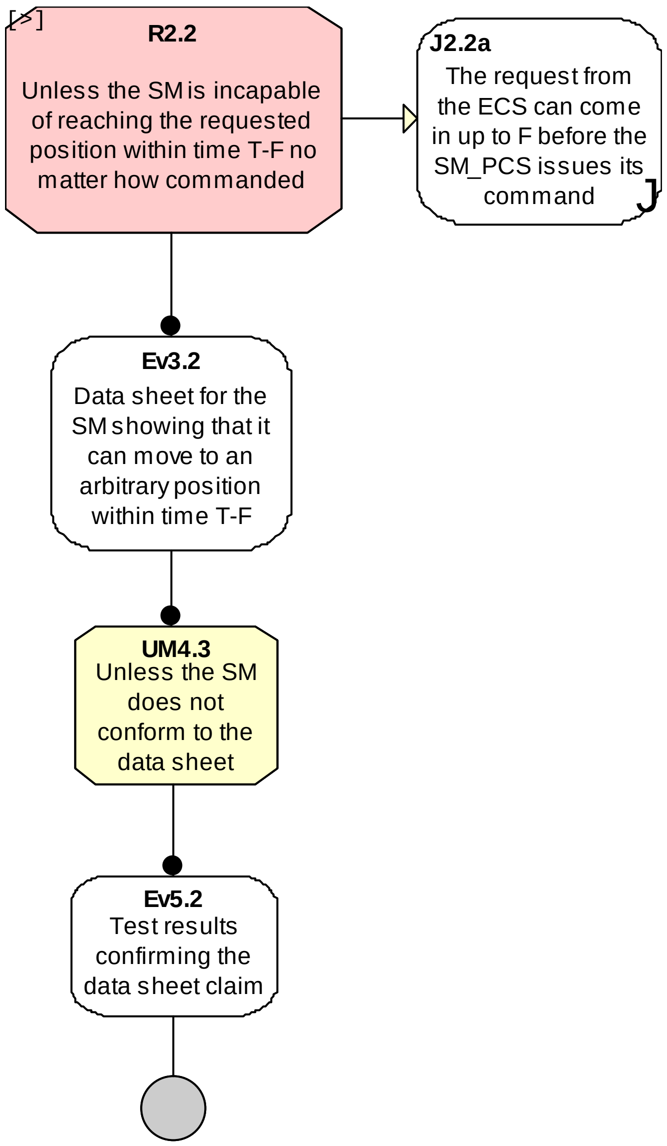

[>]
R2.2
Unless the SM is incapable of reaching the requested position within time T-F no matter how commanded
J2.2a
The request from the ECS can come in up to F before the SM_PCS issues its command
J
Ev3.2
Data sheet for the SM showing that it can move to an arbitrary position within time T-F
UM4.3
Unless the SM does not conform to the data sheet
Ev5.2
Test results confirming the data sheet claim

Figure 39: Eliminating a Defeater Regarding SM Speed

The argument eliminating defeater R2.2 is similar in structure to that eliminating R2.1 (regarding *SM* malfunction) and will not be discussed further here. Section 5.3.2 discusses the potential for the stepper motor operating at a lower than expected rate resulting in cumulative delay in completion time and argues that this delay is possibly unbounded.

C.1.3 The SM_PCS Does Not Issue Appropriate Commands (R2.4)

The fourth and final defeater of the top-level claim suggests that the SM_PCS may not issue an “appropriate command” to the *SM*. A command is appropriate if it moves the *SM* towards the desired position as quickly as possible (Cx2.4a).

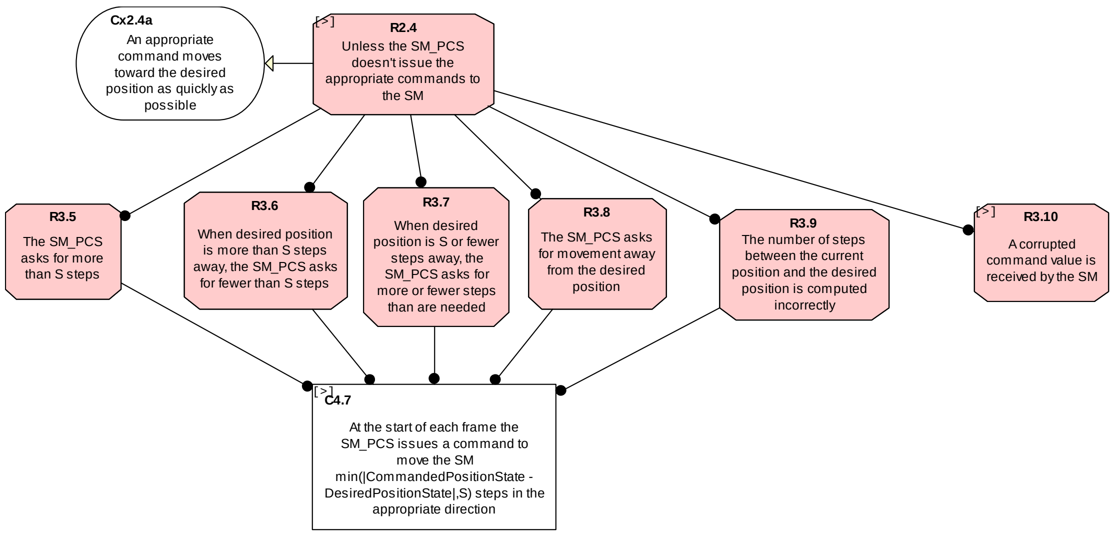

Cx2.4a
An appropriate command moves toward the desired position as quickly as possible
[>]
R2.4
Unless the SM_PCS doesn't issue the appropriate commands to the SM
R3.5
The SM_PCS asks for more than S steps
R3.6
When desired position is more than S steps away, the SM_PCS asks for fewer than S steps
R3.7
When desired position is S or fewer steps away, the SM_PCS asks for more or fewer steps than are needed
R3.8
The SM_PCS asks for movement away from the desired position
R3.9
The number of steps between the current position and the desired position is computed incorrectly
[>]
R3.10
A corrupted command value is received by the SM
[>]
C4.7
At the start of each frame the SM_PCS issues a command to move the SM min(|CommandedPositionState - DesiredPositionState|,S) steps in the appropriate direction

Figure 40: Eliminating a Defeater Regarding the SM_PCS *Issuing Appropriate Commands to the SM*

There are six sub-cases for this rebutting defeater (see the figure above), five of which are eliminated by the claim (C4.7) that at the start of each frame the SM_PCS issues the appropriate command. This claim is developed below. The sixth defeater (R3.10) deals with the situation where the command to the *SM* is corrupted. This claim is also developed below.

Note that defeater R3.5 corresponds to the possibility of a *StepCountOutOfRange* propagation. Defeaters R3.6 through 3.9 correspond to the possibility of a *ResultingPositionOutOfRange* or a *SubtleValueError* propagation. In the fault model specification we have indicated that these defeaters are expected to be eliminated.

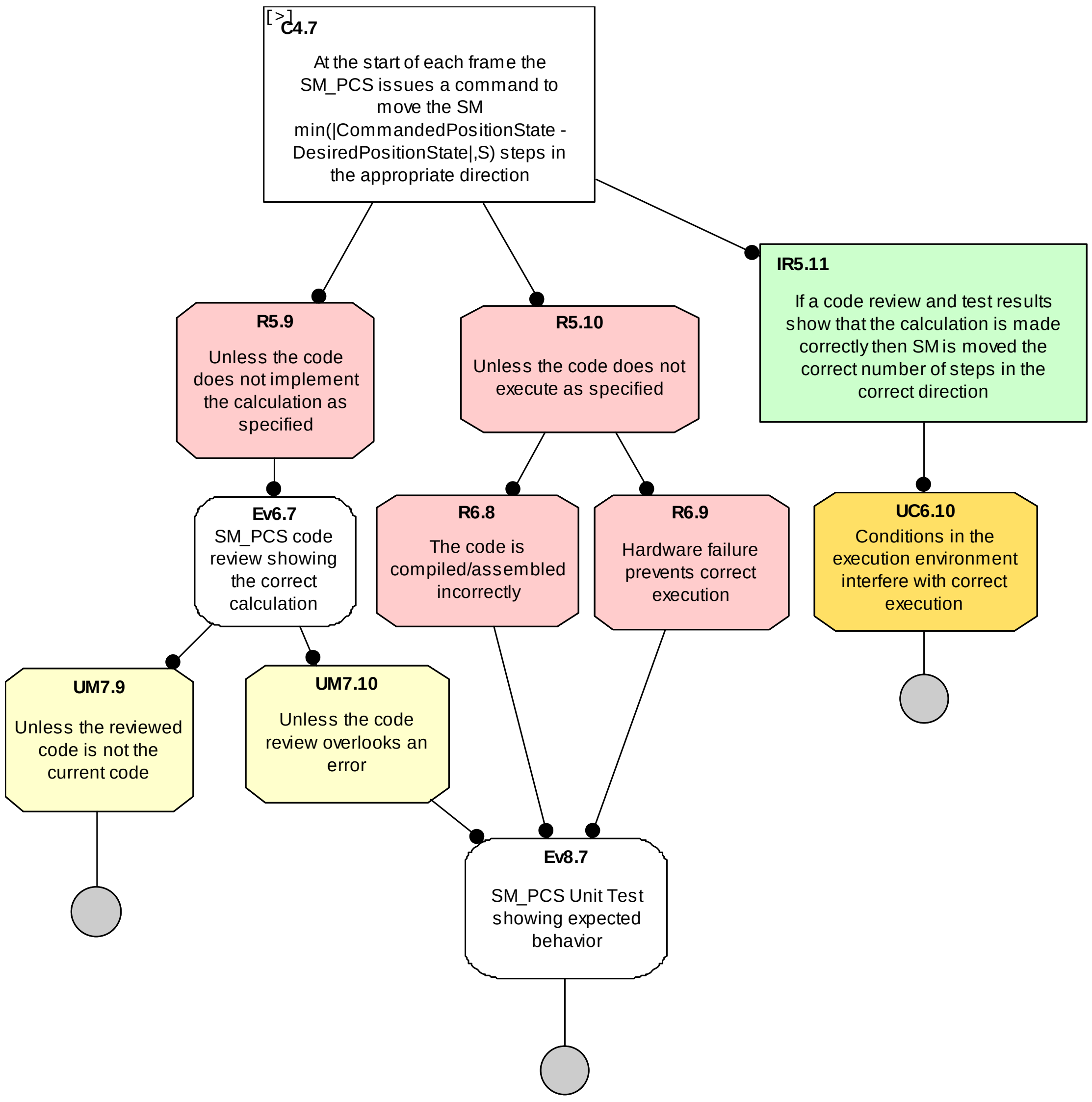
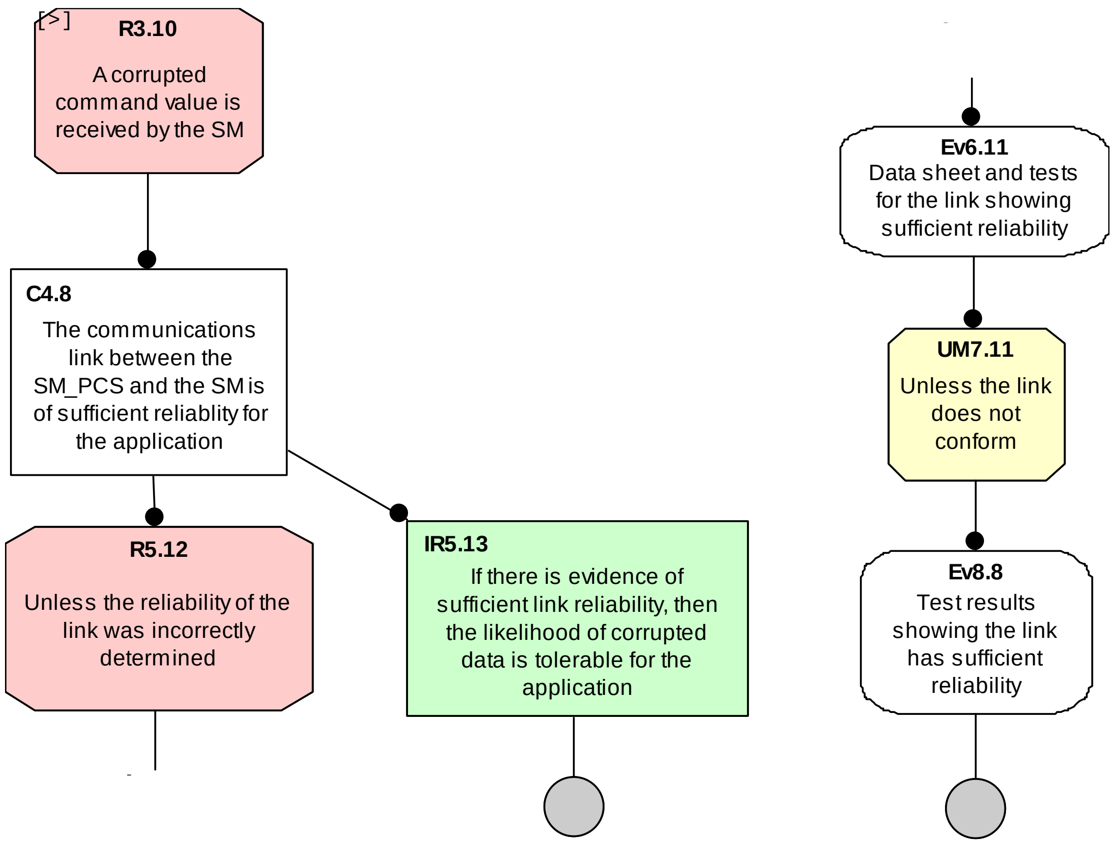

[>]
C4.7
At the start of each frame the SM_PCS issues a command to move the SM min(|CommandedPositionState - DesiredPositionState|,S) steps in the appropriate direction
R5.9
Unless the code does not implement the calculation as specified
R5.10
Unless the code does not execute as specified
IR5.11
If a code review and test results show that the calculation is made correctly then SM is moved the correct number of steps in the correct direction
Ev6.7
SM_PCS code review showing the correct calculation
R6.8
The code is compiled/assembled incorrectly
R6.9
Hardware failure prevents correct execution
UC6.10
Conditions in the execution environment interfere with correct execution
UM7.9
Unless the reviewed code is not the current code
UM7.10
Unless the code review overlooks an error
Ev8.7
SM_PCS Unit Test showing expected behavior

Figure 41: The Claim that the SM_PCS *Calculates the Command Correctly*

The claim that the SM_PCS issues the correct command is defeated if the code is not implemented as specified or the code does not execute as specified. The first can be dealt with by a code review. The second requires belief that the code is compiled correctly, that there is no hardware failure, and that there are no conditions in the execution environment that interfere with correct execution. The sub-defeaters are all eliminated by unit testing.

Note that based on the behavior specification of SM_PCS, we can utilize model checking techniques to verify that the algorithmic behavior is specified correctly. Similarly, code-level verification techniques can be used to ensure the implementation meets the specified computation requirements.



[>]
R3.10
A corrupted command value is received by the SM
C4.8
The communications link between the SM_PCS and the SM is of sufficient reliablity for the application
R5.12
Unless the reliability of the link was incorrectly determined
IR5.13
If there is evidence of sufficient link reliability, then the likelihood of corrupted data is tolerable for the application
Ev6.11
Data sheet and tests for the link showing sufficient reliability
UM7.11
Unless the link does not conform
Ev8.8
Test results showing the link has sufficient reliability

Figure 42: Eliminating a Defeater Regarding Data Corruption

Finally, there is the defeater dealing with corruption in the data transmitted between the SM_PCS and the *SM*. This is eliminated by a straightforward argument that the communications link between the two is of sufficient reliability[9] for the application.

This defeater corresponds to the *MessageCorruption* error source from the direct access memory. In the Error Model annotation we can reflect the probability of such a corruption or a message loss occurrence. In the fault model specification, we have also reflected the potential for tolerating transient corruption.

[9] Sufficient reliability for the SM will presumably vary depending on the ultimate application.

C.2 Other Defeaters from the Position-Commanded Design

C.2.1 Stepper-Motor Malfunction Defeater (Defeater R2.1)

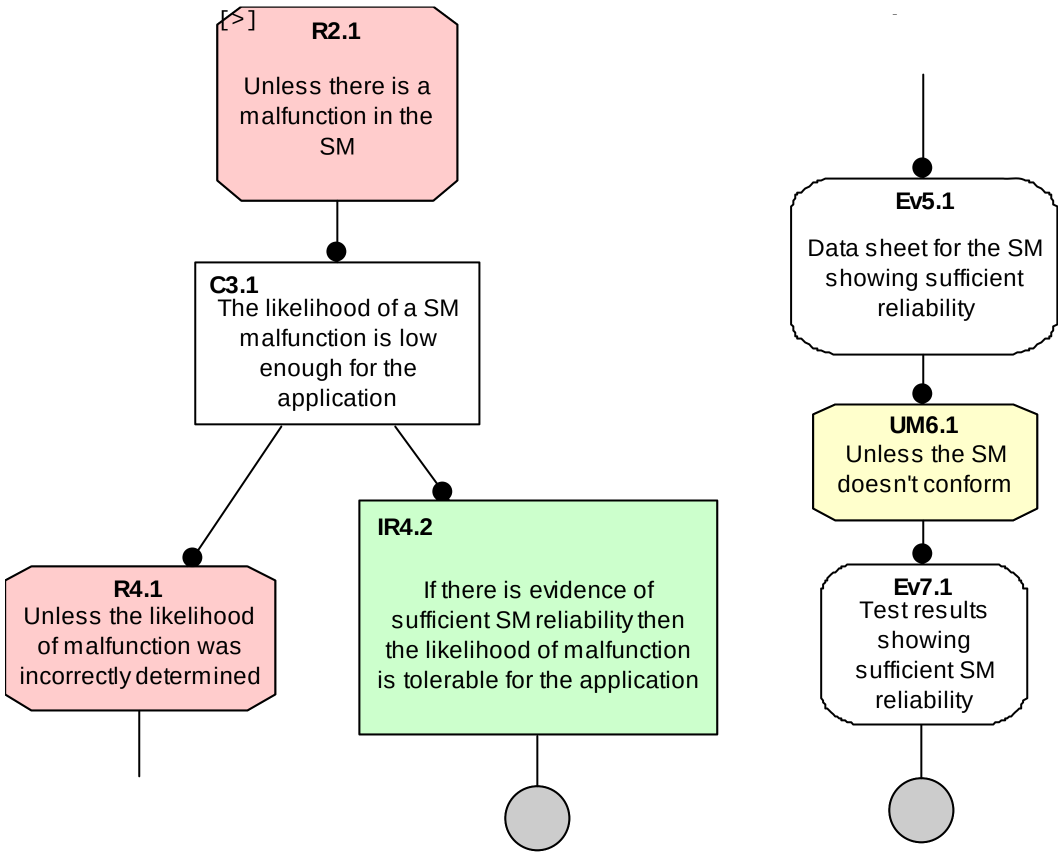

[>]
R2.1
Unless there is a malfunction in the SM
C3.1
The likelihood of a SM malfunction is low enough for the application
R4.1
Unless the likelihood of malfunction was incorrectly determined
IR4.2
If there is evidence of sufficient SM reliability then the likelihood of malfunction is tolerable for the application
Ev5.1
Data sheet for the SM showing sufficient reliability
UM6.1
Unless the SM doesn't conform
Ev7.1
Test results showing sufficient SM reliability

Figure 43: Eliminating a Defeater Regarding SM Malfunction

The confidence map for eliminating this defeater is essentially the same as for the more complex design.

C.2.2 Stepper-Motor Speed Defeater (R2.2)

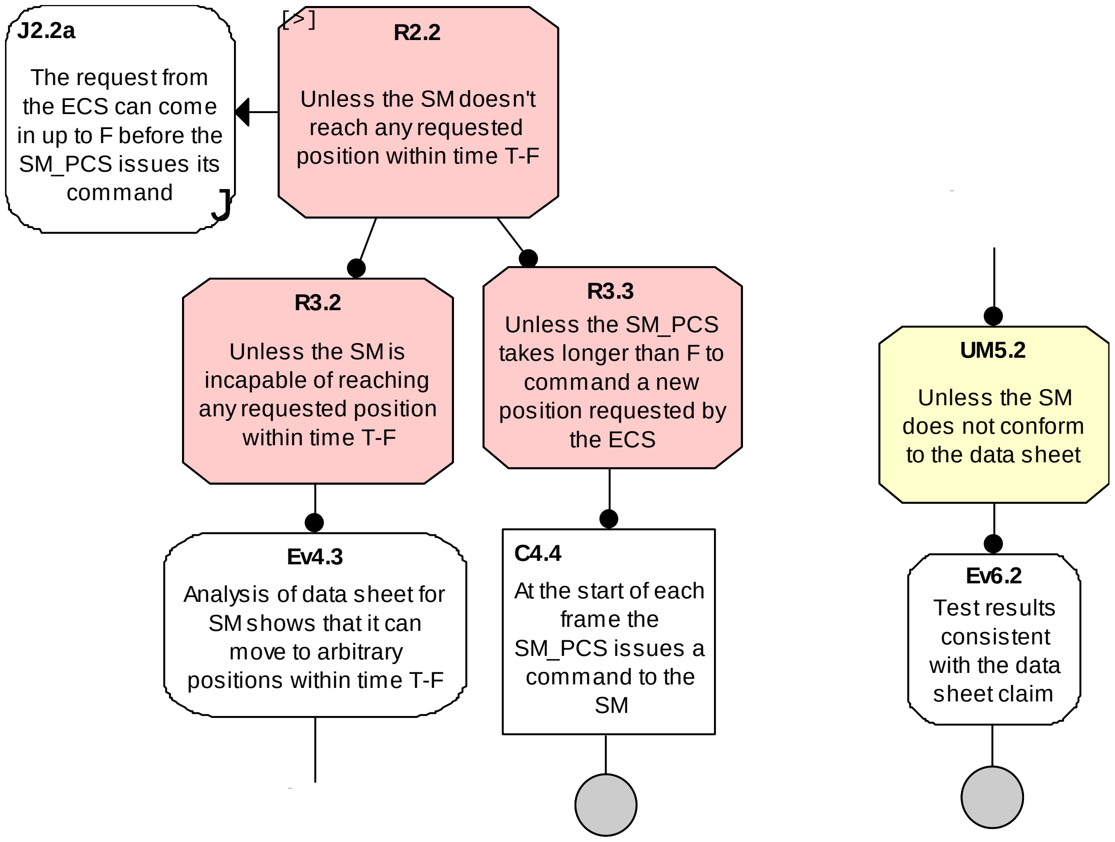

J2.2a
The request from the ECS can come in up to F before the SM_PCS issues its command
J
[>]
R2.2
Unless the SM doesn't reach any requested position within time T-F
R3.2
Unless the SM is incapable of reaching any requested position within time T-F
R3.3
Unless the SM_PCS takes longer than F to command a new position requested by the ECS
UM5.2
Unless the SM does not conform to the data sheet
Ev4.3
Analysis of data sheet for SM shows that it can move to arbitrary positions within time T-F
C4.4
At the start of each frame the SM_PCS issues a command to the SM
Ev6.2
Test results consistent with the data sheet claim

Figure 44: Eliminating a Defeater Regarding Speed of the SM

The confidence map for eliminating this defeater is similar to that in the more complex design. The major difference is the addition of defeater R3.3 that deals with the case in which the SM_PCS does not issue a new command to the *SM* in a timely manner.

C.2.3 The SM_PCS Does Not Issue Appropriate Commands (R2.3)

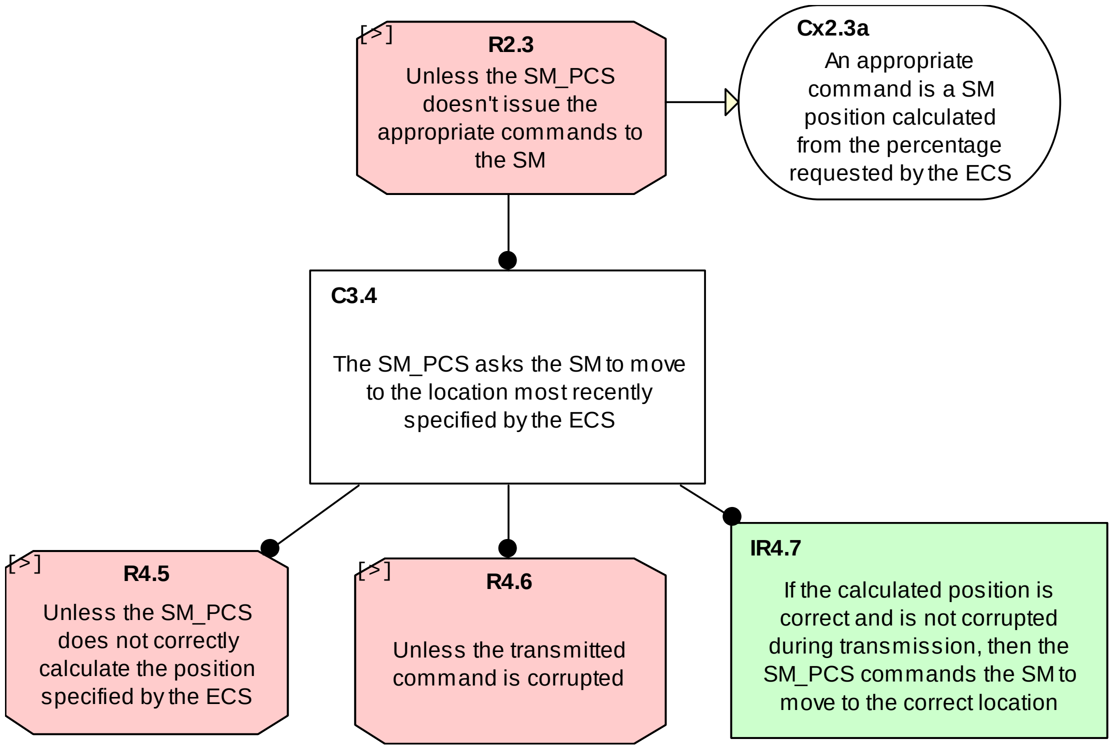

[>]
R2.3
Unless the SM_PCS doesn't issue the appropriate commands to the SM
Cx2.3a
An appropriate command is a SM position calculated from the percentage requested by the ECS
C3.4
The SM_PCS asks the SM to move to the location most recently specified by the ECS
[>]
R4.5
Unless the SM_PCS does not correctly calculate the position specified by the ECS
[>]
R4.6
Unless the transmitted command is corrupted
IR4.7
If the calculated position is correct and is not corrupted during transmission, then the SM_PCS commands the SM to move to the correct location

Figure 45: Eliminating a Defeater Regarding the SM_PCS *Issuing Appropriate Commands*

In the original design R2.3 dealt with the SM_PCS keeping track of the position of the *SM*. That is the major simplification of this alternative design and therefore that defeater is not appropriate to this design. Instead, R2.3 in this design is similar to R2.4 in the original design (similar, but also somewhat simplified). This simplification becomes clear if you compare claim C3.4 above (which directly eliminates R2.3) with the claim C4.7 in the original design (which has to eliminate R3.5 through R3.9 in order to eliminate R2.4).

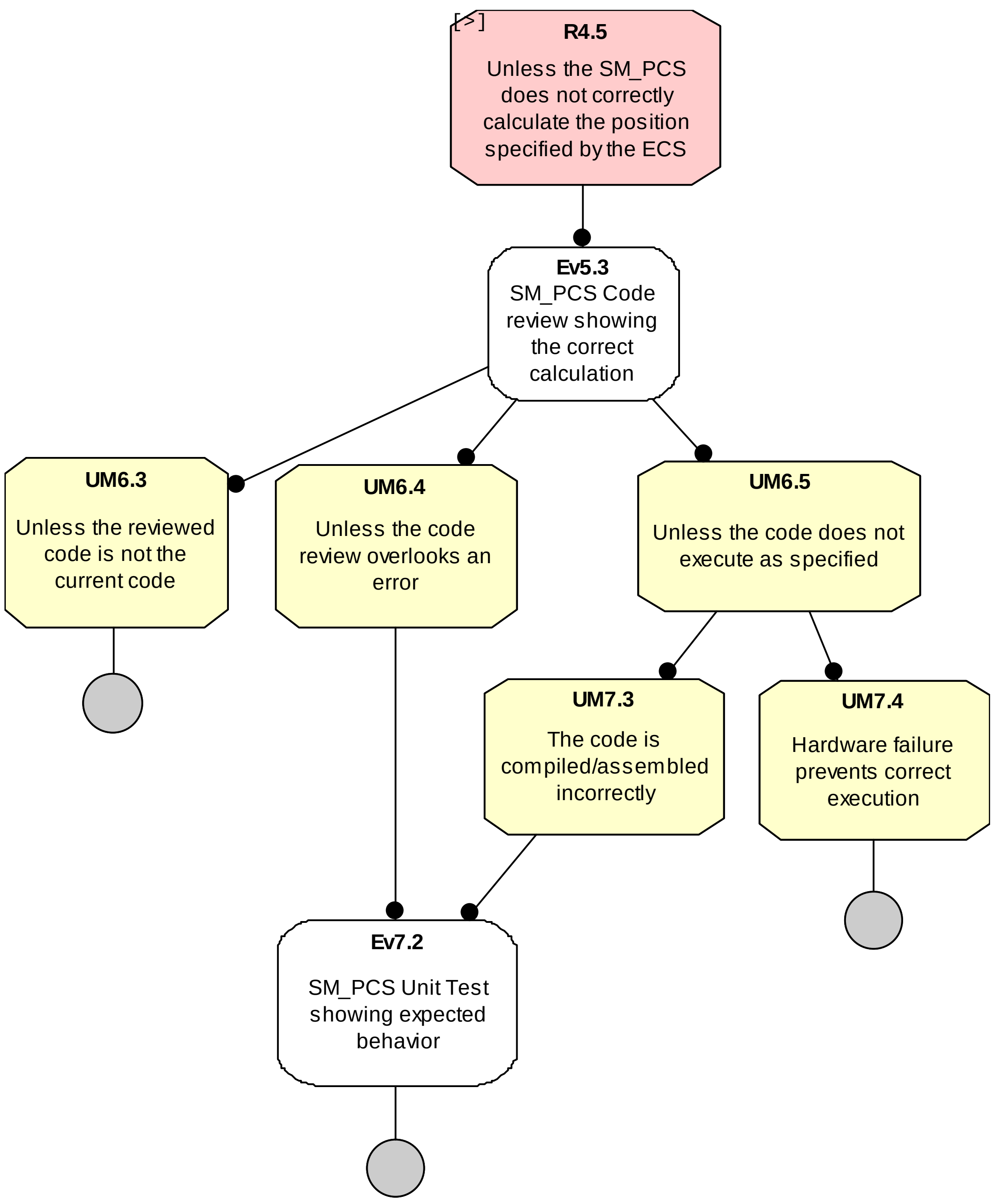

[>]
R4.5
Unless the SM_PCS does not correctly calculate the position specified by the ECS
Ev5.3
SM_PCS Code review showing the correct calculation
UM6.3
Unless the reviewed code is not the current code
UM6.4
Unless the code review overlooks an error
UM6.5
Unless the code does not execute as specified
UM7.3
The code is compiled/assembled incorrectly
UM7.4
Hardware failure prevents correct execution
Ev7.2
SM_PCS Unit Test showing expected behavior

Figure 46: Eliminating a Defeater Regarding the Position Calculation

In returning to this confidence map, we see that R4.5 is concerned with the correct translation of the position requested by the ECS (percentage open) to a position understandable by the *SM*. Again, note the similarities to the original design beginning at C4.7.

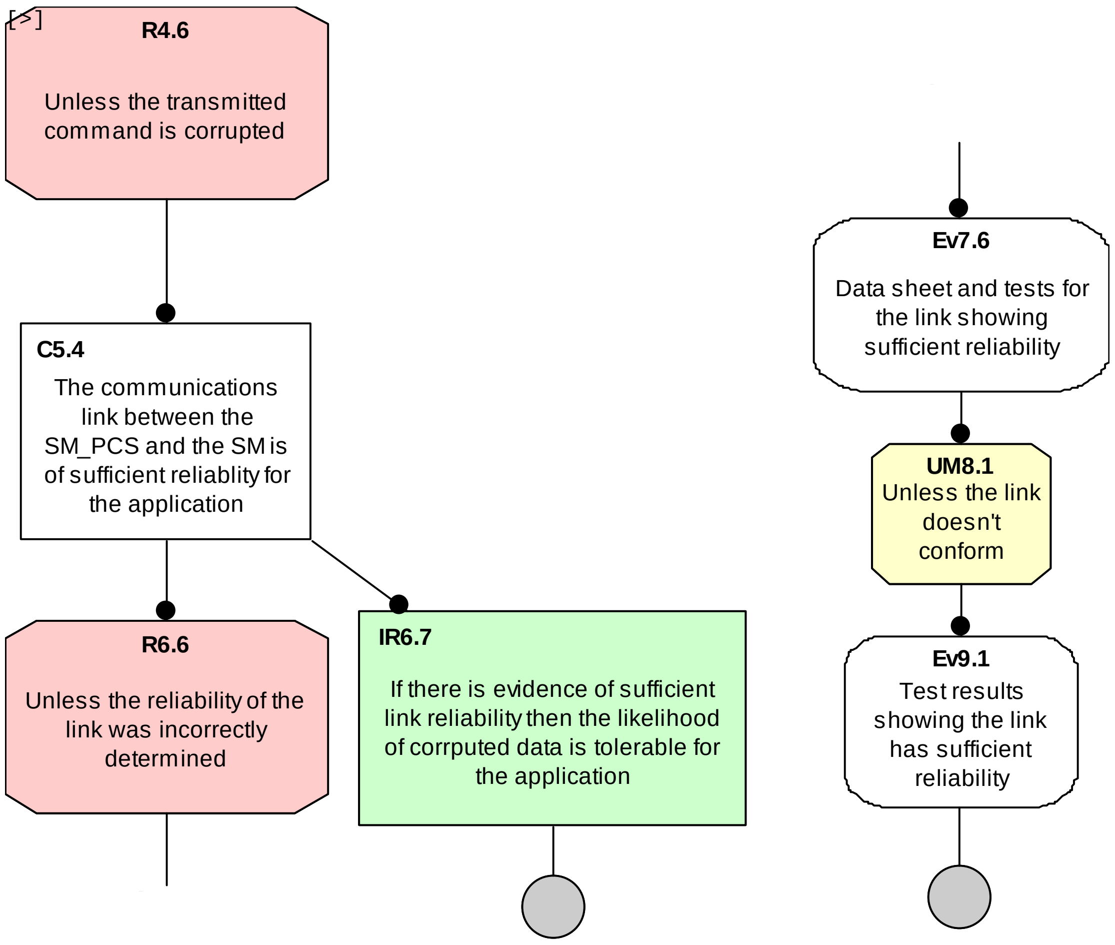

[>]
R4.6
Unless the transmitted command is corrupted
C5.4
The communications link between the SM_PCS and the SM is of sufficient reliablity for the application
R6.6
Unless the reliability of the link was incorrectly determined
IR6.7
If there is evidence of sufficient link reliability then the likelihood of corrputed data is tolerable for the application
Ev7.6
Data sheet and tests for the link showing sufficient reliability
UM8.1
Unless the link doesn't conform
Ev9.1
Test results showing the link has sufficient reliability

Figure 47: Eliminating a Defeater Regarding Data Corruption

Finally, the elimination of defeater R4.6 (regarding corrupted data transmission) is essentially identical to the elimination of defeater R3.10 in the original design.[10]

[10] Note, however, that the alternative design can be more robust in this regard. The original design commands the SM to move in increments. If a command is corrupted, the resultant position of the SM cannot be determined easily. In the alternative design, the PCS commands the SM to a specific position. If the command is corrupted in this case, and the error is transient in nature, then the SM likely will be commanded to the correct position during the next frame.

References

URLs are valid as of the publication date of this document.

[AFIS 2010]
Association Française d'Ingénierie Système. Process (Engineering). 2010. http://en.wikipedia.org/wiki/Process_%28engineering%29#Processes

[Blouin 2011]
Blouin, D., Senn, E., & Turki, S. "Defining an Annex Language to the Architecture Analysis and Design Language for Requirements Engineering Activities Support," 20–29. *Model-Driven Requirements Engineering Workshop (MoDRE)*. IEEE, 2011.

[CNSS 2010]
Committee on National Security Systems. *National Information Assurance (IA) Glossary* (CNSS Instruction No. 4009). CNSS, 2010.

[Delange 2014]
Delange, J. & Feiler, P. *Architecture Fault Modeling with the AADL Error Model Annex*. 40th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), August 2014.

[Delange 2014a]
Delange, Julien, Feiler, Peter, Gluch, David, & Hudak, John. *AADL Fault Modeling and Analysis Within an ARP4761 Safety Assessment* (CMU/SEI-2014-TR-020). Software Engineering Institute, Carnegie Mellon University, 2014. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=311884

[DeNiz 2012]
de Niz, Dionisio, Feiler, Peter, Gluch, David, & Wrage, Lutz. *A Virtual Upgrade Validation Method for Software-Reliant Systems* (CMU/SEI-2012-TR-005). Software Engineering Institute, Carnegie Mellon University, 2012.
http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=10115

[Dvorak 2009]
Dvorak, Daniel L., ed. *NASA Study on Flight Software Complexity* (NASA/CR-2005-213912). Office of Chief Engineer Technical Excellence Program, NASA, 2009.

[FAA 2009]
Federal Aviation Administration. *Requirements Engineering Management Handbook DOT/FAA/AR-08/32*. 2008. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR-08-32.pdf

[FAA 2009a]
Federal Aviation Administration. *Requirements Engineering Management Findings Report DOT/FAA/AR-08/34.* 2008. http://www.faa.gov/aircraft/air_cert/design_approvals/air_software/media/AR-08-34.pdf

[Feiler 2009a]
Feiler P., Gluch D., Weiss K., & Woodham K. “Model-Based Software Quality Assurance with the Architecture Analysis & Design Language.” *Proceedings of AIAA Infotech @Aerospace 2009.* Seattle, Washington, April 2009.

[Feiler 2010]
Feiler, P., Hansson, J., de Niz, D., & Wrage, L. “System Architecture Virtual Integration: An Case Study.” *Embedded Real-Time Software and Systems Conference (ERTS 2010).* Toulouse, France, May 2010.

[Feiler 2012]
Feiler, P. & Gluch, D., *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language*. Part of the SEI Series in Software Engineering series. Addison-Wesley Professional, 2012 (ISBN-10: 0-321-88894-4).

[Feiler 2012a]
Feiler, Peter, Goodenough, John, Gurfinkel, Arie, Weinstock, Charles, & Wrage, Lutz. *Reliability Improvement and Validation Framework* (CMU/SEI-2012-SR-013). Software Engineering Institute, Carnegie Mellon University, 2012. http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=34069

[Goodenough 2013]
Goodenough, John, Weinstock, Charles, & Klein, Ari. “Eliminative Induction: A Basis for Arguing System Confidence.” *International Conference on Software Engineering (ICSE 2013).* San Francisco, CA, May 2013. IEEE/ACM, 2013.

[GSN 2011]
Origin Consulting (York) Limited. *GSN Community Standard: Version 1.* http://www.goalstructuringnotation.info/documents/GSN_Standard.pdf

[Hayes 2003]
Hayes, J. H. “Building a Requirement Fault Taxonomy: Experiences from a NASA Verification and Validation Research Project,” 49–59. *14th International Symposium on Software Reliability Engineering (ISSRE).* Denver, CO, Nov. 2003. IEEE Computer Society Press, 2003.

[Hecht 2011]
Hecht, Myron, Lam, Alexander, & Vogl, Chris. “A Tool Set for Integrated Software and Hardware Dependability Analysis Using the Architecture Analysis and Design Language (AADL) and Error Model Annex.” *16th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS)*, 2011.

[ISO 2011]
International Organization for Standardization. ISO/IEC 15026-2:2011 *Systems and Software Engineering—Systems and Software Assurance —Part 2: Assurance Case.* 2011.
http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=52926

[Kelly 1998]
Kelly, T. "Arguing Safety—A Systematic Approach to Safety Case Management." PhD diss., University of York, Department of Computer Science, 1998.

[Lamsweerde 2003]
Van Lamsweerde, Axel & Letier, Emmanuel. "From Object Orientation to Goal Orientation: A Paradigm Shift for Requirements Engineering," 4–8. *Proceedings of Radical Innovations of Software and Systems Engineering*, LNCS, 2003. Springer-Verlag, 2003.

[Larson 2013]
Larson, B., Hatcliff, J. & Chalin, P. "Open Source Patient-Controlled Analgesic Pump Requirements Documentation," 28–34. *Proceedings of 5th International Workshop on Software Engineering in Health Care (SEHC)*, May 2013, San Francisco, CA. ACM/IEEE, 2013.

[Leveson 2012]
Leveson, Nancy. *Engineering a Safer World: System thinking Applied to Safety*. MIT Press, 2012 (ISBN 9780262016629).

[McDermid 2007]
McDermid, John A. *Developing Safety Critical Software: Fact and Fiction*. High Integrity Systems Engineering (HISE) Seminar Series, York University, 2007.
http://www.cs.york.ac.uk/hise/seminars/McDermid13Nov.ppt

[MRL 2009]
Metaphysics Research Laboratory, Center for the Study of Language and Information. *Stanford Encyclopedia of Philosophy: Defeasible Reasoning*. 2009. http://plato.stanford.edu/entries/reasoning-defeasible

[Paige 2009]
Paige, Richard F., Rose, Louis M., Ge, Xiaocheng, Kolovos, Dimitrios S., & Brooke, Phillip J. "FPTC: Automated Safety Analysis for Domain-Specific Languages," 229–242. *Models in Software Engineering.* Michel R. Chaudron, ed. *Lecture Notes In Computer Science 5421*. Springer-Verlag, Berlin, 2009.

[Pollock 2008]
Pollock, J. "Defeasible Reasoning," 451–469. *Reasoning: Studies of Human Inference and Its Foundations*. J. E. Adler & L. J. Rips, eds. Cambridge University Press, 2008.

[Powell 1992]
Powell, D. "Failure Mode Assumptions and Assumption Coverage," 386–395. *Digest of Papers, Twenty-Second International Symposium on Fault-Tolerant Computing*. Boston, MA, July 1992. IEEE Computer Society Press, 1992.

[Prakken 2010]
Prakken, H. "An Abstract Framework for Argumentation with Structured Arguments." *Argument & Computation 1*, 2 (June 2010): 93–124.

[Redman 2010]
Redman, David, Ward, Donald, Chilenski, John, & Pollari, Greg. “Virtual Integration for Improved System Design,” 57–64. *Proceedings of The First Analytic Virtual Integration of Cyber-Physical Systems Workshop* in conjunction with the Real-Time Systems Symposium (RTSS 2010). San Diego, CA, November 2010. Carnegie Mellon University, 2010. http://www.andrew.cmu.edu/user/dionisio/avicps2010-proceedings/proceedings.pdf

[SAE 2012]
SAE International. SAE AS-5506B:2012, *Architecture Analysis & Design Language (AADL)*. September 2012. Original publication in 2004.

[SAE 2011]
SAE International. SAE AS-5506/2:2011, *SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Data Modeling Annex; Annex D: Behavior Model Annex; Annex F: ARINC653 Annex*. 2011.

[SAE 2006]
SAE International. SAE AS-5506/1:2006, *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex A: Graphical AADL Notation, Annex C: AADL Meta-Model and Interchange Formats, Annex D: Language Compliance and Application Program Interface, Annex E: Error Model Annex*. 2006. Revision Error Model V2 in ballot for 2014 publication.

[Walter 2003]
Walter C. & Suri, N. “The Customizable Fault/Error Model for Dependable Distributed Systems,” 1223–1251. *Theoretical Computer Science 290* (2003).

[Weinstock 2013]
Weinstock, Charles, Goodenough, John, & Klein, Ari. “Measuring Assurance Case Confidence Using Baconian Probabilities,” 7–11. *Proceedings of 1st International Workshop on Assurance Cases for Software-Intensive Systems (ASSURE)* in conjunction with the *International Conference on Software Engineering (ICSE)*. San Francisco CA, May 2013. IEEE/ACM, 2013.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave Blank)	2. REPORT DATE March 2015	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Improving Quality Using Architecture Fault Analysis with Confidence Arguments	5. FUNDING NUMBERS FA8721-05-C-0003

6. AUTHOR(S)
Peter H. Feiler, Charles B. Weinstock, John B. Goodenough, Julien Delange, Ari Z. Klein, and Neil Ernst

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	8. PERFORMING ORGANIZATION REPORT NUMBER CMU/SEI-2015-TR-006
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) AFLCMC/PZE/Hanscom Enterprise Acquisition Division 20 Schilling Circle Building 1305 Hanscom AFB, MA 01731-2116	10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT Unclassified/Unlimited, DTIC, NTIS	12B DISTRIBUTION CODE

13. ABSTRACT (MAXIMUM 200 WORDS)

This case study shows how an analytical architecture fault-modeling approach can be combined with confidence arguments to diagnose a time-sensitive design error in a control system and to provide evidence that proposed changes to the system address the problem. The analytical approach, based on the SAE Architecture Analysis and Design Language for its well-defined timing and fault behavior semantics, demonstrates that such hard-to-test errors can be discovered and corrected early in the lifecycle, thereby reducing rework cost. The case study shows that by combining the analytical approach with confidence maps, we can present a structured argument that system requirements have been met and problems in the design have been addressed adequately—increasing our confidence in the system quality. The case study analyzes an aircraft engine control system that manages fuel flow with a stepper motor. The original design was developed and verified in a commercial model-based development environment without discovering the potential for missed step commanding. During system tests, actual fuel flow did not correspond to the desired fuel flow under certain circumstances. The problem was traced to missed execution of commanded steps due to variation in execution time.

14. SUBJECT TERMS AADL, error annex, requirements modeling, confidence maps, fault modeling, software architecture, argumentation	15. NUMBER OF PAGES 89

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT UL