Augmenting a Hazard Analysis Method with Error Propagation Information for Safety-Critical Systems

A Dissertation
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy
Computer Science

by
Fryad Khalid M. Rashid
December 2018

Abstract

Safety-critical systems need specific activities in the software development life cycle to ensure that the system will operate safely. The objective of this dissertation is to develop a new safety analysis method to identify hazards. The method uses error propagation information and the internal structure rather than the interfaces of a system. We propose development procedures to augment STPA (System-Theoretic Process Analysis) with error propagation information derived from the architecture description of a system represented in the AADL (Architecture Analysis & Design Language). We will focus on how the AADL error ontology can be used to assist in identifying errors, how those errors propagate among components, and whether the errors lead to hazards in the system. Our research shows that tracing error propagation leads to the discovery of hazards and additional information that other methods miss.

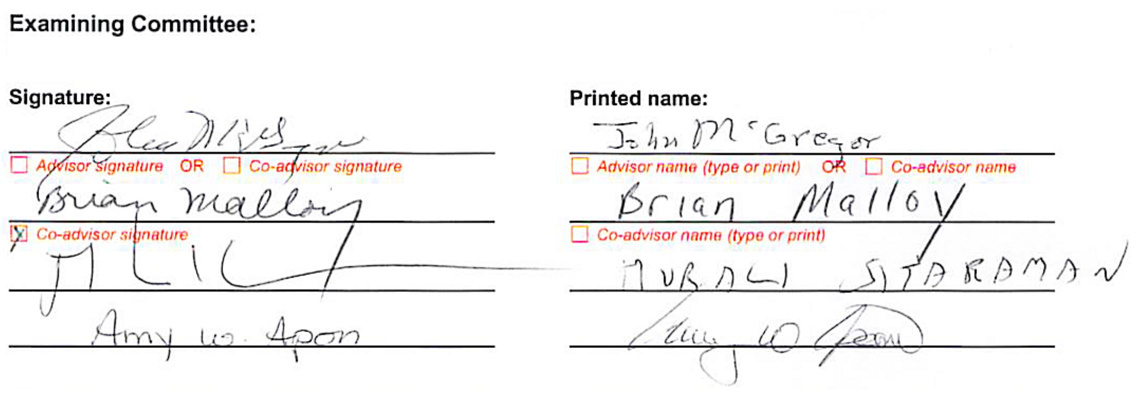
The new safety analysis method, Architecture Safety Analysis Method (ASAM), by augmenting STPA with early design information, is able to find more hazards, unsafe control actions, safety constraints and causes of the unsafe control actions than by using STPA alone. Our method leaves more false positives than STPA, but in safety analysis having false positive is preferred over missing actual hazards. We use the AADL error ontology to rigorously describe system component errors and how they propagate among components. We illustrate this rigorous description through several examples and we demonstrate that it yields hazards that an STPA analysis of the example did not find. In addition, we provide a mathematical notation and expressions so that formal analysis and verification of the hazards can be done to ensure that all causes of the hazards have been identified and that any developed safety constraints fully mitigate the hazards, through the use of compositional reasoning.

Dedication

I dedicate this doctoral dissertation to the cancer patients, nurses, physicians and professors in the MD Anderson Cancer Center at the University of Texas (Houston Campus).

Graduate School

This dissertation entitled "Augmenting a Hazard Analysis Method with Error Propagation Information for Safety-Critical Systems" and written by Fryad Khalid M. Rashid is presented to the Graduate School of Clemson University. We recommend that it be accepted in partial fulfillment of the requirements for the degree of Doctor of Philosophy with a major in Computer Science.



Examining Committee:

Signature:	Printed name:
[illegible]	John McGregor
☐ Advisor signature OR ☐ Co-advisor signature	☐ Advisor name (type or print) OR ☐ Co-advisor name
Brian Malloy	Brian Malloy
☒ Co-advisor signature	☐ Co-advisor name (type or print)
[illegible]	MURALI SITARAMAN
Amy W. Apon	[illegible]

Acknowledgments

I would like to express my deepest gratitude to my advisor, Dr. John D. McGregor, for his unwavering support, advice, and help that he has given me throughout this academic journey.

I would like to specially thank my best friend, Dr. Ethan McGee, for his help and support throughout this dissertation. In addition, I would like to thank my other fellow Ph.D. colleagues in the Strategic Software Engineering Research Group (SSERG), Yates Monteith and Wanda Moses, for their willingness to listen and offer constructive feedback about my research ideas. Also, I would like to thank my dissertation defense committee members for their feedback on this research and for their willingness to serve throughout the defense process. Finally, I would like to thank my scholarship program, The Higher Committee For Education Development (HCED) in Iraq, for helping to make my dreams come true.

I am deeply thankful to my lovely wife, Dr. Tavga A. Hussain, for her love, support, motivation and sacrifice. Without her, this doctoral degree would never have been done.

I would also like to thank everyone who has offered encouragement and support throughout my entire graduate career. This includes my parents, my siblings, my former co-workers at Sulaimani Polytechnic University / Clemson University and my close friends.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

John Knight wrote that ”Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment” [19]. His primary concern was the consequences of failure. There are many safety-critical system examples such as automotive systems, medical devices, aircraft flight controls, nuclear systems, etc. Malfunctions in these systems could have significant consequences such as severe injuries and mission failure. We know that those systems are dependent on software to the extent that they could not work without the software. So, they should be designed, verified, and validated very carefully to make sure that they obey the system specifications and requirements and are free from errors.

One of the motivations for doing this research is a study by the National Institute of Standard and Technology [42] found that 70% of software faults are introduced during the requirements and architecture design phases, while 80% of those faults are not discovered until system integration testing or later in the development life cycle. The cost of rework to fix the software problem is more expensive than finding and fixing during the requirements and design phase. This presents a big challenge in designing safety-critical embedded systems to address system faults that are recently discovered late in the development life cycle.

The software in safety-critical systems may not harm people directly, but it can control different types of equipment which may cause harm. There are many examples of safety systems that have failed due to software related faults such as ESA Rocket [5] , Therac-25 [23], PCA Pump [37], Toyota Prius [14], ARIANE 5 [24], Boeing 777-200 [41], and pacemaker types such as Medtronic/Biotronik/St.Jude Medical, [6]. This evidence shows that software errors, bugs and flaws in the requirements might have catastrophic consequences. For that reason, software operating critical functions should be very carefully designed and analyzed to avoid and prevent errors as well as their propagation.

From the system safety perspective, interaction complexity among embedded software and hardware platforms will increase criticality between system components. For that reason, safety can be described as an emergent behavior property that arises from the interaction of system components. That property requires understanding how the components are connected and how they interact with each other in the system design. Also, understanding is required because the concept of emergent behavior carries the idea that unsafe interaction among components could lead to violating safety requirements.

From the software perspective, in the early system design phase, we could notice that software components are scattered across so many nodes that it may introduce problems such as late or early delivery of data, loss of system operation or concurrent access to the same resources. Additionally, we could see in the early design phase of the software that it is possible to generate values beyond expected boundaries, that there is inconsistency between requirements or that there are mismatched timing requirements. If we know the faults cause errors and the those errors are not mitigated, then errors will propagate to the implementation and may not be caught by testing efforts. If these problems remain undetected during software testing, they could lead to serious errors and potential injury.

From safety and software perspectives, we need to deal with software aspects according to the safety requirements to build a safe system. The real challenges start here. How can one develop a safe system that fulfills safety constraints in the development of software-intensive systems? At the same time, hazard analysis techniques cannot describe the dynamic error behavior of the system and error propagation among system components. Because of that, traditional hazard analysis techniques depend on decomposition of the system with respect to the hierarchy of failure effects instead of the systems architectural model. For that reason, there is a need for a new hazard analysis method to support modeling and analyzing dynamic error behavior to find hazards in early development phases.

To the best of our knowledge, no guidance to augment existing hazard analysis techniques to find hazards and additional information that they missed exists. Also, to the best of our knowledge, no formalized guidance for hazard analysis techniques to improve the rigor of their methods exists.

1.1 Problem Statement

In this dissertation, we address the incompleteness of hazard analysis to find faults that exist beyond just the controller. We provide a systematic procedure to augment a recent hazard analysis approach, STPA

(System-Theoretic Process Analysis), with error propagation information and dynamic support contributors to find additional hazards that STPA did not find. In addition, the augmented method also addresses the lack of using mathematical models in safety analysis to ensure that all causes of a fault have been identified. Finally, we also address the lack of ability of ensure that a safety constraint fully mitigates a fault. The problem statement could be summarized as follows: The development of safety critical software would be improved if:

1. Guidance were available on how to augment the fault finding analyses to address incompleteness.

2. Guidance were available on how to formalize analysis to improve the rigor of the hazard analysis methods.

3. Recent hazard analysis methods were linked with compositional reasoning techniques to verify the safety constraints against the faults they mitigate.

1.2 Research Method

In this dissertation, we are analyzing the system before the architecture design is completed. The purpose of our work is to allow a stakeholder or safety analyst, at design time, to improve the safety of the system and build a safe product. Building on the work previously done [38], this research seeks to develop a safety analysis method that provides a deeper analysis of the system under review. The method will be developed in the form of an architectural model that describes internal failures in one of the major components (sensor, controller, actuator, controlled process, and controller input device) of the feedback control loop architecture, and we will show how the internal failure affects the other components in the form of three-way interactions. We will illustrate the method through several scenarios.

We will provide a theoretical foundation in the form of mathematical models for each communication channel and error in the feedback control architecture. This formal background will help to provide a guide to determine the expected behavior of the system because most of system design starts with a system specification and there is not necessarily a complete description of the system's behavior.

In addition to documenting and describing the error, we will also verify that the hazard situation which causes the unsafe behavior of the system is mitigated. This evaluation of the system safety verification help to verify the safety constraints based on the system / component design.

1.3 Contributions

The primary contributions of the dissertation will be as follows:

1. The introduction of a method of augmenting STPA (System-Theoretic Process Analysis) with error propagation information and error behavior state machines using AADL (Architectural Analysis and Design Language).
2. The introduction and analysis of a mathematical notation for each error flow in the feedback control loop architecture.
3. The identification of a method of verifying safety constraints / mitigating unsafe control actions in the hazard analysis process.

1.4 Dissertation Statement

In this dissertation, we assume that:

1. The error propagation information and error behavior state machine analysis can be applied to augment hazard analysis methods to find new hazards as well as specific causes and effects. Recent hazard analysis methods such as STPA (System-Theoretic Process Analysis) mechanisms can be adapted and used to provide support for this type of augmentation under different scenarios and development contexts.
2. The increasing complexity and criticality of modern computing systems are driving the need for enhanced hazard analysis methods. This augmentation is used to support modeling dynamic error behavior of the system during hazard analysis and representing a formal framework for the hazard identification in system architecture.
3. Safety-critical systems are driving the need to understand the feedback control loop architecture to manage variations, component failures and emergent behavior between system components resulting in hazards. Our framework can be applied to augment feedback control loops with mathematical notations and expressions to find hazardous conditions in a design, to improve rigor of STPA, and to formalize error descriptions in systems described in AADL.
4. Compositional reasoning, especially assume-guarantee contract techniques, can be applied to augment safety analysis methods to verify safety requirements identified for each component. The feedback control loop architecture in STPA can be adapted and used to support the contract.

1.5 Research Questions

In this dissertation, we are attempting to answer the following specific research questions in the safety-critical systems:

- RQ1) Can the safety of the system be improved by augmenting a hazard analysis method with error propagation information? We need to provide deeper analysis of the system under review. This allows us to find more faults before instantiating the product.
- RQ2) Can we analyze the internal failures of the components in order to show its effect of the other components during hazard analysis process? Modern information systems have many components. The interaction among components can make the system unreliable or unsafe. We need to know how the components interact with each other in the early design phase. For example, what will happen to the system if one of the components has internal failure?
- RQ3) How does knowledge of the interaction among the components in the early design phase help us to identify internal failures? Safety can be defined as an emergent behavior property of the system. This allows us to deal with interactions among system components.
- RQ4) How can the propagated error be presented in the form a mathematical model? We need to analyze system behavior in a formal way to expect unsafe interactions between system components. For example, what are those elements / parameters that effect the other components?
- RQ5) How can we identify the assume-guarantee contract for each component in the feedback control loop architecture? We need to know the specific information about specification and implementation for each component in the feedback control loop.

1.6 Dissertation Organization

The remainder of the dissertation is organized as follows: **Chapter 2** provides necessary background on STPA (System-Theoretic Process Analysis), AADL (Architecture Analysis and Design Language), EMV2 (Error-Model Annex version 2), safety verification methods, STPA augmentation reasons and false positive/-false negative in safety-critical systems; **Chapter 3** provides feedback control loop architecture augmentation sections such as augmenting STPA with error propagation information, mathematical notation for formalized augmentation of STPA and augmenting STPA with compositional reasoning; **Chapter 4** provides our method for analysis and evaluation; **Chapter 5** provides justifications and limitations of evaluation; **Chapter 6** discusses related work; **Chapter 7** introduces plans for future enhancements of each safety-critical application example; and **Chapter 8** provides conclusions based on the differentiation between our method and the current accepted standard.

Chapter 2

Background

In this section, we present relevant background material for this dissertation. This section will provide an overview of STPA (System-Theoretic Process Analysis), AADL (Architecture Analysis and Design Language), EMV2 (Error-Model Annex v2), STPA augmentation reasons and false positives / false negatives in safety-critical systems.

2.1 System-Theoretic Process Analysis

STAMP (Systems-Theoretic Accident Model and Processes) is a causality model built on system theory used to analyze system accidents. In this model of system safety, accidents could occur when components fail or when interactions between system components are inadequately handled by the control system. Therefore, STAMP views system safety as a control problem and introduces three main concepts: safety constraints, hierarchical control structures and process models [20].

STPA (System-Theoretic Process Analysis) is a top-down hazard analysis approach built on STAMP. The idea behind this approach is to avoid accidents by modifying the design of the system before implementation. The goal is to identify potential causes of accidents, that is scenarios that may lead to losses, so they can be controlled or eliminated in the system design or operations before damage occurs. In a nutshell, it provides scenarios to control and mitigate the hazards in the system design. The method consists of four steps to provide scenarios [21]:

1. The stakeholder establishes fundamental analyses to identify accidents and the hazards associated with those accidents. For example, if we assume that the accident is two aircraft colliding with each other,

then, the hazards could be the two aircraft violate minimum separation or the two aircraft enter unsafe atmospheric regions.

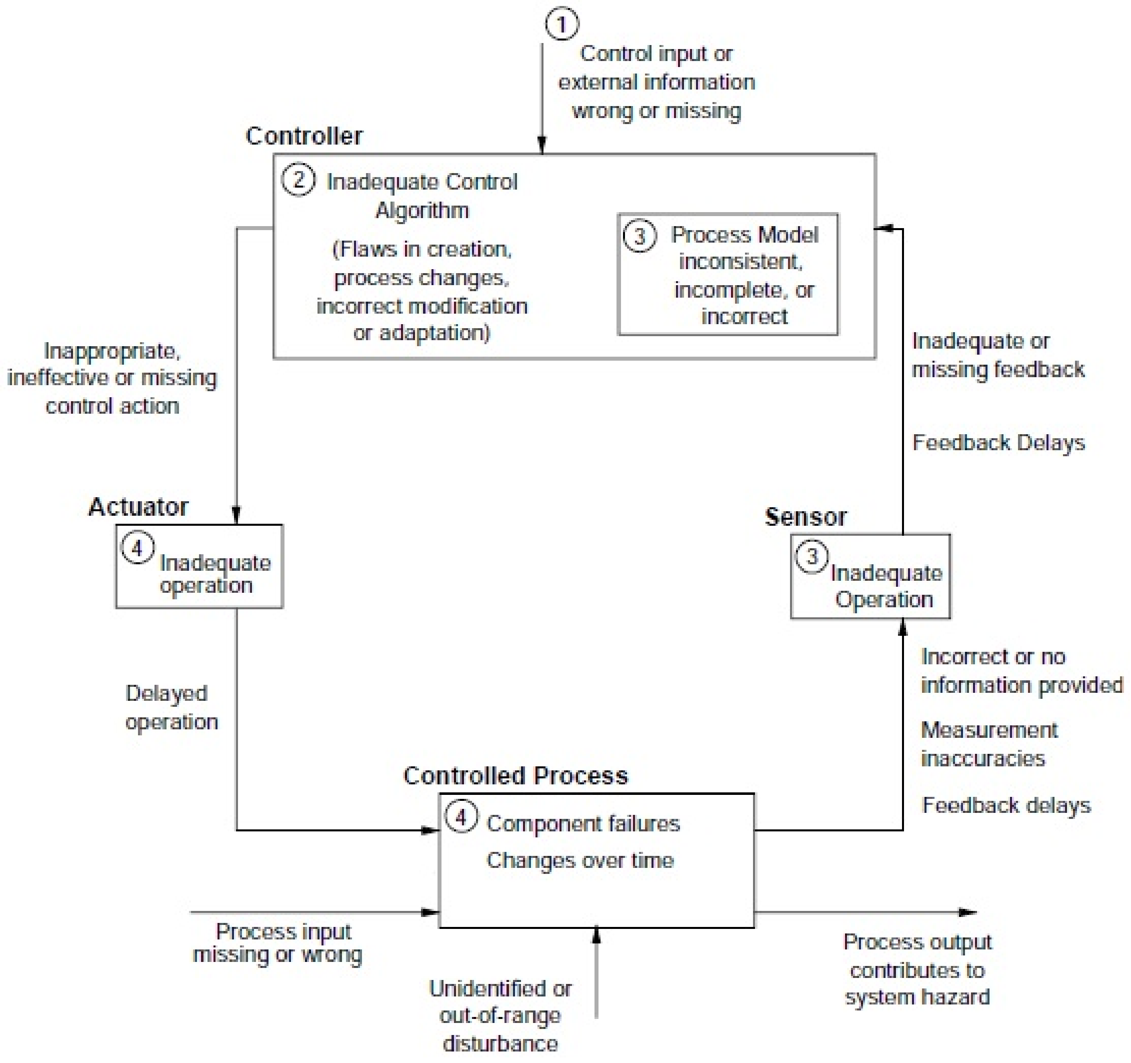
2. The stakeholder designs a feedback control loop for the system to identify major components such as sensors, controllers, actuators and the controlled process. Then, the stakeholder labels the control / feedback arrows as shown in figure 2.1. Essentially, the information from the figure 2.1 about each element is extracted into in the feedback control loop. For example, we need to know that the process model is the mental model of the controller. It is used to store information about variables and their values. The sensor is used to measure the values of the variables. The actuator is used to follow the instructions of the controller. The controlled process is used to implement commands that come from the controller through the actuator and send the result back to the controller through the sensor.The guide words among elements are used to identify hazards.

3. The stakeholder identifies unsafe control actions that could lead to hazardous states. The stakeholder should identify the following unsafe control actions, and then translate them to corresponding safety constraints:

    - A control action is needed for safety, but it is not provided.
    - An unsafe control action is provided.
    - A safe control action is provided too late or too early.
    - A safe control action is stopped too soon or applied too long.

4. The stakeholder identifies causal factors for the unsafe control actions. The safety analyst determines how each hazardous control action could occur by identifying the process model variables for the controller in the feedback control loop and analyzes each path to find out the cause of the unsafe control actions.

An example STPA analysis is provided in appendix B.



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 hazard

Figure 2.1: Feedback Control Loop in STPA

2.2 Architecture Analysis and Design Language

The Architecture Analysis and Design Language (AADL) is an architecture modeling language which uses a common description language to bind software and hardware elements together into a unified model. The language is a standard of the Society of Automotive Engineers (SAE) and is actively maintained. The language can be extended through the use of annexes; each annex is standardized independently and has separate syntax or functionality [32].

2.2.1 Language Overview

The language has keyword constructs for representing the execution platform, hardware components *(device, processor, bus, memory)*, software components *(data, subprogram, thread, process)* and the integration of the two *(system)*. AADL is a well known modeling language for safety-critical systems because it captures the hardware and software aspects of a system and their interactions. It is also possible to create *(abstract)* components in the early stage of system design that can then can be refined into either software or hardware at a later time. Additionally, AADL includes connections among components of various types such as *ports, bus access* and *data access* [32].

AADL components are composed of two separate definitions. The first is a specification of the component. It defines the component type and its external interfaces used to communicate with the rest of the world. The second is an implementation. It defines the inside of the component and how the interfaces are connected to sub-components to provide services. One of the important of specification feature is a *port*. The *port* represents the interaction point for an event / notification, data transfer or a combination of events and data. The *port* can be specified for components such as *devices, threads, processes* and *systems*. There are three types of ports [11]:

1. The *event port* represents the reception of a notification and carries the signal without any data. For example, detection of data from a sensor such as temperature being above or below a defined threshold. When using this kind of port, the software is waiting for an incoming event.

2. The *data port* represents an interface which receives data flow and keeps track of the latest values. The port value will be updated as soon as possible when a new data value is received.

3. The *event data port* represents an interface that combines the *event port* and the *data port* to exchange data with notification in the same entity. It sends the signal (event) associated with the data.

AADL also introduces the concept of *flows*. The *flow* enables the representation and analysis of logical paths through an architecture, and can reflect scenarios that can be the basis for system level analysis. AADL can support and declare *flow* specifications and implementations for the components. The *flow* specification defines a logical flow from input port to output port of the component. The logical flow can be represented as either a data flow, fault event flow or control flow. The *flow* specification can define three types of flows for a component [9, 15]:

1. The *flow source* defines a flow which originates within a component and emerges from the component through an outgoing port.

2. The *flow sink* defines a flow which terminates within a component and enters the component through an incoming port.

3. The *flow path* defines a flow which enters a component through an incoming port and emerges through an outgoing port.

The *flow* implementation elaborates the *flow* specifications through sub-components. It starts with the incoming port of the flow specification, follows a sequence of connections and sub-components and ends with an outgoing port. The last type declaration of a *flow* is the end-to-end flow which defines a complete flow path from the starting component to the final component in the system.

In this dissertation, we use the *flows* concept to analyze important system characteristics. For instance, we show (in listing 2.1 and figure 2.2) a flow within a speed control system in the vehicle. This flow originates at Brake_Pedal_Sensor, traverses through the Speed_Control_System and terminates at Throttle_Actuator. When the driver pushes the brake pedal, the Brake_Event will pass through the outgoing event data port via *flow source* to the Speed_Control_System. Then, the Speed_Control_System receives the Brake_Event through incoming event data port, adjusts the throttle setting value to decrease speed of the vehicle and sends the value to the throttle value via the *flow path* to the out event data port. The Throttle_Actuator reduces fuel into the engine based on the received setting value through the incoming event data port. The *flow sink* shows the impact of Brake_Event on the Throttle_Actuator and stops the flow.

```
device  Brake_Pedal_Sensor
    features
        Brake_Event: out event data port;
    flows
        Brake_Flow: flow source Brake_Event;
end Brake_Pedal_Sensor;

system Speed_Control_System
    features
        Brake_Event: in event data port;
        Throttle_Adjusting: out event data port;
    flows
```

```
        Cruise_Flow: flow path Brake_Event -> Throttle_Adjusting;
end Speed_Control_System;

device Throttle_Actuator
    features
        Throttle_Adjusting: in event data port;
    flows
        Throttle_Flow: flow sink Throttle_Adjusting;
end Throttle_Actuator;
```

Listing 2.1: Flow AADL Example

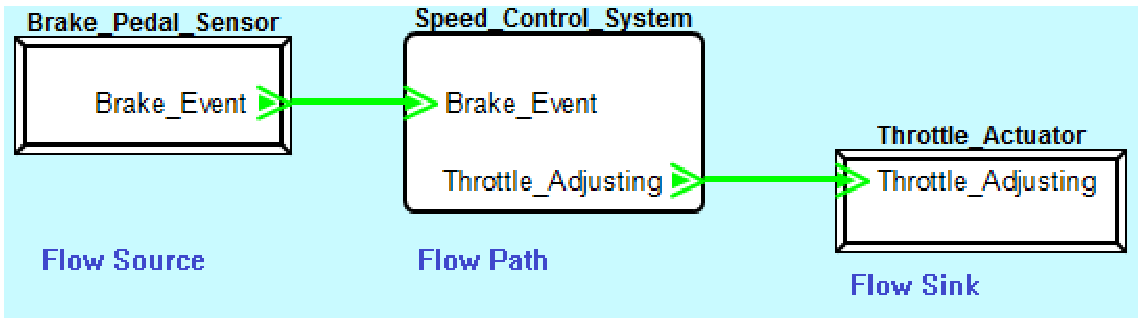


Figure 2.2: Flow Visualized

2.3 Error-Model Annex

In designing safety-critical systems, we need to show that the system is reliable and resilient to different types of failures. For that purpose, a number of different types of safety analysis methods are required in order to show the system is safe and has appropriate measures to handle critical failures. We know that the main objective of a Model-Based Engineering (MBE) approach is to automate the development process such as building model based tools to automate the safety validation process. The core AADL language does not contain the language constructs to do safety analysis. For that reason, the core language is extended with an annex dedicated to safety analysis, Error Model Annex v2, which provides the required semantics for catching safety information and showing error propagation through the architecture.

The Error Model Annex is used to capture what could go wrong in the architecture, to specify which errors could arise from the components, how those errors propagate through the architecture and how they impact components behavior. This annex has four levels of abstraction. In order, they are [9, 11, 33]:

1. **Error Ontology:** Represents error types in a hierarchical structure to support hazard analysis and assist modelers to make sure that they have considered different types of error propagation in the interaction among system components. The error ontology can distinguish errors within the architecture and how each type can propagate or impact the components' behavior. We provide a hierarchical structure of error ontology in appendix A. In here, we give a brief description of each type of error [10, 38]:

   - `Service Errors`: Represent errors which are related to delivering service for items. Service errors can be categorized as omission errors, no service delivered (such as loss of a message or command), and commission errors, which represent unexpected service provided (such as unintended incoming data).
   - `Value Errors`: Represent errors which are related to the value domain of a service. Value errors can be categorized as value errors for individual service items like out of range sensor reading, value errors for sequences of service items like bounded value change and value errors related to the service as a whole like out of calibration.
   - `Timing Errors`: Represent errors which are related to the time domain of a service. Timing errors can be categorized as individual service items like early/late item delivery, timing errors for sequences of service items like rate errors, and timing errors related to the service as a whole like early/delayed services.
   - `Replication Errors`: Represent errors which are related to delivery of replicated services. For example, replicate service items delivered for one recipient or to multiple recipients.
   - `Concurrency Errors`: Represent errors which are related to behavior of concurrent systems like executing tasks concurrently to access shared resources. Errors are distinguished between race condition errors and mutual exclusion errors.
   - `Access Control Errors`: Represent errors which are related to operation of access control services like authentication and authorization errors.

   Developers can use the error ontology in AADL, which is defined in the `ErrorLibrary.aadl` in OSATE (Open Source AADL Tool Environment), when they design a system. Although, if the existing names do not fit with their system, developers can define their own error types in a package as shown in listing 2.2.

```
package My_Error_Types
    public
        annex EMV2 { **
            error types
                BrakeEventError: type;
                BadValue: type extends BrakeEventError;
                NoValue: type extends BrakeEventError;
            end types;
        **};
end MyErrorTypes;
```

Listing 2.2: Define Error Types in AADL

2. **Error Flows in the Architecture:** Represents how errors flow through the system architecture, after we identified the type of errors for the system in the previous step. In this step, EMV2 allow us to describe:

    - **Error Propagation:** Define incoming and outgoing errors for a component. The error propagation has the ability to specify what error type the component can expect to receive or send, but it does not have the ability to specify how incoming and outgoing error types relate to each other. For example, no one knows how an error from an incoming port can generate an error on an outgoing port or how an error in the processor effects a process and results in an outgoing propagation on its data port. The error flow can solve the relation problem between errors.

    - **Error Flow:** Defines the relationship between incoming and outgoing error propagations. There are three types of error flows:

        (a) `Error Source:` Specifies that the error originates from this component. Especially, when the component has an internal failure, it will generate the error being propagated.

        (b) `Error Path:` Specifies the transmission of an incoming error propagation to an outgoing error propagation. The error is propagated by a component itself. The component is passing the error from the incoming port to the outgoing port.

(c) `Error Sink:` Specifies that the error is stopped at this component. The error can be handled or mitigated at this component because it impacts the architecture.

In listing 2.3, we show how to define error propagations for each component such as the Brake_Pedal_Sensor, Speed_Control_System and Throttle_Actuator. We can see that the device Brake_Pedal_Sensor is an *error source* for any error type which propagates out of the event data port Brake_Event. The device Throttle_Actuator is an *error sink* for any error type. The Speed_Control_System is an *error path* because an error of any type can propagate into the system through the incoming event data port, and the error can propagate out through the outgoing event data portm Throttle_Adjusting. We do not know how the propagated error impacts the component at this stage. We will show that in the next step.

```
device Brake_Pedal_Sensor
    features
        Brake_Event: out event data port;
    annex EMV2 {**
        use types ErrorLibrary;
        error propagations
            Brake_Event: out propagations {AnyError};
        flows
            Sensor_Flow: error source Brake_Event {AnyError};
        end propagations;
    **};
end Brake_Pedal_Sensor;

system Speed_Control_System
    features
        Brake_Event: in event data port;
        Throttle_Adjusting: out event data port;
    annex EMV2 {**
        use types ErrorLibrary;
        error propagations
            Brake_Event: in propagations {AnyError};
            Throttle_Adjusting: out propagations {AnyError};
        flows
```

```
            Control_Flow:error path Brake_Event -> Throttle_Adjusting;
        end propagations;
    **};
end Speed_Control_System;

device Throttle_Actuator
    features
        Throttle_Adjusting: in event data port;
    annex EMV2{**
        use types ErrorLibrary;
        error propagations
            Throttle_Adjusting: in propagations {AnyError};
        flows
            Actuator_Flow: error sink Throttle_Adjusting {AnyError};
        end propagations;
    **};
end Throttle_Actuator;
```

Listing 2.3: Error Propagation Concept in AADL

3. **Component Error Behavior:** Represents how errors impact a component's behavior. For instance, an error can change the operating mode of a component from nominal mode to failure mode. For that purpose, EMV2 provides an error behavior state machine (EBSM) to catch safety behavior of a component according to its states, internal failure/internal error events and errors coming from outside world.

   We can define the states of a component regarding its error behavior. Basically, error behavior state machines (EBSM) allows stakeholders to define states according to system operational environment. For example, we consider two states for each component: Operational state (which means the component is active and it is operating without any error), and Failed state (which means the component is possibly active but not without error). A transition can define the condition under which a state change could occur. Each transition consists of a source state which is the initial state of the component and a destination state which is the final state after the transition is triggered. A condition is an error event that needs to be activated or triggered to activate the transition. We show the error behavior of a component in listing 2.4, by two transitions. The first transition is from Operational state to Failed state when the Failure error event is raised. The second is from Failed state to Operational state when the Recovery event is activated.

```
error behavior Fail_And_Recover
    events
        Failure_E: error event;
        Recovery_E: recover event;
    states
        Operational: initial state;
        Failed: state;
    transitions
        t1: Operational -[Failure_E]->Failed;
        t2: Failed -[Recovery_E]->Operational;
end behavior;
```

Listing 2.4: Error Behavior State Machine

4. **Composite Error Behavior:** Represents the state of the component according to state of the sub-components. In another context, the state of a system depends on the state of the components. For example, a computer system is in the Failed state when one of its sub-parts (processor or memory) is Failing. EMV2 can support this through composite error behavior specifications. We show this in listing 2.5; when a system consists of two sensors to provide temperature, the main system will be in the failure if both sensors are failing. Otherwise, it will be in the operational state.

```
composite error behavior
    states
        sensor1.Failed and sensor2.Failed -> Failed;
        sensor1.Operational or sensor2.Operational -> Operational;
end composite;
```

Listing 2.5: Composite Error Behavior

2.4 Safety Verification Methods

Generally, verification concentrates on showing consistency among system implementation and functional specification but for the safety-critical systems that is not enough. The implementation of the system must satisfy the safety requirements. Specifically, the safety verification is different than functional verification. The functional verification ensures that the system fully satisfies its specification, but safety verification uses the result of safety analysis methods to ensure that the system meets the safety constraints [46]. We can classify safety verification into two types:

2.4.0.1 Safety Verification in Software Testing

The software safety verification can be achieved by in two ways [2]:

1. Dynamic Safety Testing: requires the execution of the software, the result can be evaluated with respect to system safety features. In addition, testers monitor behavior of the system during testing like entering erroneous values / out of range values.

2. Static Safety Analysis: looks over the code and design document of the system. It is used to verify accuracy of safety feature with regard to the actual system/software.

2.4.0.2 Safety Verification in Software Architecture

Compositional reasoning is a partitioning formal analysis of a system architecture into a sequence of verification tasks which correspond to the decomposition of the system architecture. This partitioning of verification effort will help to find proof for each subcomponent within the system architecture so that the analysis will handle a large system design. In a nutshell, system level properties depend on the properties of the components within the system. For example, from safety perspective, system safety depends on the safety properties of the components. For that purpose, AGREE (Assume Guarantee REasoning Environment) was created for managing proofs of components described in AADL [7].

AGREE is a compositional verification tool that can be used to confirm the behavior of a component. It follows the popular assume guarantee reasoning model which states that provided the assumptions about a component's inputs are met the component can provide certain guarantees about its output. The guarantees correspond to component requirements, and the assumptions correspond to the environmental requirements that were used in verifying the component requirements. The contract specifies the information which is needed to reason about the component's interaction with other components of the system [26].

As an example, let's build a safety constraint based on error values: In an ACC (Adaptive Cruise Control) system, the sensor should measure the distance between vehicles in the specific range (0-100) feet. Any values beyond that limit are likely erroneous and distance cannot be negative. We will focus on the actions of the controller. If the controller has an assumption that it will receive values in the range, it will perform a safe action if the values it received are in the specified range. Outside of this range, whether a safe action is performed cannot be guaranteed. An implementation of this example is shown in listing 2.6

```
system sensor
    features
        distance: out data port Base_Types::Integer;
    annex agree {**
        guarantee "Sensor output is between 0 and 100": distance >= 0 and
            distance <= 100;
    **};
end sensor;

system implementation sensor.i
    annex agree {**
        assert distance = if distance < 0 then 0 else if distance > 100 then
            100 else distance;
    **};
end sensor.i;

system controller
    features
        distance: in data port: Base_Types::Integer;
    annex agree {**
        assume "Controller should receive values among 0-100": distance <= 100
            and distance >= 0;
    **};
end controller;
```

Listing 2.6: AADL AGREE Annex ACC system example

Additionally, the Strategic Software Engineering Research Group (SSERG) at Clemson University extended AGREE to XAGREE (eXtended Assume Guarantee REasoning Environment) to support the inheritance features of the AADL language. This addition allows XAGREE to produce more maintainable and less complex verification assets than AGREE for architectures that incorporate inheritance such as Software Product Lines and Dynamic Software Product Lines [25].

2.5 System-Theoretic Process Analysis Augmentation Reasons

From our experience in applying STPA on different types of safety-critical systems, a systematic literature review that we conducted in the domain and the understanding of the method in the system safety community, we come to the conclusion that STPA needs significant augmentation because of the following reasons:

1. *STPA lacks precise information about each major component:* STPA is a system method built on the system theory. Nancy Leveson states ”The systems approach focuses on systems taken as a whole, not on the parts taken separately” [21]. This means that system theory allows STPA to provide accidents, hazards, and causes at the system level not the component level. However, in AADL, systems include hardware and software components as described in section 2.2.1. This allows stakeholders to deeply analyze the system and identify specific causes of system failure. The causes can be either software, hardware or both together leading to the system failure.

2. *STPA uses a feedback control loop as a system model but doesn’t use the feedback control loop used as an architectural model:* STPA is a recent hazard analysis technique and depends on a monolithic decomposition of the system with respect to the hierarchy of failure effects instead of the system’s architectural model. The reason for using monolithic decomposition is because of system complexity. Nancy Leveson states ”Increased interactive complexity and coupling make it difficult for the designers to consider all the potential system states or for operators to handle all normal and abnormal situations and disturbances safely and effectively” [20]. This means that system theory does not let STPA access lower subcomponents in the feedback control loop because of the system complexity. At the same time, software engineering principles allow stakeholders to decompose systems into subsystems to solve the complexity problem. For example, in AADL, we can decompose the system into two layers with each layer containing subsystems. We can bind the layers together, and we can show error propagation between the layers. This deeper analysis could help stakeholders identify the source of faults or hazards

in the feedback control loop architecture. For instance, system software can have failures because of error propagation from hardware to software. If the stakeholder upgrades the software to resolve the problem, the problem will come back after the upgrade because the problem is not the software but the hardware. The stakeholder needs to change the hardware not upgrade the software to resolve the problem.

3. *STPA does not use the reliability principle (fault$\rightarrow$ error $\rightarrow$ failure):* STPA looks for causes of hazards based on system thinking, and it considers the result of the hazard as an accident. There is not a component reliability concept in STPA because it analyzes safety based on the system. Nancy Leveson states ”Safety then can be treated as a dynamic control problem, rather than a component reliability problem” [22]. She also states ”In general, words like ”failure” and ”error” should be avoided, even as causes, because they provide very little information and are often used to avoid having to provide more information or to think hard about the cause” [22].

   However, reliability engineering principles provide system safety based on the reliability of each individual component. The system will not be safe if the components have failures. Additionally, in AADL, we use the error ontology to support hazard analysis. This allows stakeholders to identify hazards based on the reliability concept. The basic principles of reliability needed in this dissertation are: *fault* is root cause of the error, *error* is the difference between the measured value and the correct value, and *failure* is the termination of a product’s ability to do a required function.

4. *STPA does not use a formalization concept:* STAMP provides a framework for STPA to investigate and analyze accidents. This is a major limitation of STPA. Nancy Leveson states ”Three basic constructs underlie STAMP: safety constraints, hierarchical safety control structures, and process models” [21]. This limitation tells us that STPA does not have any connection with formal principles such as system architecture analysis, mathematical models, state machine analysis, error propagation, etc. However, the formalization concept helps improve the rigors of safety analysis methods.

2.6 False Positive and False Negative in Safety-Critical Systems

The safety analysts / stakeholders are expecting the system failure before it occurs. They apply preventative techniques to the systems to avoid the failure or reduce time to repair by preparing the system for upcoming failure events. For that purpose, stakeholders proposed proactive fault handling methods to deal with faults in order to prevent the system failure or to minimize the system downtime. This method combines failure prediction and proactive action. The preventative actions are triggered by prediction of upcoming failure events, and the repair actions are prepared for imminent failure to reduce time to repair. Thus, the systems require the prediction of failures to judge a faulty situation or to make a decision to handle the fault. It is very possible that the failure predictor will be wrong in its predictions. For example, the predictor might detect an upcoming failure event that never occurs, this is called false positive (FP). Also, the predictor might fail to predict a failure, this is called false negative (FN) [43].

Now, we provide the real world examples to identify false positives and false negatives. For example, from the software testing perspective, when a test is executed and, in spite of it running correctly, the test shows us there is an error, the tester will search for the nonexistent bug adding a lot of cost. That is a false positive (FP). In addition, when the execution of a test shows no faults although there is a bug in the software application. That is a false negative (FN).

From the the system safety perspective, in ACC system, if a warning / automatic braking happens before the critical distance, this is a false positive (FP). Also, if a warning / automatic braking happens too late, this is a false negative (FN) too [28].

Chapter 3

Feedback Control Loop Architecture

3.1 Augmenting a Hazard Analysis Method with Error Propagation Information

As stated in section 1.3, the first goal of this research is:

- The introduction of a method of augmenting STPA (System-Theoretic Process Analysis) with error propagation information and error behavior state machines using AADL (Architectural Analysis & Design Language).

The first goal is associated with three of the research questions shown in section 1.5:

- Can the safety of the system be improved by augmenting a hazard analysis method with error propagation information?
- Can we analyze the internal failures of the components in order to show its effect on the other components during hazard analysis process?
- How does the interaction of the components in the early design phase help us to identify internal failures?

The traditional version of STPA cannot describe dynamic error behavior, state transitions, and error propagations among system components. This is because of the reasons described in section 2.5, thus, as part of this dissertation, we have augmented STPA with error propagation information and error behavior state machines to support modeling / analyzing the dynamic error behavior of the system during hazard analysis process and to identify potential internal failures of the major components during application of STPA.

This will allow us to identify additional hazards based on different criteria for existing application examples, identify additional unsafe control actions in different analysis, identify general / specific causes for unsafe control actions and blend safety constraints to the systems architecture to build a safe product.

We propose a method that helps stakeholders or safety analysts, during hazard analysis, to consider the error behavior of a component, to identify each path in the form of three-way interactions among components in the feedback control loop architecture and to analyze the trace of each hazard. We illustrate this method by describing several scenarios and examples to support it. We will provide models for the scenarios by using the Architecture Analysis and Design Language (AADL) supported by Open Source AADL Tool Environment (OSATE) [16] to develop a architecture representation. We extend each step in the STPA process with error propagation information and dynamic error behavior. In addition, we add a final step to merge safety aspects of the system into the systems architecture. The proposed method consists of the following steps:

1. **Identify hazards using different criteria:** This step involves identifying accidents based on the system operational context. The error ontology provides guidance in identifying hazards.

2. **Build control structures with contributors:** This step is the construction of a feedback control loop with finite state machines for describing dynamic behavior of the system, and add error propagation specifications across the system to analyze the trace of the hazards which may lead to the accidents.

3. **Identify unsafe control actions using tracing:** This step helps to identify the unsafe control actions based on the error propagations tracing. It identifies the error behavior of a component which can lead to inadequate control actions that could become an unsafe action. This step also helps to identify any error flow for which a corresponding safety constraint needs to be created to mitigate the identified hazards.

4. **Identify specific causes:** This step helps to identify causes for the unsafe control actions in the feedback control loop. Generally, it needs to select the component first and then specifically look for the causes which relate internally and external to the component.

5. **Develop safety architecture:** A safety architecture is implemented based on the safety constraints identified in the previous steps. It blends safety aspects of the system into the overall system architec-

ture.

We will create this augmentation to help the stakeholder or the safety analyst in identifying and evaluating dysfunctional behavior of the system during hazard analysis. The main goal of analyzing the behavior of the system is to identify hazardous control actions by considering the specification of error propagations across the system and operating states of the system which can have an effect on control actions.

Using the characteristics of event driven models and error propagation information in the safety analysis can assist in providing an effective way to annotate and assess all possible paths in the feedback control loop. In this method, steps 1 through 4 can identify the source of errors and their impact on the other components during operation. Step 5 helps to enhance the safety of the system by feeding the identified hazardous situations or propagated error events into the system's architecture to create constraints to mitigate the hazards.

The heart of the STPA approach is a feedback control loop to analyze safety of the system. We wish to improve STPA to facilitate a deeper analysis of the system. Figure 3.1 shows the feedback control loop with error propagation and finite state machine models. Figure 3.1 consists of four component types: sensors, controllers, actuators, and the controlled process. These components have incoming and outgoing ports used to send / receive data as well as error events. The connections among components represent the nominal control flow as well as the error propagation path.

The error propagation path follows port connections from sensors to controllers, controllers to actuators, actuators to the controlled process and tje controlled process to the sensors. First, the sensors are used to measure the values of attributes and send them to the controller. Each sensor has two states, operational and failed. Second, the controller acquires information about the state of the process from measured variables and controlled variables, then the controller uses this information for an initial action by manipulating controlled variables to maintain the operational process within predefined limits. The controller is used to regulate the process variables and send commands to the actuator. The controller consists of a process model which is used to present variables and their values. The controller has two states: normal and error states. The normal state shows the status of the variables in normal operation. The error state shows abnormal values of the variables. Third, the actuator follows the controller's instructions to execute the commands. The actuator has two states, operational and failed. Fourth, the controlled process is used to show processes inside of the controller and implement the controller's decision. The error flow passes errors inside of the component from an incoming port to an outgoing port.

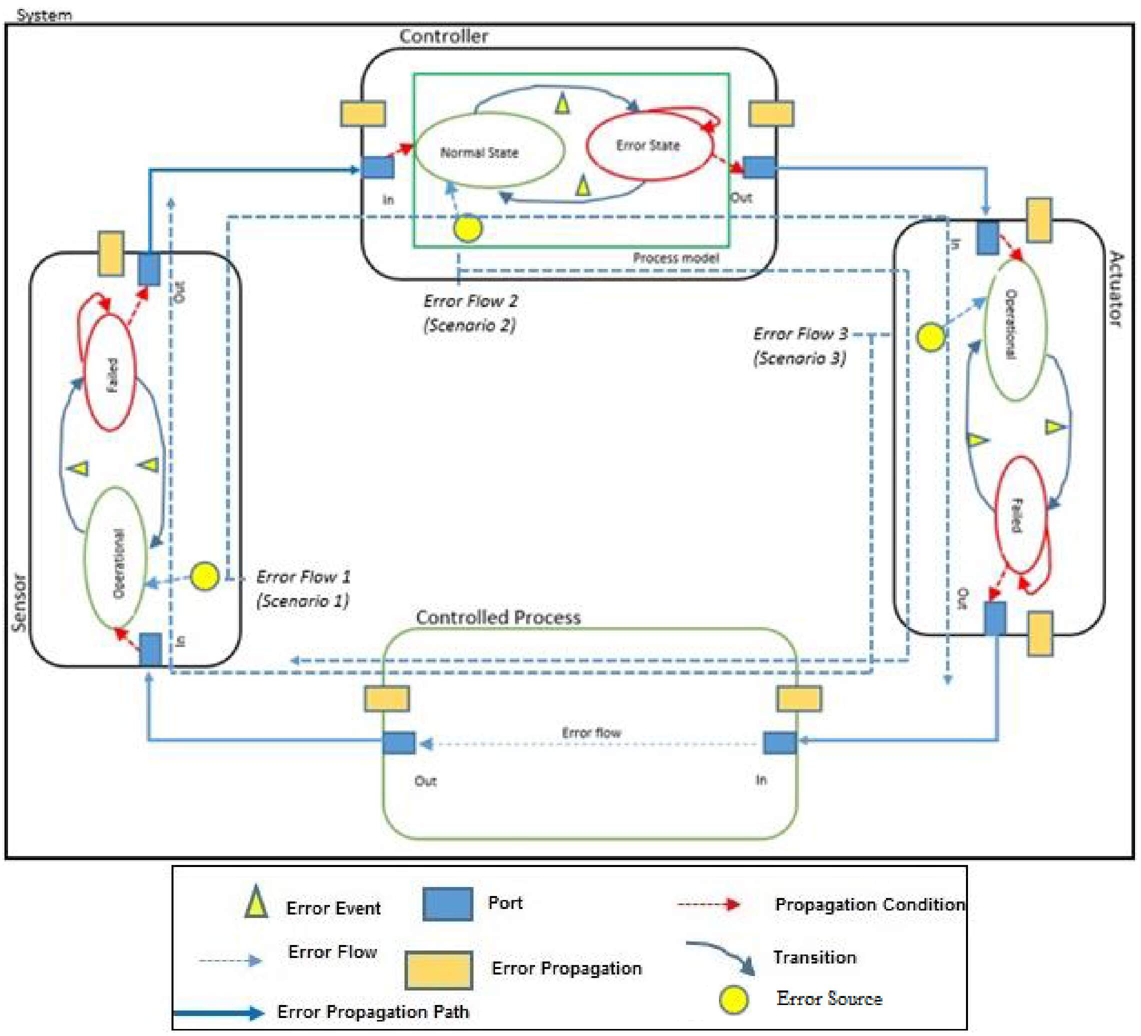


Figure 3.1: Feedback control loop augmented with error propagation and finite state machines

In the figure 3.1, we show three scenarios to demonstrate our method and to show the need for augmentation. We now detail each scenario.

3.1.1 Scenario 1:

Scenario 1 begins with the sensor in the operational state, and the failed state is entered when the sensor detects an internal failure. In this scenario, we demonstrate how the error propagates from the sensor to actuator and becomes hazardous to the system. Consider when the sensor detects an internal failure. The error continuously effects the normal operational state of the sensor. If the sensor is able to recover, the error will not propagate to the next component. But, if it does not recover, the error propagates through the outgoing port to the controller. The controller receives the error through its incoming port and processes it using the process model. In this case, the process model does not understand the propagated error. The propagated error then becomes an error in the controller. If the controller is able to handle the error, then no hazard would exist. The controller will automatically go back to its normal state. But, if the controller is not able to resolve the error, then a hazard would exist because the controller could possibly send a command to the actuator when it shouldn't. To diagnose the cause of this unsafe control action, we go back to the system controller because we expect the controller to be faulty due to its sending of commands when it should not. However, in this situation the controller does not have any faults. We need to go back another step to find the exact cause of the fault. In conclusion, using this scenario, we have shown that the propagated error from sensor to actuator has three important effects in the system. First, it effects the state of the sensor itself. Second, it effects the decision of the controller. Third, it effects the actuator.

3.1.2 Scenario 2:

Scenario 2 begins with the controller in the normal state, and the error state is entered when the controller detects an internal failure. The generated error does not propagate out; it stays within controller. Different types of errors propagate through the outgoing port to the actuator instead of the generated error. For example, the controller generates an (out of range) error, but it sends a (no data) error to the actuator because the controller is unable the out of range error. In this case, the controller becomes a source of errors. In this potentially hazardous situation, for each propagated error, the actuator is executing a different type of unsafe control action. For instance, if these errors are generated in the controller (i.e. stuck value error, out of range error, and out of calibration error) the actuator is doing an unacceptable action for the transformed error (i.e. no value, bad value, and incorrect value). After that, the controlled process receives the propagated error from the actuator as an inadequate or ineffective action through the error propagation path. The error then passes through the error flow inside of the controlled process to the outgoing port. Therefore, the output of the controlled process would be an incorrect, inaccurate or delayed measurement. For example, if the input to the controlled process is delayed, then it is likely its output would also be delayed. In conclusion, using this scenario, we have illustrated that the propagated error from the controller to the controlled process has three important effects on the system. First, it effects the controller decision. Second, the actuator performs an inadequate action. Third, it effects the controlled process by selecting the inappropriate process for the action. This illustration allows us to identify the source of the hazard by back-tracing for the error.

3.1.3 Scenario 3:

Scenario 3 begins with the actuator in the operational state, and the failed state is entered when the actuator detects an internal failure. The actuator receives a command from the controller through the incoming port, but the command is not executed as the controller intended because of the actuator's internal failure. Thus an error occurs in the actuator leading to the actuator entering the failed state. If the actuator is able to resolve the error, the controller's command will not be impacted. If the actuator is not able to resolve the error, a hazardous situation could result impacting the actuator's output, like causing the actuator to produce output late. The delayed operation directly effects the controlled process because the timing error passes through error flow to the output port. The output of the controlled process becomes an input to the sensor, but it has a feedback delay because of propagating the timing error. At the same time, the output of the sensor becomes an input to the controller, but it also has a feedback delay because of propagating the timing error. Therefore, the propagated error from the actuator to the sensor has three important effects on the system. First, it effects of the actuator state. Second, it effects of the controlled process by having a delay time to select the appropriate process for the action. Third, it effects of the sensor by having the delay time to obtain measured values.

3.2 Additional Scenarios to the Augmented Method

In this section, we provide additional scenarios that answer the following questions:

- How can an error outside of the feedback control loop effect a component's behavior inside of the loop? (Scenario 4)

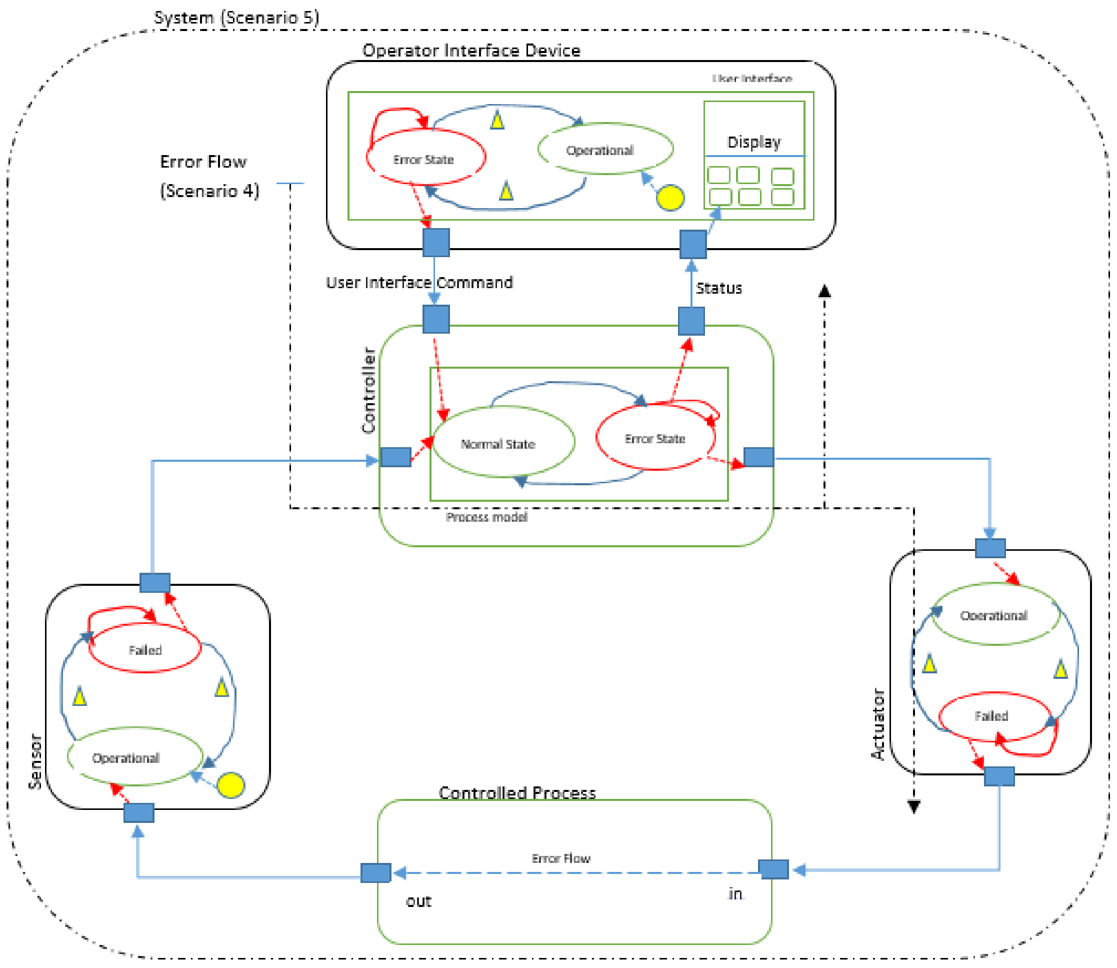


Figure 3.2: The augmented method for additional scenarios

- How can we identify the system states for each scenario? (Scenario 5)

Figure 3.2 will be used for the additional scenarios on the feedback control loop architecture. Consider the following scenarios:

3.2.1 Scenario 4:

As shown in figure 3.2, scenario 4 begins with a device which is connected to the feedback control loop. The device has a user interface which is used to set values of the controller and monitor the status of the system. The purpose of this scenario is that we wish to understand how the internal failure of the connected device effects the feedback control loop architecture and how errors propagate from devices external to the loop to components in the loop.

We begin with the operator interface device (OID) in the operational state, and the failed state is entered when the OID detects an internal failure. The error continuously effects the normal operational of the device. If the device is able to recover, the error will not propagate to the feedback control loop. But, if it does not recover, the error propagates through an outgoing port to the controller.

The controller receives the error through its incoming port processes it using the process model. But, the process model does not understand the propagated error. Because of this, the propagated error becomes an error event which may lead to the controller transitioning from the normal state to the error state. If the controller is able to resolve the error event, no hazard would be possible. But, if the controller is not able to resolve the error event, a harzardous event leading to an unsafe control action being sent to the actuator and the feedback control loop state becoming unsynchronized with the OID could occur, as shown in the figure 3.2.

In conclusion, using this scenario, we demonstrate the propagated error from the connected device has three important effects in the system. First, it effects the state of the connected device which leads to the sending of an ineffective/inadequate command to the controller. Second, it effects the decision of the controller. Third, it effects the actuator causing it to perform an ineffective/inadequate action.

3.2.2 Scenario 5:

As shown in figure 3.2, scenario 5 is used to find the system or top level, states of the feedback control loop according to the previous scenarios. As a reminder, we present each error flow based on three-way interactions back-tracing for the error. Each error flow could assist in identifying failure states of the system.

Scenario 1: We have this error flow ($Sensor \rightarrow Controller \rightarrow Actuator$), and we need to find out how this flow can show us the system states. Specifically, we need to provide a safety requirement for the failure state:

$$[(Sensor.FailedState) \lor (Controller.ErrorState) \lor (Actuator.FailedState)] \rightarrow System.FailedState \quad (3.1)$$

$$[(Sensor.OperationalState) \land (Controller.NormalState) \land (Actuator.OperationalState)] \rightarrow System.OperationalState \quad (3.2)$$

Scenario 2: We have this error flow ($Controller \rightarrow Actuator \rightarrow ControlledProcess$), and we need to find out how this flow can show us the system state. Specifically, we need to provide a safety requirement for the failure state:

$$[(Controller.ErrorState) \lor (Actuator.FailedState) \lor (ControlledProcess.ErrorState)] \rightarrow System.FailedState \quad (3.3)$$

$$[(Controller.NormalState) \land (Actuator.OperationalState) \land (ControlledProcess.NormalState)] \rightarrow System.OperationalState \quad (3.4)$$

Scenario 3: We have this error flow ($Actuator \rightarrow ControlledProcess \rightarrow Sensor$), and we need to find out how this flow can show us the system state. Specifically, we need to provide a safety requirement for the failure state:

$$[(Actuator.FailedState) \lor (ControlledProcess.ErrorState) \lor (Sensor.FailedState)] \rightarrow System.FailedState \quad (3.5)$$

$$[(Actuator.OperationalState) \land (ControlledProcess.NormalState) \land (Sensor.OperationalState)] \rightarrow System.OperationalState \quad (3.6)$$

Scenario 4: We have this error flow ($InterfaceDevice \rightarrow Controller \rightarrow Actuator$), and we need to find out how this flow can show us the system state. Specifically, we need to provide a safety requirement for the failure state:

$$[(InterfaceDevice.ErrorState) \lor (Controller.ErrorState) \lor (Actuator.FailedState)] \rightarrow System.FailedState \quad (3.7)$$

$$[(InterfaceDevice.NormalState) \land (Controller.NormalState) \land (Actuator.OperationState)] \rightarrow System.OperationalState \quad (3.8)$$

3.2.3 Notifications

Section 3.1 and its examples have been published as a technical paper in the 35th International System Safety Conference (ISSC) in Albuquerque, New Mexico, USA. The evaluation results of the scenarios are described in section 4.

3.3 Using Formal Notations to Augment a Hazard Analysis Method

As stated in section 1.3, the second goal of this research is:

- The introduction and analysis of a theoretical framework to identify unsafe system behavior.

The second goal is associated with a research question as stated in section 1.5:

- How to find more hazardous possibilities in the feedback control loop architecture?

The primary goal of augment the analysis processes of STPA is to perform a deeper analysis of system safety. We know that STPA identifies hazardous control actions based on the values of the process model variables and then changes the hazards into safety constraints for the controller. To identify hazards, each relevant value will be examined to determine the context of the control action because the controller will decide which action to perform based on the value. If one of the existing values in the process model is not compatible within the control action context that action can be considered an unsafe control action. From a software perspective, what will happen if the value/variable is not defined or missed in that context? Does this lead to missing safety constraints for the system? Finally, how does STPA resolve this problem? From our point of view, we need to know the specific information about each component in the feedback control loop such as functions, inputs, outputs, variables, and values. In addition, we need to know specifically how the process model in the controller makes a decision based on the existing values and variables.

Specifically, operation of the safety-critical system can be expressed as a function relating system inputs and outputs. There is a relationship among inputs and outputs where each output value will be related to an input value. When the system is working correctly, the system satisfies all functional requirements. This does not make the system safe. The system should be analyzed regarding the safety aspects and verified according to its safety requirements.

Our objective is to find new hazards and more safety constraints to further ensure safe. In this case, we need to identify safety constraints for unsafe actions because any unsafe behavior in the operation of the component. For this purpose, we now introduce the theoretical background of our analysis process. Our background consists of several functions describing the behavior of the feedback control loop, and by providing details of the functions, potential hazards can be identified and mitigated.

1. **Specify criteria to identify unsafe behavior of a component:** This step involves specifying an identifier to identify a faulty component in the feedback control loop and giving an error to the component. This step can be expressed formally as a two-tuple: *<INTF, Err>* where:

   - $< INTF >$ is faulty component identifier. This means a component will deliver incorrect service to the next component becoming an error source:

     $[INTF \in DC]$, where DC is the set of detection conditions.

   - $< Err >$ is the error. It is the result of an internal failure of a component. The error has an effect on the component's behavior. If the error propagates to the other components it may / may not lead to unsafe interaction among components.

     $[Err \in EO]$, where EO is the set of error ontology.

2. **Build notation to specify input/output/function for each component:** The purpose of this step is the construction of the feedback control loop with mathematical notation for describing the functional behavior of each component as is shown in figure 3.3

Controller - tuple: $< F_C(Vme, F_{PM}(Var, Val, Cond)) = CA >$, where:

- $< F_C >$ is the function of the controller. Normally, it takes an input from the sensor/interface device, processes it, and gives a command to the actuator.
- $< Vme >$ is the measured variable. It is the input to the controller. It is used to measure the value of the variable which is read by the sensor.

  $[Vme \in Var]$, where Var is the set of variables in process model. But if $[Vme \rightarrow Err]$, where $[Err \notin F_{PM}]$
- $< F_{PM} >$ is the process model function. It is the mental model of the controller. It can be represented as another component inside of the controller. It contains variables and their values. It is used to determine what control action is needed based on receiving content of the measured variables. It consists of three important elements:
  - $< Var >$ is the set of variables which are used by the process model.
  - $< Val >$ is the set of values that can be used by the variables.
  - $< Cond >$ is the set of conditions that can be used by the process model to make a decision, where: $[\{Var, Val\} \in Cond]$. The condition is that the received values via measured variable should be matched with the defined values in the process model function.
- $< CA >$ is the control action. It is the output of the controller. It can be considered as a command to actuator, where the $\{CA \rightarrow True\ iff\ Cond \rightarrow True\}$. This means the control action is going to be a safe action if and only if the condition is true. Otherwise, the control action is going to be inadequate action that can become an unsafe action.

Actuator-tuple:$< F_A(CA) = V_{Ma} >$ where:

- $< CA >$ is the control action, input to the actuator from the controller. If this is an expected control action, then the actuator will behave as expected. However, if this is an unsafe control action, the actions of the actuator might lead to a hazardous situation.
- $< Vma >$ is the manipulated variable, input to the controlled process from the actuator. The control action (CA), as previously mentioned, determines the output of the actuator.

Controlled Process-tuple: $< F_{CP}(V_{ma}) = Vc>$ where:

- $< Vma >$ is the manipulated variable, input to the controlled process. If the values are manipulated according to the controller's command, then the controlled process will be modified according to the limits defined in the controller.
- $< Vc >$ is the controlled variable, output of the controlled process. The actual behavior of the controlled process is monitored through the controlled variable which is dependent on input. If the input is in the range defined by the controller, the output will be in range as well.

Sensor-tuple: $< F_S(Vc) = Vme >$ where:

- $< Vc >$ is the controlled variable, input to the sensor, which is used to update information that produced by the sensor.
- $< Vme >$ is the measured variable, output of the sensor. The sensor sends information to the controller in the form of measured values. If the sensor has an internal failure, it will provide incorrect information to the controller. Additionally, the sensor can give incorrect information to the controller if the sensor updates its information using incorrect values that have been provided by the controlled processes through controlled variables.

Top-tuple: <*System*> where:

- $System$ is the set of sub-components that the top component decomposes into. If a component has an internal failure, how will the system react to it? Does the system have a safe behavior or will it react unsafely?

3. **Identify unsafe functional behavior and find more hazardous possibilities using the error ontology:** This step helps to identify unsafe functional behavior of a component that violates safety requirements for the associated system. The internal failure of a component can give inadequate results that could lead to an unsafe function. This step also helps to identify corresponding safety constraints which need to be created to mitigate unsafe functional behavior. In this step, we are using the error ontology to find the possibilities of unsafe functional behavior which leads to the system behaving incorrectly. If we find the hazardous possibilities before we develop the system, it helps to improve our confidence that the system will behave correctly when it faces hazardous conditions.
4. **Identify mathematical notation for each error flow:** This step identifies formal expressions for each error flow. We know that the propagated error cuts across three components. For that reason, each

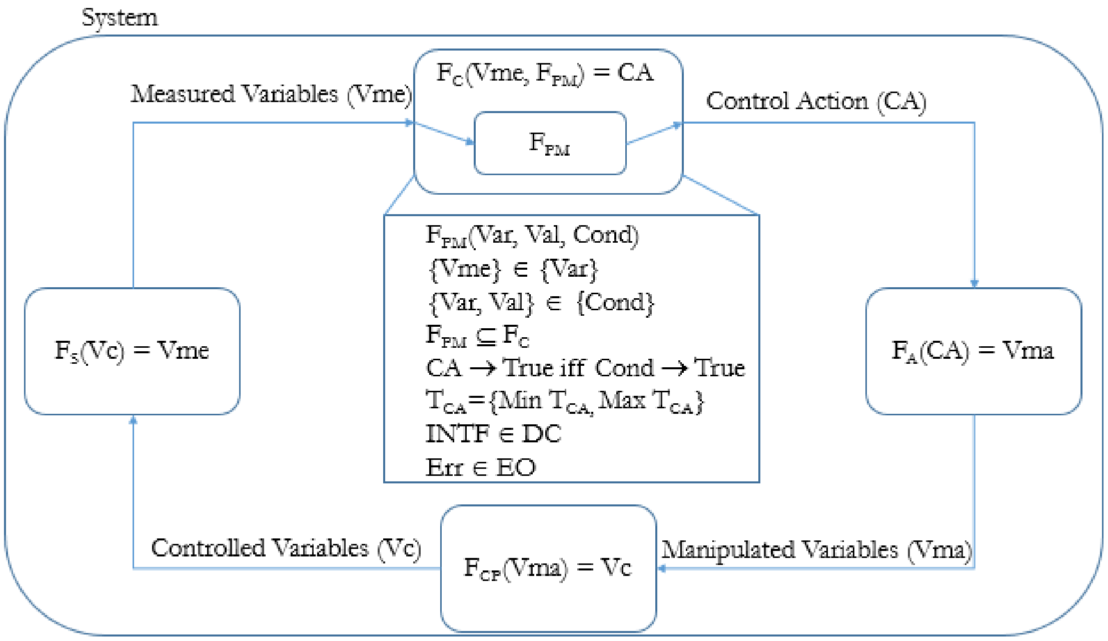


Figure 3.3: A framework for feedback control loop inputs/outputs and functions

error flow should contain three functions. In this case, the proposed safety constraints should cover three components to mitigate the effect of the error flow.

3.3.1 Notations and Expressions for Train Automated Door Control System

We take the same example described in appendix B which is an automated door controller for a train, and we apply our notations and expressions to find more hazardous possibilities:

1. **Specify criteria to identify unsafe behavior of a component:** Doors are locked. The train driver is unable to open them because the sensor provides incorrect readings to the controller. This leads to the passengers being stuck inside the train when the train is stopped at the station platform.

$$F_S(Vc) = \begin{cases} Vme & \text{if } Measured\ Values \in Vme; \\ Err & \text{if } Measured\ Values \notin Vme,\ where\ F_S(INTF) \rightarrow Err,\ Err \in EO. \end{cases}$$

The sensor sends measured values to the controller via measured variables. The controller is able to recognize the values that were previously defined in the variables (i.e measured variables). When the sensor has an internal failure, it will send different values to the controller. Then, the controller is not able to understand those values leading to the stuck door.

2. **Build notation to specify input/output and function for each component:** As shown in figure 3.3.

3. **Identify unsafe functional behavior and find more hazardous possibilities using the error ontology:** These functions are identified by predicting the behavior of the sensor and can help to determine the response of the controller.

Table 3.1: Unsafe Functional Behavior

Function	Sensor Function	Controller Function
Open Door	$F_S(Vc, INTF) = Err, assume \quad Err \in ErrValues$	$F_C(F_{PM}(Var, Val, Cond)) \neq CA, if \quad ErrValues \notin Val, Then \quad Cond \rightarrow False$

As shown in table 3.1, the function of the sensor shows the internal failure of the component and provides incorrect values to the controller (i.e the output of the sensor is an error (*Err*) which can be considered as one of the errors in the error ontology [like error values].) The function of the controller shows that the incoming error value is not an element in the defined values of the process model. For that reason, the decision of the process model is going to be unknown. This effects the output of the controller because the condition of the decision is false. Then, the output of the controller is not equal to the safe control action. As shown in table 3.2 by formalizing the error ontology, we can find more hazardous possibilities.

Table 3.2: Finding More Possibilities

Finding possibilities in controller	Description	Is this possibility hazardous?
If ErrValue $>$ MaxVal	The controller does not open the door when it gets above of the range values	Yes
If ErrValue $<$ MinVal	The controller does not open the door when it gets below of the range values	Yes
If ErrValue $\notin$ Val	The controller does not open the door when value error occur (none of the values)	Yes
If ErrValue = ”Null Value”	The controller does not open the door when it gets omitted error value	Yes
If ErrValue = ”Stuck Value”	The controller does not open the door when it gets the same repeated value	Yes

4. **Identify mathematical notation for each error flow:** As shown in table 3.3, we identify formal expressions for each error flow or scenario that was described in previous sections. This helps to identify safety constraints with respect to error contexts to mitigate the effects of the error on the system. For example, *SC(S, C, A, Err)* means the controller *(C)* is required to provide control action for the actuator *(A)* with respect to error type *(Err)* context for that message which comes from the sensor *(S)*.

Table 3.3: Mathematical Notations for Error Flows and Safety Constraints

Scenarios	**Error Flow Expressions**	**Safety Constraints (SC)**
1	$F_S(Vc, INTF) \rightarrow F_C \rightarrow F_A(Vma, Err)$	*SC(S, C, A, Err)*
2	$F_C(Vme, F_{PM}, INTF) \rightarrow F_A \rightarrow F_{CP}(Vc, Err)$	*SC(C, A, CP, Err)*
3	$F_A(CA, INTF) \rightarrow F_{CP} \rightarrow F_S(Vme, Err)$	*SC(A, CP, S, Err)*
4	$F_{OID}(Vme, INTF) \rightarrow F_C \rightarrow F_A(Vma, Err)$	*SC(OID, C, A, Err)*

3.3.2 Notifications

This section 3.3 and its example 3.3.1 have been published as a technical paper in the 36th International System Safety Conference (ISSC), Phoenix, Arizona, USA. In that session, 21 system safety researchers and practitioners are voted for 10 questions that have been prepared about the notations. The results of the votes are described in the appendix H.

3.4 Introducing Feedback Control Loop Architecture with Compositional Reasoning

This section can be considered the last step of the proposed method. The main reason behind doing this as part of the dissertation is that STPA has no method of verifying that the safety constraints proposed are correct. Thus our associated research question is How to introduce STPA with assume-guarantee contract to verify safety constraints? What are the steps to verify the safety constraints against system model?

System / software verification concentrates on showing functional correctness of the system and demonstrating that the system fully satisfies all functional requirements. However, the functional requirements do not make the system safe or reduce the risk level of the system. Therefore, the system should be analyzed with respect to safety aspects and verified against its safety requirements at component / system level.

The overall objective of this part of the dissertation is to demonstrate the possibility of verifying safety-critical software (i.e architecture model) against safety constraints which are derived at the component / system level. Generally, to control the criticality of safety critical software, we need to identify the potential hazards using the error ontology as a guide and then demonstrate that a potential hazard could not occur (i.e the software of the system cannot contribute to an unsafe state).

The contribution of this work is to introduce the feedback control loop architecture with safety verification procedures to verify the identified safety constraints and to show that hazards have been mitigated or controlled to an acceptable level of risk. In addition, our contribution could assist stakeholders with reducing the amount of time and effort by doing safety analysis and verification of the safety requirements on the same system architecture rather than doing them separately.

The procedure consists of the following steps:

1. **Identify an assume-guarantee contract for each component:** This step is the construction of a feedback control loop architecture with assumptions (A) describing expectations the component makes about the environment and guarantees (G) describing the behavior the component makes about its output if the assumptions are met.

2. **Identify mathematical notations for each component:** This step involves identifying expressions for each component based on the mathematical notations which are described in previous sections. For example, using F_S, F_C, F_A, F_{CP} as functions for the components in the implementation of the component, and using Vme, CA, Vma, Vc as input/output for components in the specification.

3. **Identify safe behavior for the top_level system:** This step assists in identifying guarantee of safe behavior in the overall system which depends on the guarantee of correct output of each component in the feedback control loop architecture. The notation inside the the figure 3.4 demonstrates this.

4. **Satisfy safety constraints:** This step assists in verifying the identified safety requirements against system model. We need to show that the model of the system satisfies the derived safety constraints. Previously, we built the safety constraints with regard to context of the error types. For example, our safety constraint, (The system should not accept null values), is identified as an omission error in the error ontology; it can also be mapped to one of the unsafe control actions in STPA which is (not

providing control action). If the system does not accept null values, it means the system satisfies the safety constraint against system model.

Consider figure 3.4 which introduces the feedback control loop architecture with compositional reasoning. The main advantage of this introduction is it mitigates the unsafe control action, the wrong decision of the controller that has been made based on the received incorrect values. To eliminate this action, we can make a guarantee for the output of the sensor, at the same time, constructing an assumption input for the controller. As a result, validate that the sensor does not send incorrect values to the controller leading to a safer system. This type of analysis can provide a certainty about behavior for the system and ensure that the system is not behaving incorrectly.

Table 3.4 helps to ensure the controller does not follow/execute the unsafe control actions because there is a contract among components such as sensor-controller, controller-actuator, actuator-controlled process, and controlled process-sensor. For example, we build the following safety constraints based error ontology context:

SC1: The controller has a guarantee to not receive null values from the sensor. (Omission error value)!! SC2: The controller has a guarantee to not receive unintended incoming values from the sensor. (Commission error value)!! SC3: The controller has a guarantee to do an action within defined time range. (Timing error)!!

When we verify the SC1, the controller mitigates one of the unsafe control actions which is [Action required but not provided] that can lead to hazards. In addition, when we verify SC2, the controller mitigates another unsafe control action which is [Unsafe action provided], and so on.

Table 3.4: Mitigating the unsafe control actions

STPA	**Error Ontology**	**A/G Contract**
Action required but not provided	Omission service error types	SC1
Unsafe action provided	Commission service error types	SC2
Action provided too early/too late	Service timing errors	SC3
Provided action stopped too soon/applied too long	Sequence commission (early start/late termination)	SC3

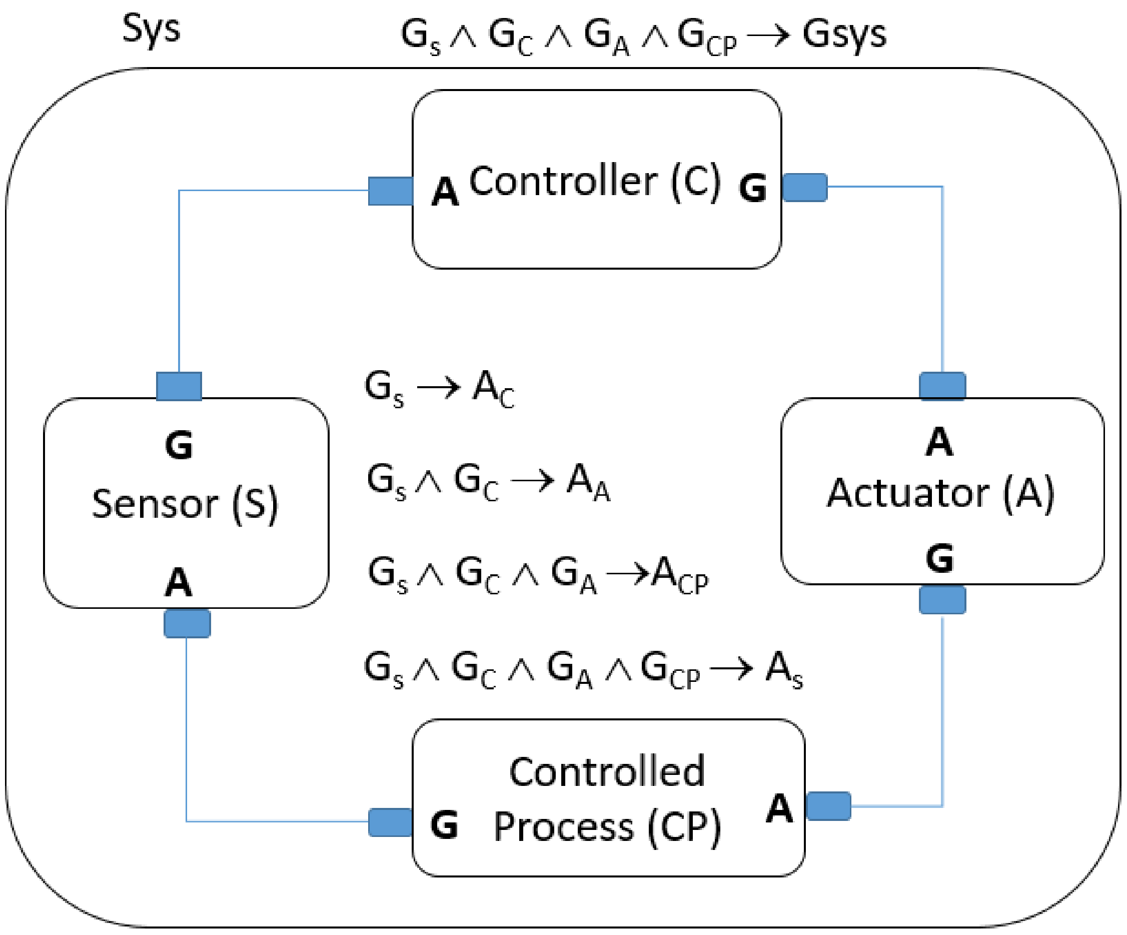

Sys
$G_s \wedge G_C \wedge G_A \wedge G_{CP} \rightarrow Gsys$
A Controller (C) G
G
Sensor (S)
A
A
Actuator (A)
G
$G_s \rightarrow A_C$
$G_s \wedge G_C \rightarrow A_A$
$G_s \wedge G_C \wedge G_A \rightarrow A_{CP}$
$G_s \wedge G_C \wedge G_A \wedge G_{CP} \rightarrow A_s$
Controlled
G Process (CP) A

Figure 3.4: Introduce feedback control loop architecture with assume-guarantee reasoning

Chapter 4

Analysis and Evaluation

In this section, we evaluate our compositional safety analysis method, Architecture Safety Analysis Method (ASAM), used for developing safety-critical systems. Our method will allow system stakeholders or safety analysts to mitigate the effects of errors through safety constraints developed during the construction of software architecture. In the method, we focus on finding safety constraints and verifying them within the system model. To do so, we inject the AADL error ontology into the architecture model to identify the unsafe control actions and causal factors of the unsafe control actions. We verify the safety constraints that we generate within the system model to ensure the hazardous situations are sufficiently mitigated through the use of compositional reasoning on the system model.

From a technical perspective, ASAM is a new software safety analysis tool which works with AADL models annotated with an implementation of our formulas (i.e. asam annex, introduced in section 3.3) and supported by the Open Source Architectural Tool Environment (OSATE). It is used to analyze and generate a report based on information attached to each component in the feedback control architecture. In fact, ASAM provides several statements to support discovering hazards such as error statements, error propagation statements, internal failure statements and safety constraint statements that either handle or prevent hazards. The goal of the statements is to describe errors in order to determine their source and prevent propagation. If error propagation cannot be prevented, ASAM can describe how the error is eventually handled as well as what component handles it. Additionally, ASAM lets the safety analyst record the severity level of the hazard for the specific error type based on the probability of occurrence for that type of error as well as verify that a safety constraint properly mitigates the error.

In this research, we focus on evaluating ASAM based on concerning claims:

Table 4.1: Evaluation Examples Overview

Safety-Critical System Examples	**C1**	**C2**	**C3**
Adaptive Cruise Control System -(ACCS)	4.1.2	4.2.2	–
Train Automated Door Control System-(TADCS)	4.1.3	4.2.3	4.3.2
Medical Embedded Device-Pacemaker (PM)	4.1.4	4.2.4	4.3.3
Infant Incubator Temperature Control System-Isolette (ISO)	4.1.5	–	–

- **Claim C1:** Whereas STPA only finds hazards that exist in the communication between the controller and the actuator, ASAM finds errors, that may or may not lead to hazards, that exist in not only in the controller / actuator but other components as well.
- **Claim C2:** Each error flow in ASAM assists in identifying a safety constraint of appropriate breadth to eliminate the unsafe control actions caused by an identified error. In addition, each error flow can guide us to identify specific causes for the unsafe control action.
- **Claim C3:** ASAM uses verification, unlike existing safety analysis methods, to further ensure that the safety constraint fully mitigates the hazardous condition improving the quality of the safety constraints.

Each claim is discussed individually in the remaining sections. Throughout the evaluation, we will use the safety-critical system examples that were described in the previous chapter such as the adaptive cruise control system (scenario 3.1.1), the train automated door control system (scenario 3.1.2), the pacemaker (scenario 3.1.3), and the isolette (scenario 3.2.1). For each of our selected examples, we will present the implementation of the scenarios, some of which will be reused across evaluations.

4.1 Demonstrating Three-Way Communication Format

Claim C1: Whereas STPA only finds hazards that exist in the communication between the controller and the actuator, ASAM finds errors, that may or may not lead to hazards, that exist in not only in the controller / actuator but other components as well.

4.1.1 Evaluation Plan

To demonstrate claim C1, we will evaluate each of our selected examples. For each system, we will first introduce the system and its architecture. We will then produce a report for that system under the described scenario. Each scenario evaluates an error flow based on three-way interaction format for the specific example to identify the source of the hazard.

ASAM shows the effect of unsafe interactions based on the three-way communication format among components while back-tracing the error. As a reminder, generally, the current accepted standard identifies unsafe interactions based on the two-way communication format. Specifically, it focuses on the interaction between the controller and actuator to identify unsafe control actions.

4.1.2 Adaptive Cruise Control System

The first system we introduce is an adaptive cruise control system for modern vehicles. The goal of this system is to monitor the distance and speed of the vehicle in front of the monitored vehicle (i.e. the system compares the speed of the monitored vehicle with the vehicle in front based on repeated measurements of the distance between them). This example is used to support (scenario 3.1.1) and show how our method helps to identify additional hazardous situations for an existing application example, described in the [22]. For demonstration purposes, we evaluate what will happen if the sensor in adaptive cruise control (ACC) estimates the incorrect values for speed and distance of the vehicle in front of the monitored vehicle (e.g., the sensor estimates close enough, but in reality is not) during driving because of an internal failure. When the component has an internal failure, the result is propagation of an error. The error values propagate from the sensor to the controller causing the controller to process the erroneous value. This will cause the controller to send an incorrect command to the actuator. For this situation, the ACC system warns the driver to apply the brake because the monitored vehicle is close enough with the vehicle in front. If the driver does not apply the brake, the system automatically will do it. Either way, an accident will happen because the monitored vehicle is doing an unacceptable action based on a decision made with incorrect values. In this case, either the driver or the system will apply the brakes in the middle of the road possibly causing the monitored vehicle to be hit by the vehicle behind it.

The initial results for this ACC system example have been published in [38] which are:

1. The error ontology identifies wrong estimation as incorrect values. The hazard is ”incorrect estimation values for the monitored vehicle from the ACC system”.

2. The feedback control loop system has been built from the sensor to the actuator as shown in Figure 3.1 which is error flow 1 (scenario 3.1.1).

3. The unsafe control action is that ”The system applies the brakes in the middle of the road”. The safety constraint for the unsafe control action is that ”The system must not apply the brakes when it has incorrect values”.

4. The general cause for the unsafe control action is that ”The sensor has an internal failure” and the specific cause is that ”The propagating of incorrect values causes an error event from sensor to actuator”.

5. This is shown in the developed safety architecture subsection, as shown in figure 3.1.

We will now provide and discuss the individual core components of the feedback control loop architecture for ACC system.

```
system ACC_sensor
    features
        ACC_Speed_s: out data port;
        ACC_Distance_s: out data port;
        ACC_State_s: in data port;
    flows
        f_source1: flow source ACC_Speed_s;
        f_source2: flow source ACC_Distance_s;
end ACC_sensor;
```

Listing 4.1: ACC System

In listing 4.1, we show the first primary component of the architecture: the interface for the ACC sensor. This interface shows that the *ACC_sensor* component produces 2 values and receives 1 value. The two outputs are a *ACC_Speed_s* and a *ACC_Distance_s* value that represent the speed and distance of the in front of car. These two values also represent the source of information flow. The final value *ACC_State_s* is the controlled process state. The implementation of this interface is shown in listing 4.2.

```
system implementation ACC_sensor.impl
    annex asam {**
        Errs => [{
            --Identify specific error types for the sensor, identify chance of
                harming people.
            Type =>AboveRangeSpeed(Critical,p=0.1),
            --For the specific error type, what kind of unsafe action will the
                sensor perform?
            UCA => UCA1: "ACC sensor dispatches above range of speed values to
                controller",
            --What are the causes of the Unsafe Control Action(UCA) of the
                sensor?
            Causes => {
                General => "ACC sensor has an internal failure",
                Specific => "ACC sensor receives above range of speed value
                    from feedback"
            },
            --Identify Safety Constraints (SC) for the sensor to mitigate
                effect of the UCA.
            SC => SC1: "ACC sensor should not send incorrect speed values to
                controller for computation"
        },
        {
            Type =>BelowRangeDistance(Catastrophic,p=0.1),
            UCA => UCA2: "ACC sensor dispatches below range of distance values
                to controller",
            Causes => {
                General => "ACC sensor has an internal failure",
                Specific => "ACC sensor receives below range distance value
                    from feedback"
            },
            SC => SC2: "ACC sensor must not dispatch incorrect distance values
                to controller for computation"
        }]
    **};
end ACC_sensor.impl;
```

Listing 4.2: ASAM for ACC System Sensor

In listing 4.2, the ASAM annex allows stakeholders to record specific information in the implementation part of each component in the ACC architecture model. For example, we have recorded the following information in the ACC sensor's implementation: error *(Err)* types for the sensor's internal failure (above range of speed and below range of distance), unsafe control actions *(UCA)* for each error type, general or specific causes for each unsafe control action, probability of occurrence *(P)* for each error type, severity level of hazards for each error type, and safety constraints *(SC)* to mitigate the effects of the error types or unsafe control actions.

```
system ACC_controller
    features
        ACC_Speed_c: in data port;
        ACC_Distance_c:in data port;
        ACC_Speed_cmd:out data port;
        ACC_Ditance_cmd: out data port;
    flows
        f_path1: flow path ACC_Speed_c -> ACC_Speed_cmd;
        f_path2: flow path ACC_Distance_c -> ACC_Ditance_cmd;
end ACC_controller;
```

Listing 4.3: ACC System

In listing 4.3, we have the *ACC_controller* interface that controls the speed and distance of the vehicle. This interface has two in data ports, *ACC_Speed_c* and *ACC_Distance_c*, that allow the controller to receive speed and distance values from the sensor. The interface also has two out data ports:*ACC_Speed_cmd* and *ACC_Ditance_cmd*. *ACC_Speed_cmd* sends the speed command to actuator and *ACC_Ditance_cmd* sends the distance command to actuator. The interface has a flow path to make a link between input and output ports. For example, *f_path1* is a flow path used to transfer speed information within the controller from input port *ACC_Speed_c* to output port *ACC_Speed_cmd*. Also, *f_path2* is another flow path used to pass distance information within the controller from *ACC_Distance_c* to *ACC_Ditance_cmd*. The implementation of this interface is shown in listing 4.4.

```
system implementation ACC_controller.impl
    annex asam {**
        Errs => [{
            Type =>AboveRangeSpeed(Critical,p=0.0003),
            UCA => UCA3: "ACC controller warns the driver to apply the brakes",
            Causes => {
                General => "ACC controller has an internal failure",
                Specific => "ACC controller receives above range of speed value
                    from sensor"
            },
            SC => SC3: "ACC controller should not do a computation when it has
                incorrect speed values"
        },
        {
            Type =>BelowRangeDistance(Catastrophic ,p=0.0003),
            UCA => UCA4: "ACC controller sends command to apply the brakes in
                the middle of the road",
            Causes => {
                General => "ACC controller has an internal failure",
                Specific => "ACC controller receives below range distance value
                    from feedback"
            },
            SC => SC4: "ACC controller must not apply the brakes when it has
                incorrect distance values"
        }]
    **};
end ACC_controller.impl;
```

Listing 4.4: ASAM for ACC System Controller

In listing 4.4, we have recorded two incoming errors from the ACC sensor such as above range speed and below range distance. For the first error, above range speed, the controller warns the driver to apply the brakes because the speed of your monitored vehicle is higher than the vehicle in front. The error is estimated by the sensor incorrectly and propagated to the controller through the nominal control flow as well as the error propagation path. In this case, the action of controller is going to be an unsafe action. The causes for that action can be classified into two types: 1) a general cause, ACC controller has an internal failure and 2) a specific cause, ACC controller received incorrect speed values from the sensor. To mitigate the effects of that unsafe action, we provide a safety constraint: ACC controller should not do a computation when it has incorrect speed values. We have attached a safety requirement for that error to know how to correct the error if it happens. Also, we have recorded the probability of occurrence for above range speed error and its hazard severity level.

For the second error, below range distance, the ACC controller sends a command to apply the brakes in the middle of the road because your car is too close to the car in front. The error is estimated by the sensor incorrectly and propagated to the controller. In this case, the action of controller is going to be an unsafe control action. The causes for that action can be divided into two types: 1) a general cause, ACC controller has an internal failure, and 2) a specific cause, ACC controller received incorrect distance values from the sensor. We provide a safety constraint to mitigate the effects of the unsafe actions: ACC controller must not apply the brakes when it has incorrect distance values. We have attached a safety requirement for that error which helps to know how to correct the error if it happens. Also, we have recorded the probability of occurrence for below range distance error and its hazard severity level.

```
system ACC_actuator
    features
        ACC_EXE_Speed_a: in data port;
        ACC_EXE_Distance_a: in data port;
        Unsafe_Action_a: out data port;
    flows
        f_path3: flow path ACC_EXE_Speed_a->Unsafe_Action_a;
        f_path4: flow path ACC_EXE_Distance_a->Unsafe_Action_a;
end ACC_actuator;
```

Listing 4.5: ACC System

In listing 4.5, we have the *ACC_actuator* interface that follows the controller's instructions to execute commands. This interface has two in data ports, *ACC_EXE_Speed_a* and *ACC_EXE_Distance_a*, that allow the actuator to execute speed and distance commands from the controller. The interface also has an out data port, *Unsafe_Action_a*, that allows the actuator to send commands to the controlled process. In this case, inadequate commands from controller lead to unsafe commands from the ACC actuator. The interface also has a flow path to make a link among in and out data ports. For example, *f_path3* is a flow path within the actuator used to transfer data from *ACC_EXE_Speed_a* to *Unsafe_Action_a*. Also, *f_path4* is another flow path within the actuator used to transfer data from *ACC_EXE_Distance_a* to *Unsafe_Action_a*. The implementation of this interface is shown in listing 4.6.

```
system implementation ACC_actuator.impl
    annex asam {**
        Errs => [{
            Type =>AboveRangeSpeed(Critical ,p=0.2),
            UCA => UCA5: "ACC actuator executes the 'warn the driver' command",
            Causes => {
                General => "ACC actuator has an internal failure",
                Specific => "ACC actuator receives inadequate command from
                    controller"
            },
            SC => SC5: "ACC actuator should not execute the incorrect command"
        },
        {
            Type =>BelowRangeDistance(Catastrophic ,p=0.3),
            UCA => UCA6: "ACC actuator executes the 'apply brakes' command",
            Causes => {
                General => "ACC actuator has an internal failure",
                Specific => "ACC actuator receives an incorrect command from
                    controller"
            },
            SC => SC6: "ACC actuator must not execute the inadequate command"
        }]
    **};
end ACC_actuator.impl;
```

Listing 4.6: ASAM for ACC System Actuator

In listing 4.6, we have two incoming commands from the ACC controller to the ACC actuator through two in data ports: *ACC_EXE_Speed_a* and *ACC_EXE_Distance_a*. But, each command contains an error. For example, there is an above range of speed error in *ACC_EXE_Speed_a*, and a below range of distance error in *ACC_EXE_Distance_a*. For the first error, above range of speed, the actuator executes the 'warn the driver' command because the actuator has an internal failure or received an inadequate command from the controller. For the second error, below range of distance, the actuator executes the 'apply brakes' command in the middle of the road because the actuator has an internal failure or receives an inadequate command from controller. To know how to mitigate the effects of the errors, we have attached safety constraints which help mitigate incorrect or inadequate commands.

```
system ACC_controlled_process
    features
        Controlled_Action: in data port;
        Feedback: out data port;
    flows
        f_sink: flow sink Controlled_Action;
end ACC_controlled_process;
```

Listing 4.7: ACC System

In listing 4.7, we show the last interface, *ACC_controlled_process*. This interface gives a *Feedback* value to the sensor through the out data port. The interface also receives *Controlled_Action* through an in data port from the actuator. The flow sink represents the end of the information flow which comes from the sensor.

```
system implementation entire_system.impl
    subcomponents
        ACC_sensor: system ACC_sensor.impl;
        ACC_controller: system ACC_controller.impl;
        ACC_actuator: system ACC_actuator.impl;
        ACC_controlled_process: system ACC_controlled_process.impl;
    connections
        conn_1: port ACC_sensor.ACC_Speed_s -> ACC_controller.ACC_Speed_c;
        conn_2: port ACC_sensor.ACC_Distance_s -> ACC_controller.ACC_Distance_c;
        conn_3: port ACC_controller.ACC_Speed_cmd ->
            ACC_actuator.ACC_EXE_Speed_a;
        conn_4: port ACC_controller.ACC_Ditance_cmd ->
            ACC_actuator.ACC_EXE_Distance_a;
        conn_5: port ACC_actuator.Unsafe_Action_a ->
            ACC_controlled_process.Controlled_Action;
        conn_6: port ACC_controlled_process.Feedback -> ACC_sensor.ACC_State_s;
    flows
        etoef_speed: end to end flow
            ACC_sensor.f_source1->conn_1->ACC_controller.f_path1
        ->conn_3->ACC_actuator.f_path3->conn_5->ACC_controlled_process.f_sink;
        etoef_distance: end to end flow
            ACC_sensor.f_source2->conn_2->ACC_controller.f_path2
            ->conn_4->ACC_actuator.f_path4->conn_5->ACC_controlled_process.f_sink;
end entire_system.impl;
```

Listing 4.8: ACC System

In listing 4.8, we have represented the core components of the entire ACC system architecture. We have represented the connections among components. We have also represented the end to end flow information which goes from the sensor to the actuator.

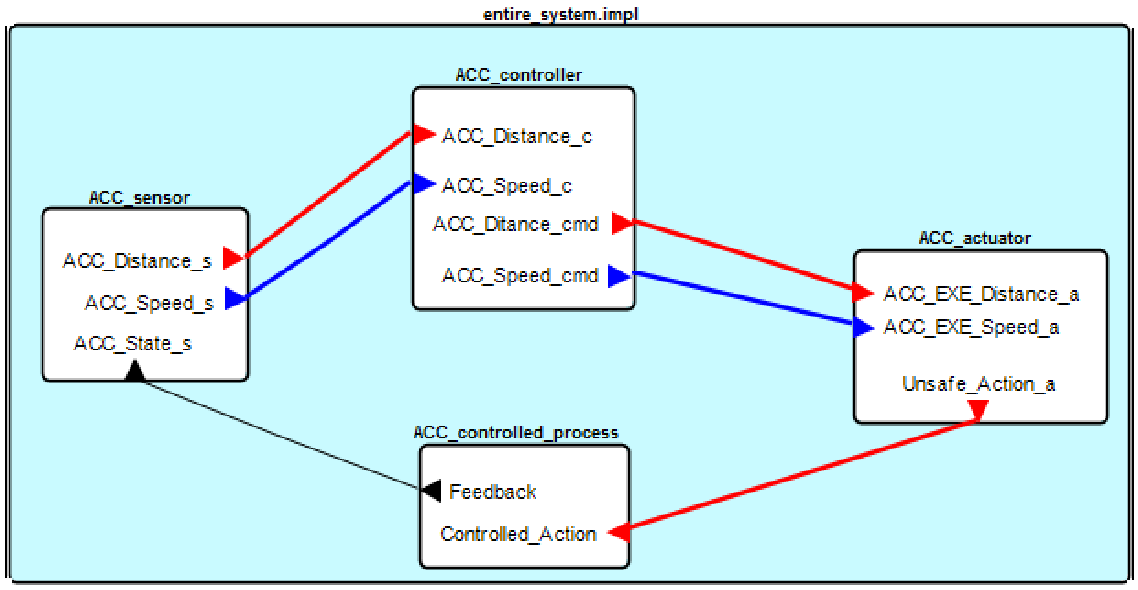
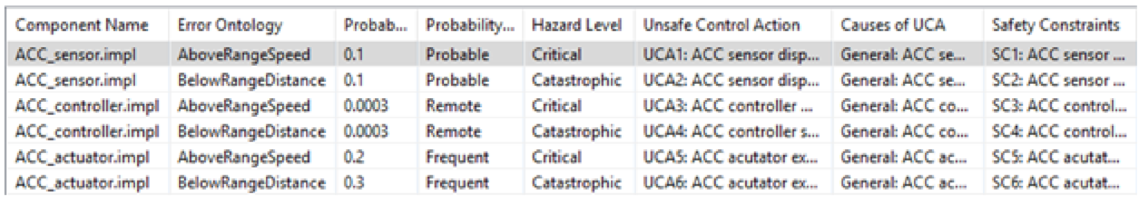


Figure 4.1: Three-Way Communication Format for Adaptive Cruise Control System Architecture

Figure 4.1, a feedback control loop architecture has been built for the ACC system. Figure 4.1 shows the error flow for each error type based on the three-way interaction format. Figure 4.1 shows two error flows: one for the distance error and the other for the speed error. The first error flow (i.e., red line) shows the error propagation from *ACC_sensor* to *ACC_actuator*. The error in the measuring of distance propagates to the *ACC_controller* through the output data port. The error distance value in the sensor can be prevented from being communicated through a safety constraint statement. But, if it is not prevented, it propagates to the controller. The controller also can handle the incoming error distance value through the in data port according to the safety constraint. If the controller is not able to handle that, it sends an inadequate command to the actuator. Then, the actuator will execute an inadequate action. The effect of that error changes the manipulated variable which is the output of the actuator. The second error flow (i.e. blue line) has the same description.

ASAM generates a report for the adaptive cruise control system architecture based on component names, error ontology, probability values, probability thresholds, hazard levels, unsafe control actions, causes of unsafe control actions and safety constraints. This stream of information is attached to each imple-



Component Name	Error Ontology	Probab...	Probability...	Hazard Level	Unsafe Control Action	Causes of UCA	Safety Constraints
ACC_sensor.impl	AboveRangeSpeed	0.1	Probable	Critical	UCA1: ACC sensor disp...	General: ACC se...	SC1: ACC sensor ...
ACC_sensor.impl	BelowRangeDistance	0.1	Probable	Catastrophic	UCA2: ACC sensor disp...	General: ACC se...	SC2: ACC sensor ...
ACC_controller.impl	AboveRangeSpeed	0.0003	Remote	Critical	UCA3: ACC controller ...	General: ACC co...	SC3: ACC control...
ACC_controller.impl	BelowRangeDistance	0.0003	Remote	Catastrophic	UCA4: ACC controller s...	General: ACC co...	SC4: ACC control...
ACC_actuator.impl	AboveRangeSpeed	0.2	Frequent	Critical	UCA5: ACC acutator ex...	General: ACC ac...	SC5: ACC acutat...
ACC_actuator.impl	BelowRangeDistance	0.3	Frequent	Catastrophic	UCA6: ACC acutator ex...	General: ACC ac...	SC6: ACC acutat...

Figure 4.2: ASAM's report for Adaptive Cruise Control System Architecture

mentation part of the error flow (flow source [*ACC_sensor.impl*], flow path[*ACC_controller.impl*], and flow sink[*ACC_actuator.impl*]). The report is shown in figure 4.2. That analysis shows the source of faults in the ACC model, which is the sensor. Each fault in the sensor produces errors. Each error has a chance to harm people based on the probability of occurrence for that error and result of that error. For example, an "above range of speed" error results in a critical level and a "below range of distance" error results in a catastrophic level. Each error in the sensor gives an unsafe control action to the controller which results in sending an inadequate command to the actuator. For example, the ACC controller warns the driver to apply the brake. If the driver does not do that action, the system will automatically do it. Also, general and specific causes for each unsafe control action have been identified. Finally, we have safety constraints to know how to mitigate the effects of the unsafe control actions or the errors.

4.1.3 Train Automated Door Control System

The second system we introduce is a train automated door control system. The goal of this system is to monitor the open and close events of the train door. This example is used to support (scenario 2 3.1.2). The train door example is described in [22]. We need to extend the same example to improve safety of the system from better to best. For instance, what will happen if train automated door controller sends inadequate commands instead of correct commands because of internal failure? Will it select the correct doors to open, and only open those doors when the train is stopped completely at the station platform? If the controller has an internal failure, it produces an error. The error becomes an event leading to changing the normal operational state of the process model to the error state. If the controller is able to handle the error event, a hazardous state will not occur. The controller will go back to its normal state. But, if the controller is not able to handle the error event, the error will be transformed into an outgoing propagation error from controller to actuator. As the result, the actuator is will perform an incorrect action. For example, it might select the right side doors processes in the controlled process to execute instead of left side doors. This can definitely lead to a hazardous situation because the system guides people in the wrong direction.

The initial result for this train automated door control system example has been published in [38] which are:

1. The error ontology identifies wrong selection as incorrect values. Now, the hazard is ”wrong side selection to open the wrong doors at the station platform”.

2. The feedback control loop has been built from the controller to the controlled process as shown in figure 3.1 which is error flow 2 (scenario 2).

3. The unsafe control action is that ”system selects right side doors to open instead of left side”. The safety constraints for the unsafe action is that ”the system must not open the doors when it has incorrect values”.

4. The general cause for the unsafe control action is that ”the controller has an internal failure” and the specific cause is that ”the propagating error event from the door controller to the physical door”.

5. This is shown in the developed safety architecture subsection, as shown in figure 3.1.

We will now provide and discuss the individual core components of the feedback control loop architecture for train automated door control system.

```
system Door_controller
    features
        door_open_c: in data port;
        door_blocked_c: in data port;
        open_door_c: out data port;
        close_door_c: out data port;
    flows
        c_source1: flow source open_door_c;
        c_source2: flow source close_door_c;
end Door_controller;
```

Listing 4.9: Train Automated Door Control System

In listing 4.9, we have the *Door_controller* interface that opens and closes the train door when it is required. This interface has two in data ports, *door_open_c* and *door_blocked_c*, that allow the controller to receive open door state and blocked door state values. The interface also has two out data ports: *open_door_c* and *close_door_c*. *open_door_c* is used to send the open door command to the physical door and *close_door_c* is used to send the close door command to the physical door as well. The interface has two flow sources of information, *c_source1* and *c_source2*. The *Door_controller* sends open door commands through *c_source1*, and sends close door commands through *c_source2*. The implementation of this interface is shown in listing 4.10.

```
system implementation Door_controller.impl
    annex asam {**
        Errs => [{
            --Identify specific error types for the door controller, identify
                chance of harm people.
            Type =>ServiceCommission(Critical,p=0.1),
            --For the specific error type, what kind of unsafe action will the
                door controller do?
            UCA => UCA1: "Door controller selects right side doors to open
                instead of left side",
            Causes => {
                --What are the causes of the Unsafe Control Action(UCA) of the
                    door controller?
                General => "The door controller has an internal failure",
                Specific => "The propagating error event from the sensor to
                    door controller"
            },
            --Identify Safety Constraints (SC) for the door controller to
                mitigate effects of the UCA.
            SC => SC1: "The door controller must not open the doors when
                unexpected service or data is provided for analysis"
        }]
    **};
end Door_controller.impl;
```

Listing 4.10: ASAM for Train Door Controller

In listing 4.10, we have identified a specific error type, the service commission, which represents unexpected service provided by the controller because of the controller's internal failure. For that type of error, we have also identified the probability of occurrence for the error and the hazard's severity level. In addition, we have recorded the unsafe action of the door controller for that type of error: Door controller selects right side doors to open instead of left side. The general cause for that unsafe action is the controller's internal failure. The specific cause is propagating an error from the sensor of the door to controller. We also have identified a safety constraint, the controller should not do any analysis for unintended incoming data, to mitigate the effects of the controller's unsafe action within the model.

```
system Door_actuator
    features
        open_door_a: in data port;
        close_door_a: in data port;
        door_state_a: out data port;
    flows
        a_path1: flow path open_door_a-> door_state_a;
        a_path2: flow path close_door_a-> door_state_a;
end Door_actuator;
```

Listing 4.11: Train Automated Door Control System

In listing 4.11, we have the *Door_actuator* interface that follows the door controller's to execute commands. The interface has two in data ports, *open_door_a* and *close_door_a*, that allow the door actuator to execute open and close commands from the door controller. The interface also has an out data port, *door_state_a*, that allows the door actuator to show the physical door status after executing open or close commands. The interface also has two flow paths to make a link between the in and out data ports within door actuator. For example, *a_path1* is a flow path within the actuator used to transfer data from *open_door_a* to *door_state_a*. Also, *a_path2* is another flow path within the actuator used to transfer data from *close_door_a* to *door_state_a*. The implementation of this interface is shown in listing 4.12.

```
system implementation Door_actuator.impl
    annex asam {**
        Errs => [{
            --Identify specific error types for the door actuator, identify the
                chance of harming people.
            Type =>ServiceCommission (Critical,p=0.002),
            --For the specific error type, what kind of unsafe action will the
```

```
                door actuator do?
            UCA => UCA2: "Door actuator follows the door controller's commands
                which opens the wrong side door",
            Causes => {
            --What are the causes of the Unsafe Control Action(UCA) of the door
                actuator?
                General => "The door actuator has an internal failure",
                Specific => "The door actuator received inadequate command from
                    door controller"
            },
            --Identify Safety Constraints (SC) for the door actuator to
                mitigate effect of the UCA.
            SC => SC2: "The door actuator must not execute incorrect commands"
        }]
    **};
end Door_actuator.impl;
```

Listing 4.12: ASAM for Train Door Actuator

In listing 4.12, we have identified that the actuator opens the wrong doors. That's because of two reasons: 1) the door actuator receives an inadequate command from the controller or 2) the door actuator has an internal failure. In this case, we need to provide a safety constraint to know how to mitigate the effects of actuator's unsafe actions.

```
system Physical_Door
    features
        door_state_in_cp: in data port;
        door_state_out_cp: out data port;
    flows
        cp_path3: flow path door_state_in_cp -> door_state_out_cp;
end Physical_Door;
```

Listing 4.13: Train Automated Door Control System

In listing 4.13, we show the physical door interface in the train door system architecture. The interface has an in data port, *door_state_in_cp*, that allows it to take the executed commands from door actuator.

Also, the interface has an out data port, *door_state_out_cp*, that allows it to show the status of executed process that has been selected by the door controller. The data of the executed process automatically passes within the physical door interface from *door_state_in_cp* to *door_state_out_cp*. The implementation of this interface is shown in listing 4.14.

```
system implementation Physical_Door.impl
    annex asam {**
        Errs => [{
            --Identify specific error types for the physical door, identify the
                chance of harming people.
            Type =>ServiceCommission (Marginal,p=0.00001),
            --For the specific error type, what kind of unsafe action will the
                physical door do?
            UCA => UCA3: "The wrong process has been selected to execute",
            Causes => {
                --What are the causes of the Unsafe Control Action(UCA) of the
                    physical door?
                General => "The door controller's wrong decision",
                Specific => "The door actuator's wrong execution"
            },
            --Identify Safety Constraints (SC) for the physical door to
                mitigate effect of the UCA.
            SC => SC3: "The physical door should not allow the selection of the
                wrong process"
        }]
    **};
end Physical_Door.impl;
```

Listing 4.14: ASAM for Physical Door Interface

In listing 4.14, the physical door implementation tells us the wrong process has been executed because of two reasons: 1) the wrong decision has been made by the door controller or 2) the wrong execution has been made by the door actuator. We provide a safety constraint, the physical door should not allow the selection of the wrong process. This helps to mitigate the effects of the unsafe action of the physical door. Finally, this case tells us that the open or close processes in the physical door have been updated by abnormal data because of a service commission error which comes from the controller. We know the physical door passes the wrong updated process to sensor of the door. This means the sensor's information will be updated incorrectly.

```
system Door_sensor
    features
        door_state_s: in data port;
        door_open_s: out data port;
        door_blocked_s: out data port;
    flows
        s_sink: flow sink door_state_s;
end Door_sensor;
```

Listing 4.15: Train Automated Door Control System

In listing 4.15, we show the last component of the train automated door control system architecture (TADCS): the interface for the TADCS sensor. This interface shows that the *Door_sensor* component receives one value and produces two values. The first received value *door_state_s* represents door state. The other two values are *door_open_s* and *door_blocked_s*, representing door open state and door blocked state. The flow sink represents the end of the information flow which comes from the *Door_controller*. The implementation of the entire system is shown in listing 4.16.

```
system implementation entire_system.impl
    subcomponents
        Door_sensor: system Door_sensor.impl;
        Door_controller: system Door_controller.impl;
        Door_actuator: system Door_actuator.impl;
        Physical_Door: system Physical_Door.impl;
    connections
        conn_1: port Door_sensor.door_open_s -> Door_controller.door_open_c;
        conn_2: port Door_sensor.door_blocked_s ->
            Door_controller.door_blocked_c;
        conn_3: port Door_controller.open_door_c -> Door_actuator.open_door_a;
        conn_4: port Door_controller.close_door_c -> Door_actuator.close_door_a;
        conn_5: port Door_actuator.door_state_a ->
            Physical_Door.door_state_in_cp;
        conn_6: port Physical_Door.door_state_out_cp ->
            Door_sensor.door_state_s;
    flows
        etoef_open: end to end flow
            Door_controller.c_source1->conn_3->Door_actuator.a_path1
        ->conn_5->Physical_Door.cp_path3->conn_6->Door_sensor.s_sink;
        etoef_close: end to end flow
            Door_controller.c_source2->conn_4->Door_actuator.a_path2
        ->conn_5->Physical_Door.cp_path3->conn_6->Door_sensor.s_sink;
end entire_system.impl;
```

Listing 4.16: Train Automated Door Control System

In listing 4.16, we show the core components of the entire train automated door control system architecture. We represent the connections among components and we also represent the end to end flow information which comes from the door controller to the controlled process (i.e.physical door).

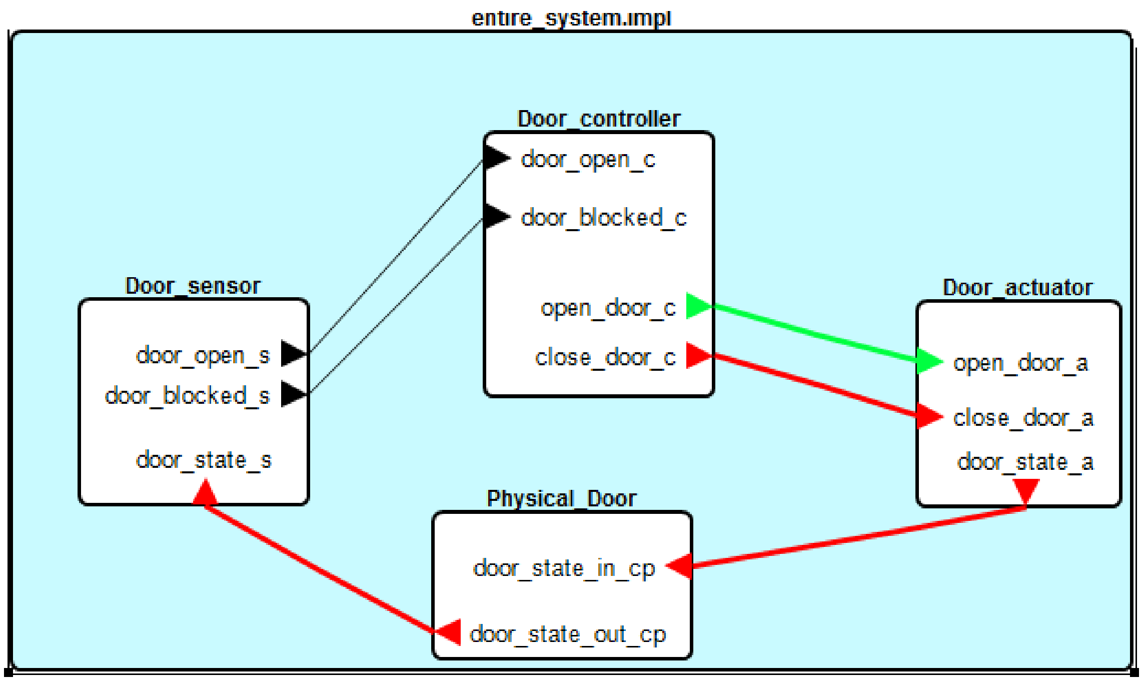
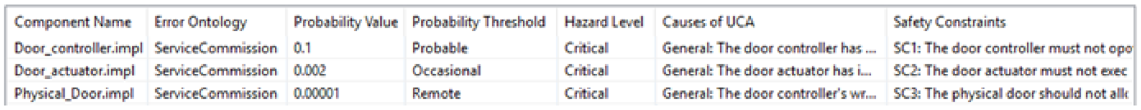


Figure 4.3: Three-Way Communication Format for Train Automated Door Control System Architecture

Figure 4.3 shows the feedback control loop architecture which has been built for the train automated door control system. Figure 4.3 shows error flows based on the three-way interaction format for that train door model. The *Door_controller* has an internal failure. This leads to the production of a service commission error (i.e. door controller closes the door instead of open). That error propagates to the *Door_actuator* in the form of inadequate command. Then, the actuator executes the command which contains the error because the actuator follows the *Door_controller*'s order. The *Physical_door* shows the process has been executed by the door actuator and updates that process incorrectly. Finally, we show the propagated error which cuts across three components, from *Door_controller* to *Door_actuator*, and from *Door_actuator* to *Physical_door*. Also, the propagated error has three important effects on the train automated door control system. First, it effects the *Door_controller* decision. Second, the *Door_actuator* performs inadequate action. Third, it effects the controlled process *Physical_door* by selecting the inappropriate process for the action. This illustration allows us to identify the source of the hazard by back-tracing for the error.



Component Name	Error Ontology	Probability Value	Probability Threshold	Hazard Level	Causes of UCA	Safety Constraints
Door_controller.impl	ServiceCommission	0.1	Probable	Critical	General: The door controller has ...	SC1: The door controller must not ope
Door_actuator.impl	ServiceCommission	0.002	Occasional	Critical	General: The door actuator has i...	SC2: The door actuator must not exec
Physical_Door.impl	ServiceCommission	0.00001	Remote	Critical	General: The door controller's wr...	SC3: The physical door should not allc

Figure 4.4: ASAM's report for Train Automated Door Control System Architecture

ASAM generates a report for the train automated door control system architecture based on component names, error ontology, probability values, probability thresholds, hazard levels, unsafe control actions, causes of unsafe control actions and safety constraints. This stream of information is attached to each implementation part of the component. Specifically, besides identifying error propagation information for each component, we have recorded additional information for each part of the error flow (flow source [*Door_controller.impl*], flow path[*Door_actuator.impl*], and flow sink[*Physical_door.impl*]. The report is shown in figure 4.4. That analysis report shows the source of the faults in the train door model, which is the *Door_controller.impl*. Each fault in the controller produces an error, which is of the type *service commission*. Each error in the door controller has a chance of harming people based on the probability of occurrence for that error, represented as (*p=0.1*). The report tells also tells us what the unsafe action of the door controller is for that type of error as well as the general and specific causes for that error. Finally, the report shows the safety constraint for the safety analysts or stake holders to mitigate the effects of the error.

4.1.4 Medical Embedded Device-Pacemaker

The third system we introduce is a safety-critical embedded system, an implantable medical device known as a pacemaker. The pacemaker is used to regulate abnormal heart rhythms. It has two main tasks, sensing and pacing. In pacing, it paces the heart in case the heart's own rhythm is irregular or too slow. In sensing, it monitors the heart's natural electrical activity. If the pacemaker senses a natural heart beat, it will not stimulate the heart. According to our method we need to specify major components of the pacemaker like the controller (DCM:Device-Controller Monitor), actuator (PG: Pulse Generator), controlled process (heart), and sensor (electrode/lead) [29, 31]. We can connect the components as shown in figure 3.1 . Our example starts from the actuator to the sensor's output as shown in figure 3.1, error flow 3 (scenario 3: PG $\rightarrow$ heart$\rightarrow$electrode/lead).

The initial results for this medical device has been published in [38] which are:

1. The error ontology identifies timing errors like late delivery. Now, the hazard is "the pacemaker is not working properly".

2. The feedback control loop has been built from the actuator to the sensor as shown in figure 3.1 which is error flow 3 (scenario 3).

3. The unsafe control action is that "the pacemaker does not provide an electrical pulse when its required". The safety constraints for the unsafe action is that "the pacemaker should provide an electrical pulse whenever its needed".

4. The general cause for the unsafe control action is that "the actuator has an internal failure" and the specific cause is that "the propagation of timing error from the electrode to pulse generator".

5. This is shown in the developed safety architecture subsection, as shown in figure 3.1

We will provide and discuss the feedback control loop architecture for the pacemaker.

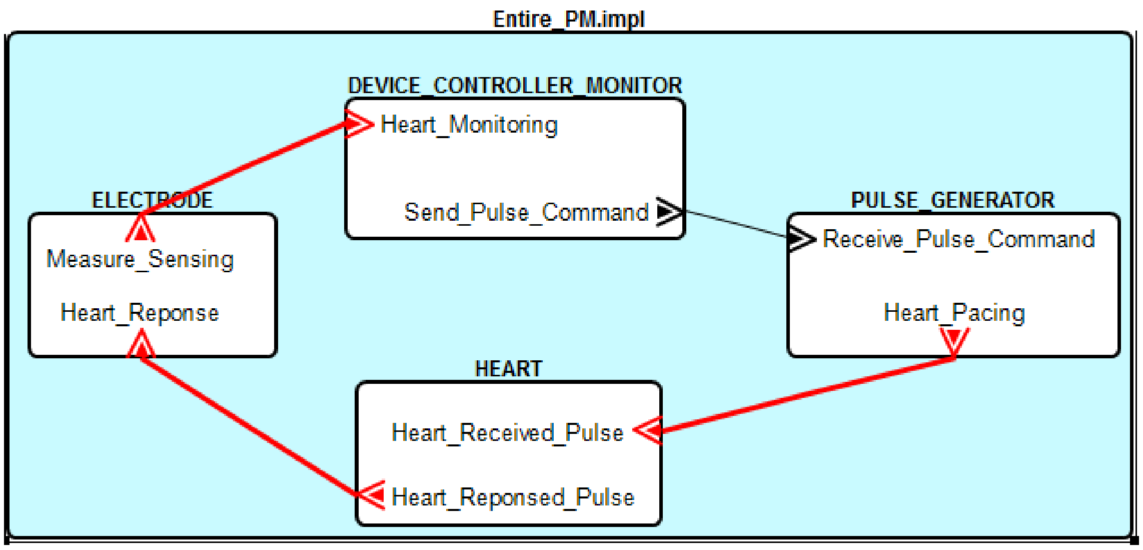
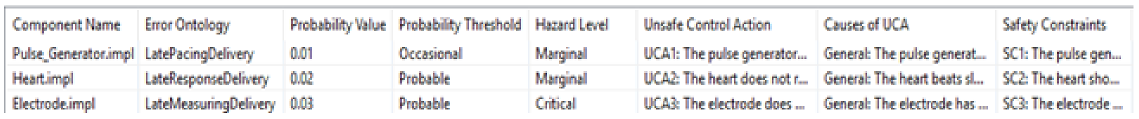


Figure 4.5: Three-Way Communication Format for Embedded Medical Device

Figure 4.5 shows the feedback control loop architecture that has been built for the pacemaker. Figure 4.5 shows the error flow based on the three-way interaction format for the pacemaker.*PULSE_GENERATOR* has an internal failure. We know the result of each internal failure is an error and the error propagates to the next component through nominal control flow as well as the error propagation path. When the *ELECTRODE* detects that the *HEART* needs pacing, the *DEVICE_CONTROLLER_MONITOR* decides to send the command to the *PULSE_GENERATOR* to send an electrical pulse to the *HEART*. The *PULSE_GENERATOR* receives the command, but it is not executed instantly because it has internal failure. In this situation, the produced error inside of the *PULSE_GENERATOR* constantly effects the operational state of the *PULSE_GENERATOR* and also directly impacts the device *CONTROLLER_MONITOR*'s command. If the *PULSE_GENERATOR* is able to resolve this problem that would not be hazardous situation. But, if it is not, the situation will be hazardous for the patient such as sending pulse to the heart late.



Component Name	Error Ontology	Probability Value	Probability Threshold	Hazard Level	Unsafe Control Action	Causes of UCA	Safety Constraints
Pulse_Generator.impl	LatePacingDelivery	0.01	Occasional	Marginal	UCA1: The pulse generator...	General: The pulse generat...	SC1: The pulse gen...
Heart.impl	LateResponseDelivery	0.02	Probable	Marginal	UCA2: The heart does not r...	General: The heart beats sl...	SC2: The heart sho...
Electrode.impl	LateMeasuringDelivery	0.03	Probable	Critical	UCA3: The electrode does ...	General: The electrode has ...	SC3: The electrode ...

Figure 4.6: ASAM's report for Pacemaker System Architecture

ASAM generates the report for the pacemaker system architecture based on the following parameters: component names, error ontology, probability values, probability thresholds, hazard levels, unsafe control actions, causes of unsafe control actions and safety constraints. These streams of information are attached to each implementation part of the component. Besides identifying error propagation information for each component, we have recorded additional information for each part of the error flow (flow source[*Pulse_Generator.impl*], flow path[*Heart.impl*] and flow sink [*Electrode.impl*]. The report is shown in figure 4.6. This analysis report shows the source of the error in the safety-critical embedded system model(i.e. pacemaker), which is *LatePacingDelivery*. Each error in the pacemaker has a chance of harming the patient's heart based on the probability of occurrence for that type of error, represented with (*p=0.01*). ASAM compares the *p* value with probability of occurrence thresholds to find the probability of the hazard occurring. In this example, it found (*occasional*) and the hazard level is (*Marginal*). The report tells us the unsafe action of the pulse generator for that type of error, which is "*The pulse generator does not provide an electrical pulse when its required*". Also, the report shows the general and specific causes for that error, which are "*general: the pulse generator has an internal failure, specific: the propagating of timing error from electrode to the pulse generator*". Finally, the report shows the safety constraint for the safety analysts to know how to mitigate the effects of the error, which is "*the pulse generator should provide an electrical pulse whenever its needed*".

4.1.5 Infant Incubator Temperature Control System- Isolette

The fourth system we introduce is another safety-critical system, known as an isolette. The isolette is an infant incubator temperature control system. The goal of this system is to monitor an infant at a safe or comfortable temperature and warn the nurse if the infant becomes too hot or too cold. In general, the system is used to control air temperatures inside of incubator so that they remain in a desired temperature range. We provide this example to support scenario 4 3.2.1. We evaluate what will happen if the operator interface device (OID) device in an infant incubator (isolette) gives an out of range value for the air temperature inside the isolette because of an internal failure? (e.g. the OID device displays correct values on the screen, but in reality does not produce an air temperature within the target range specified by the nurse). At a result, the infant is harmed by air temperature on the inside isolette being too hot / too cold. To solve this problem, the controller should send notification to the screen of the OID device about error values and should not send the command to the actuator.

The initial result for isolette example are:

1. The error ontology identifies out of range errors as an incorrect values. Now, the hazard is ”the infant is not safe with (out of range) temperature values such as below or above of the desired range”.
2. The feedback control loop has been built from the operator interface device to the actuator as shown in figure 3.2 which is error flow 4 (scenario 4).
3. The unsafe control action is that ”the operator interface device does not send the message to warn the nurse when its required”. The safety constraints for the unsafe action is that ”the device should dispatch the message to warn the nurse whenever its needed”.
4. The general cause for the unsafe control action is that ”the connected device has an internal failure” and the specific cause is that ”the propagating of error values from connected device to actuator”.
5. This is shown in the developed safety architecture subsection, as shown in figure 3.2.

We will provide and discuss the feedback control loop architecture for the infant incubator temperature control system.

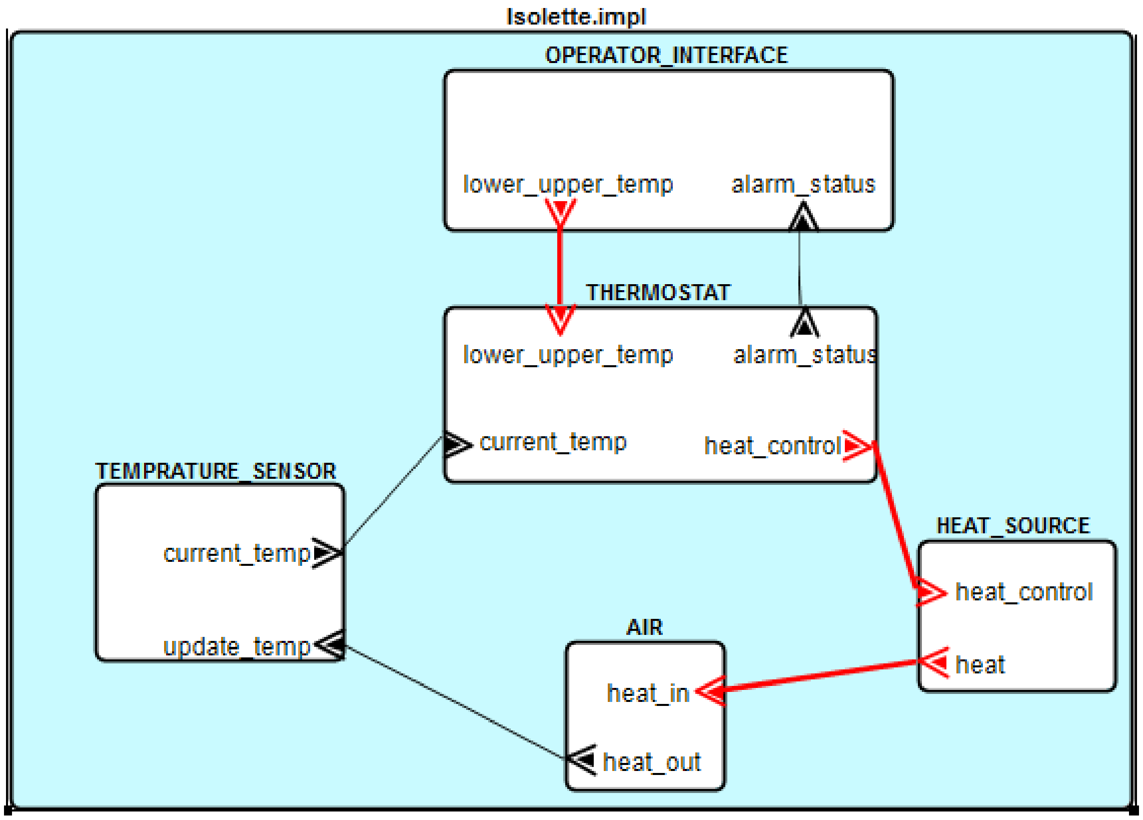
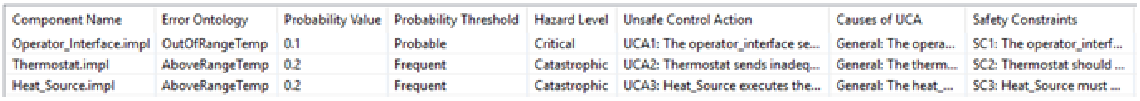


Figure 4.7: Three-Way Communication Format for Isolette System Architecture

Figure 4.7 shows the feedback control loop architecture which has been built for the isolette. Figure 4.7 shows the error flow based on three-way communication format for that isolette system. This example is different from the previous example because we show the effects of the internal failure of in *OPERATOR_INTERFACE_DEVICE* which is not a fundamental element in the feedback control loop architecture. The *OPERATOR_INTERFACE_DEVICE* is used to set *THERMOSTAT* and monitor the status of the system. When *OPERATOR_INTERFACE_DEVICE* has an internal failure, this leads to the sending an out of range temperature error values to *THERMOSTAT*. In this case, the *THERMOSTAT* will receive above range or below range temperature values. In this example, we have selected the above range temperature error value. This error effects the *THERMOSTAT*'s decision because it is not able to understand the temperature value that has been defined previously in the system controller. In this situation, if the *THERMOSTAT* is able to resolve this problem, it would not be a hazardous situation. But, if it is not, the situation will be hazardous for the infant such as sending inadequate commands to the *HEAT_SOURCE* actuator. The inadequate command affects the infant because the air temperature inside of the incubator is going to be higher than maximum temperature value. The infant will be harmed because incubator is going to be too hot. Also, that inadequate command does not send the message to the *OPERATOR_INTERFACE_DEVICE* to warn the nurse of the malfunction.



Component Name	Error Ontology	Probability Value	Probability Threshold	Hazard Level	Unsafe Control Action	Causes of UCA	Safety Constraints
Operator_Interface.impl	OutOfRangeTemp	0.1	Probable	Critical	UCA1: The operator_interface se...	General: The opera...	SC1: The operator_interf...
Thermostat.impl	AboveRangeTemp	0.2	Frequent	Catastrophic	UCA2: Thermostat sends inadeq...	General: The therm...	SC2: Thermostat should ...
Heat_Source.impl	AboveRangeTemp	0.2	Frequent	Catastrophic	UCA3: Heat_Source executes the...	General: The heat_...	SC3: Heat_Source must ...

Figure 4.8: ASAM's report for Isolette System Architecture

ASAM generates a report for the isolette system architecture based on the following parameters: component names, error ontology, probability values, probability thresholds, hazard levels, unsafe control actions, causes of unsafe control actions and safety constraints. This particular information is connected to each component's implementation portion. In spite of identifying error propagation information for each component's specification, we have recorded additional information for each portion of error: flow source (flow source[*Operator_Interface.impl*], flow path[*Thermostat.impl*], and flow sink[*Heat_Source.impl*]). The report is shown in figure 4.8. That report shows the component's fault source in the infant incubator temperature control system, which is *Operator_Interface.impl*. Each fault gives an error. The operator interface's fault gives an *OutOfRangeTemp* error. Each error in the isolette system has a chance of harming the infant based on the probability of occurrence for that type of error, represented as (*p=0.1*). ASAM compares the *p* value with the probability of occurrence thresholds to find the probability of hazard occurring; it found (*probable*) so the hazard level is (*critical*). The report tells us what unsafe action the operator interface performs for that type of specific error, which is "*The operator_interface sends an out of range temperature error values to thermostat*". Also, the report shows the general and specific causes for that type of error, which are "*general: the operator_interface has an internal failure, specific: The operator_interface accidentally receives incorrect setting values by the nurse*". Finally, the report shows the safety constraint for the stakeholders to know how to mitigate the effects of the error, which is "*The operator_interface device should not send incorrect values to the thermostat*".

4.1.6 Summary

In this section, we have introduced four examples of varying complexity and scenarios for each example. For each scenario, we have provided an instantiation of the ASAM annex representing the introduced three-way communication format scenarios as well as a report of the output produced by the ASAM evaluator.

The purpose of these examples is to demonstrate that ASAM finds more hazards using the three-way communication format as opposed to using the two-way format. ASAM identify hazards for the system scenarios based on the following parameters: component specification and implementation information, error ontology, probability values, probability thresholds, hazard levels, unsafe control actions, causes of unsafe control actions and safety constraints. For each scenario, we have also demonstrated not only the ability to represent the scenario in the architecture model but the ability to evaluate the representation to produce reports.

4.2 Demonstrating Error Mitigation by Safety Constraints

Claim C2: Each error flow in ASAM assists in identifying a safety constraint of appropriate breadth to eliminate the unsafe control actions caused by an identified error. In addition, each error flow can guide us to identify specific causes for the unsafe control action.

4.2.1 Evaluation Plan

To demonstrate claim C2, we take the previous complex examples that we analyzed with respect to the scenarios and we perform error mitigation analysis by safety constraints on each example. For that purpose, we will use ASAM to focus on error behavior of the components in the feedback control loop architecture and follow each error (port by port) and (component by component) until the destination. This following of the error allows us to identify the source of the hazard and provides safety constrains to mitigate it.

We will evaluate our selected examples that were described previously to support the scenarios. In these examples, we need to identify the major component types (i.e. sensor, controller, actuator, controlled process), the component's internal failure which causes error on the output ports, and the component's safety constraint need to handle errors through in ports and prevent errors through out ports. Then, we will produce an error mitigation analysis report for each example to ensure that error has been mitigated by the safety constraint in each specific component.

4.2.2 Error Mitigation Analysis for Adaptive Cruise Control System

We will now provide and discuss error mitigation by safety constraint for implementation part of each individual core components of the feedback control loop architecture for ACC system.

```
system ACC_sensor
    features
        ACC_State_s: in data port;
        ACC_Speed_s: out data port;
        ACC_Distance_s: out data port;
end ACC_sensor;

system implementation ACC_sensor.impl
    annex asam {**
        --Identify the component type
        type => sensor;
        --ACC_sensor's internal failure causes errors on out ports
        internal failure INTF1: "ACC sensor has internal failure" causes
            [SensorReadsErrorValues(Critical,p=0.02)] on [ACC_Speed_s,
            ACC_Distance_s];
    **};
end ACC_sensor.impl;
```

Listing 4.17: ASAM Error Mitigation Analysis for ACC System

In listing 4.17, we have represented the *ACC_sensor* interfaces: specification and implementation. The *ACC_sensor* specification interface is already described in section 4.1.2. The *ACC_sensor* implementation interface shows the component's sensor. Also, it shows the *ACC_sensor*'s internal failure which causes *SensorReadsErrorValues* on the output ports *ACC_Speed_s* and *ACC_Distance_s*. This port *ACC_Speed_s* represents an interface which sends the measured values of the speed of your own car as compared to the car in front. This port *ACC_Distance_s* represent an interface which sends the measured values of distance between your own car and the car in front.

```
system ACC_controller
    features
        ACC_Speed_c: in data port;
        ACC_Distance_c: in data port ;
        ACC_Speed_cmd: out data port;
        ACC_Ditance_cmd: out data port;
end ACC_controller;

system implementation ACC_controller.impl
    annex asam {**
        --identify the component type
        type => controller;
        --ACC controller have internal failure causes error on out ports.
        internal failure INTF2: "ACC controller has an internal failure" causes
            [ControllerComputeErrorValues(Catastrophic,p=0.03)] on
            [ACC_Speed_cmd, ACC_Ditance_cmd];
        --Error comes from the ACC sensor and causes error on out ports.
        [SensorReadsErrorValues] on [ACC_Speed_c, ACC_Distance_c] causes
            [SensorReadsErrorValues(Catastrophic,p=0.03)] on [ACC_Speed_cmd,
            ACC_Ditance_cmd];
        --ACC controller have a SC to handle error(s) on in ports.
        safety constraint SC1:"The ACC controller should handle errors that
            come from the ACC sensor" handles [SensorReadsErrorValues] on
            [ACC_Speed_c, ACC_Distance_c];
    **};
end ACC_controller.impl;
```

Listing 4.18: ASAM Error Mitigation Analysis for ACC System

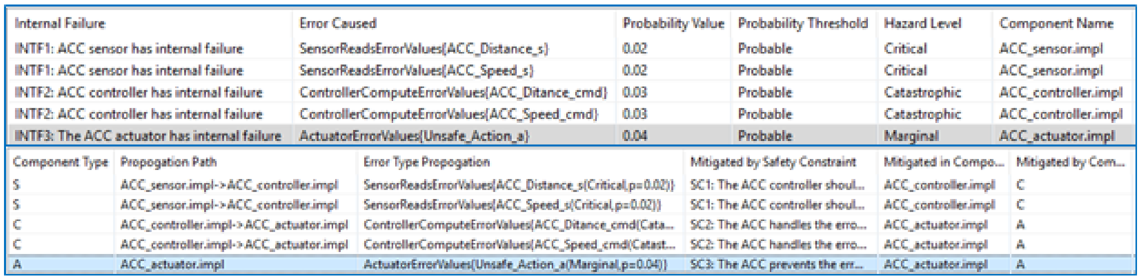
In listing 4.18, we show the *ACC_controller*'s specification and implementation. The *ACC_controller* specification interface is already described in 4.1.2. The *ACC_controller* implementation interface shows the component's controller. Also, it shows the *ACC_controller*'s internal failure causes *ControllerComputeErrorValues* on the output ports *ACC_Speed_cmd* and *ACC_Distance_cmd*. That error effects of the *ACC_controller*'s decision by giving incorrect commands to the actuator, which are catastrophic. At the same time, *ACC_sensor*'s error, *SensorReadsErrorValues*, is propagated to *ACC_controller*'s in ports *ACC_Speed_c* and *ACC_Distance_c* through the error propagation path. In this case, we provide a safety constraint to handle the *ACC_controller*'s incoming errors. Also, the *ACC_controller* does not let sensor's error propagate to the next component.

```
system ACC_actuator
    features
        ACC_EXE_Speed_a: in data port;
        ACC_EXE_Distance_a: in data port;
        Unsafe_Action_a: out data port;
end ACC_actuator;

system implementation ACC_actuator.impl
    annex asam {**
     --identify the component type
        type => actuator;
        --ACC actuator's internal failure causes errors on out ports
        internal failure INTF3: "The ACC actuator has an internal failure"
            causes [ActuatorErrorValues(Marginal,p=0.04)] on [Unsafe_Action_a];
        --Error comes from the ACC controller and causes error on out ports.
        [ControllerComputeErrorValues] on [ACC_EXE_Speed_a, ACC_EXE_Distance_a]
            causes [ControllerComputeErrorValues(Critical,p=0.1)] on
            [Unsafe_Action_a];
        --ACC actuator has a SC to handle incoming errors from controller
        safety constraint SC2:"The ACC handles the error that comes from the
            ACC controller" handles [ControllerComputeErrorValues] on
            [ACC_EXE_Speed_a,ACC_EXE_Distance_a]
        --ACC actuator has a SC to prevent its errors to out
        safety constraint SC3:"The ACC prevents the error of the actuator"
            prevents [ActuatorErrorValues]on [Unsafe_Action_a];
    **};
end ACC_actuator.impl;
```

Listing 4.19: ASAM Error Mitigation Analysis for ACC System

In listing 4.18, we show the *ACC_actuator*'s specification and implementation. The *ACC_actuator* specification interface is already described in 4.1.2. The *ACC_actuator* implementation interface shows the component's actuator. Also, it shows the *ACC_actuator*'s internal failure which causes *ActuatorErrorValues* on an output port *Unsafe_Action_a*. At the same time, the *ACC_controller*'s error, *ControllerComputeErrorValues*, is propagated to *ACC_actuator* in ports *ACC_EXE_Speed_a* and *ACC_EXE_Distance_a* through the error propagation path. In this case, we need to provide two safety constraints, one to handle the *ACC_actuator*'s incoming errors, the other to prevent the errors of *ACC_actuator* itself.



Internal Failure	Error Caused	Probability Value	Probability Threshold	Hazard Level	Component Name
INTF1: ACC sensor has internal failure	SensorReadsErrorValues{ACC_Distance_s}	0.02	Probable	Critical	ACC_sensor.impl
INTF1: ACC sensor has internal failure	SensorReadsErrorValues{ACC_Speed_s}	0.02	Probable	Critical	ACC_sensor.impl
INTF2: ACC controller has internal failure	ControllerComputeErrorValues{ACC_Ditance_cmd}	0.03	Probable	Catastrophic	ACC_controller.impl
INTF2: ACC controller has internal failure	ControllerComputeErrorValues{ACC_Speed_cmd}	0.03	Probable	Catastrophic	ACC_controller.impl
INTF3: The ACC actuator has internal failure	ActuatorErrorValues{Unsafe_Action_a}	0.04	Probable	Marginal	ACC_actuator.impl

Component Type	Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Compo...	Mitigated by Com...
S	ACC_sensor.impl->ACC_controller.impl	SensorReadsErrorValues{ACC_Distance_s(Critical,p=0.02)}	SC1: The ACC controller shoul...	ACC_controller.impl	C
S	ACC_sensor.impl->ACC_controller.impl	SensorReadsErrorValues{ACC_Speed_s(Critical,p=0.02)}	SC1: The ACC controller shoul...	ACC_controller.impl	C
C	ACC_controller.impl->ACC_actuator.impl	ControllerComputeErrorValues{ACC_Ditance_cmd(Cata...	SC2: The ACC handles the erro...	ACC_actuator.impl	A
C	ACC_controller.impl->ACC_actuator.impl	ControllerComputeErrorValues{ACC_Speed_cmd(Catast...	SC2: The ACC handles the erro...	ACC_actuator.impl	A
A	ACC_actuator.impl	ActuatorErrorValues{Unsafe_Action_a(Marginal,p=0.04)}	SC3: The ACC prevents the err...	ACC_actuator.impl	A

Figure 4.9: ASAM's Error Mitigation report for ACC System Architecture

Figure 4.9 shows ASAM's generated report about error mitigation analysis by safety constraints for the adaptive cruise control system architecture. ASAM generates the report based on the following information: internal failure for each major component, each internal failure causes error, probability value of error occurrence, probability thresholds to identify probability of which hazard could occur, major component implementation, major component specification, propagation path to identify error mitigation process step by step, error specification propagation gathers information about each error, mitigated error by safety constraint, mitigated error in component's implementation and mitigated error in component's specification. We iteratively explain figure 4.9 below:

1. Internal failure (INTF1) recorded in *ACC_sensor* causes *SensorReadsErrorValues* in *ACC_Distance_s* and *ACC_Speed_s*. The probability of occurrence for that error in measuring of distance and speed is (0.02). That error gives a *critical* level of hazard for the ACC system because *ACC_sensor* reads incorrect values of distance and speed for the car in front. In the error propagation path, we show that these two error values are propagated from sensor to controller [*ACC_sensor.impl* $\rightarrow$ *ACC_controller.impl*]. Also, in the error propagation type, we gather information about each error such as error name, port to propagate, hazard's severity level and p value. In this case, the controller has enough information about each propagated error. To mitigate each error, *ACC_controller* has a safety constraint (SC). For example, "*SC1: The ACC controller should handle errors that come from the sensor*" is used to mitigate the effects of the error measuring in distance and speed. This mitigation process has been

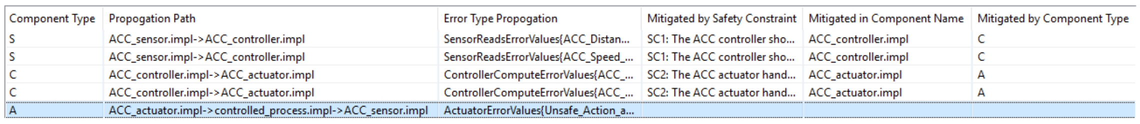
executed in *ACC_controller.impl*. Finally, we show that the errors are propagated from sensor *(S)* or *ACC_sensor.impl* as a source and have been mitigated in controller *(C)* or *ACC_controller.impl* as a destination.

2. Previously, we show that how the *ACC_controller* mitigates the incoming errors. But, what will happen if the *ACC_controller* itself has an internal failure? How does the *ACC_controller* solve this problem? For that purpose, we have recorded an internal failure (INTF2) in the *ACC_controller* which causes error, *ControllerComputeErrorValues*, in commands *ACC_Distance_cmd* and *ACC_Speed_cmd*. The probability of occurrence for that error in the commands is (0.03). That's (*catastrophic*) because the *ACC_controller*'s command is considered as an inadequate command. In the error propagation path, we show that these two error values have been propagated from controller to actuator [*ACC_controller.impl* $\rightarrow$ *ACC_actuator.impl*]. In addition, in the error propagation information, we collect information about each error such as error name, port to propagate, hazard's severity level and p value. In this case, the actuator has enough information about each propagated error from controller. To mitigate each error on an in ports, *ACC_actuator* has a safety constraint (SC) for it. For example,"*SC2: The ACC actuator handles the errors that come from the ACC controller*" is used to mitigate the effects of the error in controller's commands. This mitigation process has been executed in *ACC_actuator.impl* as a destination point. Finally, we show that the errors in the commands are propagated from controller *(C)* or *ACC_controller.impl* and have been mitigated in actuator *(A)* or *ACC_actuator.impl*.

3. Now, we know that how the *ACC_controller* handles incoming errors from the *ACC_sensor* and also we show that how *ACC_actuator* handles the incoming errors from the *ACC_controller*. But, what will happen if the *ACC_actuator* itself has an internal failure? How does this impact the controller's commands? For that purpose, we have recorded an internal failure (INTF3) in the *ACC_actuator* which causes *ActuatorErrorValues*. The probability of occurrence for that error during execution is (0.04). We consider the severity level for that error is (*marginal*) because the error effects the controller's commands during execution and leads to produce an unsafe action in *Unsafe_Action_a*. But, previously we show that how *ACC_actuator* mitigates the incoming propagated errors from *ACC_controller*. Now, we need to show that the *ACC_actuator* prevents errors that have been created by itself. In the error propagation path, we show that the source of the error is *ACC_actuator.impl* and we have enough information about it. To prevent that error to out, the *ACC_actuator* has a safety constraint (SC) for it. For example, "*SC3: The ACC actuator prevents the error of actuator itself*" is used to mitigate

the effects of the actuator itself. This mitigation process has been executed in *ACC_actuator.impl* as a destination point. Finally, we show that the errors produced in the actuator *(A)* or *ACC_actuator.impl* have been mitigated in the actuator *(A)* or *ACC_actuator.impl* itself.

4. What will happen if the *ACC_actuator.impl* is not able to prevent *ActuatorErrorValues* error out? Figure 4.10, the error propagation path shows that the *ActuatorErrorValues* error is propagated from the actuator to the sensor through controlled process, [*ACC_actuator.impl* $\rightarrow$ *controlled_process.impl* $\rightarrow$ *ACC_sensor.impl*]. This means *Unsafe_Action_a* of the *ACC_actuator.impl* updates the *ACC_sensor.impl*'s information by incorrect values of speed and distance. This leads to the *ACC_controller.impl* not being able to control speed and distance of the car in front because of the *ACC_actuator.impl*'s internal failure. To ensure that the error is propagated and has not been mitigated, the figure 4.10 shows empty cells in some specific parameters such as mitigated error by safety constraints, mitigated in component name and mitigated by component type.



Component Type	Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component Type
S	ACC_sensor.impl->ACC_controller.impl	SensorReadsErrorValues{ACC_Distan...	SC1: The ACC controller sho...	ACC_controller.impl	C
S	ACC_sensor.impl->ACC_controller.impl	SensorReadsErrorValues{ACC_Speed_...	SC1: The ACC controller sho...	ACC_controller.impl	C
C	ACC_controller.impl->ACC_actuator.impl	ControllerComputeErrorValues{ACC_...	SC2: The ACC actuator hand...	ACC_actuator.impl	A
C	ACC_controller.impl->ACC_actuator.impl	ControllerComputeErrorValues{ACC_...	SC2: The ACC actuator hand...	ACC_actuator.impl	A
A	ACC_actuator.impl->controlled_process.impl->ACC_sensor.impl	ActuatorErrorValues{Unsafe_Action_a...			

Figure 4.10: ASAM's report for Non-Mitigated Error in ACC system

4.2.3 Error Mitigation Analysis for Train Automated Door Control System

We will now provide and discuss error mitigation by safety constraint for the implementation of each individual core component of the feedback control loop architecture for the train automated door control system.

```
system Door_sensor
    features
        door_open_s: out data port;
        door_close_s: out data port;
        door_state_s: in data port;
        door_close_on_person: out data port;
end Door_sensor;

```

```
system implementation Door_sensor.impl
    annex asam {**
        --Identify type of major component
        type => sensor;
        --Door sensor has an internal failure causes error(s)on out ports.
        internal failure INTF1: "door reports it is closed when a person is in
            the doorway" causes [DoorSensorCommissionError(Critical,p=0.02)] on
            [door_close_on_person, door_open_s, door_close_s];
        --Door sensor has a safety constraint to prevent error out.
        safety constraint SC1: "door should be opened when the person is in the
            doorway" prevents [DoorSensorCommissionError] on [door_open_s,
            door_close_s];
    **};
end Door_sensor.impl;
```

Listing 4.20: ASAM Error Mitigation Analysis for Train Automated Door Control System

In listing 4.20, we have represented the *Door_sensor* specification and implementation. In the specification part, the interface has three out data ports, *door_open_s* and *door_close_s* are used to send the open and close status of the door to the controller, *door_close_on_person* is used to send status of the the door to the controller when an object is in the doorway. The interface also has one in data port, *door_state_s*, that allows the sensor to receive the last update values from physical door feedback about door status. In the implementation part, the interface shows the component's major type is sensor. Also, it shows the *Door_sensor*'s internal failure which causes *DoorSensorCommisionError* on the output ports,*door_close_on_person*, *door_open_s* and *door_close_s*. We have also represented the probability of occurrence of the error and hazard's severity level. Also, we need to provide a safety constraint, SC1, for the *Door_sensor* interface to prevent *DoorSensorCommisionError* through output ports.

```
system Door_controller
    features
        door_open_c: in data port;
        door_close_c: in data port;
        open_door_cmd: out data port;
        close_door_cmd: out data port;
        door_state_c: in data port;
        door_open_train_not_stopped: out data port;
end Door_controller;

system implementation Door_controller.impl
    annex asam {**
        --Identify type of major component
        type => controller;
        --Door controller has internal failure causes error(s) on out ports.
        internal failure INTF2: "door reports it is opened when the train is
            not stopped" causes
            [DoorControllerCommissionError(Marginal,p=0.04)] on
            [door_open_train_not_stopped, close_door_cmd, open_door_cmd];
        --Error comes from door sensor and causes error on controller's output
            ports.
        [DoorSensorCommissionError] on [door_open_c, door_close_c,
            door_state_c] causes
            [DoorSensorCommissionError(Catastrophic,p=0.002)] on
            [door_open_train_not_stopped, close_door_cmd, open_door_cmd];
        --Door controller have safety constraints to handle incoming error and
            prevent outgoing error.
        safety constraint SC2:"doors should be opened when the person is in the
            doorway" handles [DoorSensorCommissionError] on [door_open_c,
            door_close_c, door_state_c];
        safety constraint SC3:"doors should be closed when the train is not
            stopped at the station platform" prevents
            [DoorControllerCommissionError]on [close_door_cmd, open_door_cmd];
    **};
end Door_controller.impl;
```

Listing 4.21: ASAM Error Mitigation Analysis for Train Automated Door Control System

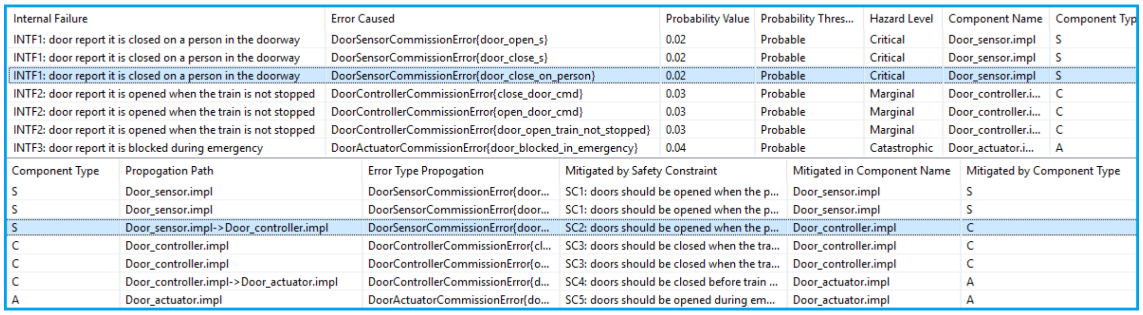
In listing 4.21, we have represented the *Door_controller* specification and implementation. In the specification part, *Door_controller* has three in data ports, *door_open_c* and *door_close_c* are used to receive open and close door status from the sensor respectively, *door_state_c* is used to receive any abnormal status of the door such as an object in the doorway but the door is closed. Also, *Door_controller* has three out data ports, *open_door_cmd* and *close_door_cmd* are used to send open and close door commands to actuator, *door_open_train_not_stopped* is used to send abnormal status of the door to actuator such as door train is opened before train is stopped at the station platform. In the implementation part, *Door_controller* shows the type of the component is the controller. Also, it shows the *Door_controller*'s internal failure which causes *DoorControllerCommissionError* on output ports *door_open_train_not_stopped*, *open_door_cmd* and *close_door_cmd*. This error effects the *Door_controller*'s decision by giving inadequate commands to door actuator. At the same time, *Door_sensor* error, *DoorSensorCommisionError*, is propagated from *Door_controller*'s in ports *door_open_c*, *door_close_c* and *door_state_c* through the error propagation path. This effects the *Door_controller*'s decision as well. In this case, we need to provide two safety constraints, SC2 and SC3, to mitigate the effects of the incoming and outgoing errors respectively.

```
system Door_actuator
    features
        open_door_a: in data port;
        close_door_a: in data port;
        door_state_a: in data port;
        door_blocked_in_emergency: out data port;
end Door_actuator;

system implementation Door_actuator.impl
    annex asam {**
        --Identify type of major component
        type => actuator;
        --Door actuator has internal failure causes error on out ports
        internal failure INTF3: "door reports it is blocked during emergency"
            causes [DoorActuatorComissionError(Catastrophic,p=0.2)] on
            [door_blocked_in_emergency];
        --Error comes from door controller and causes error on out port.
        [DoorControllerCommissionError] on [open_door_a, close_door_a,
            door_state_a] causes
            [DoorControllerCommissionError(Critical,p=0.1)] on
            [door_blocked_in_emergency];
        -- Door actuator have safety constraints to handle incoming errors and
           prevent outgoing errors.
        safety constraint SC4:"doors should be closed before train is moving
           and doors should be opened after train is stopped at the station
           platform" handles [DoorControllerCommissionError] on
           [open_door_a,close_door_a, door_state_a];
        safety constraint SC5:"doors should be opened during emergency"
           prevents [DoorActuatorComissionError]on [door_blocked_in_emergency];
    **};
end Door_actuator.impl;
```

Listing 4.22: ASAM Error Mitigation Analysis for Train Automated Door Control System

In listing 4.22, we have represented the *Door_actuator* specification and implementation. From the specification perspective, *Door_actuator* has three in data ports, *open_door_a* and *close_door_a* are used to receive open and close door commands from the controller, *door_state_a* is used to receive any abnormal status of the door such as doors are not closed before moving. Also, *Door_actuator* has an out data port, *door_blocked_in_emergency* is used to send abnormal status of the door to controlled process such as doors are blocked during an emergency which leads to people stuck inside the train. From the implementation information perspective, the *Door_actuator* shows the component's major type, which is the actuator. Also, it shows the *Door_actuator*'s internal failure which causes *DoorActuatorComissionError* on an output port *door_blocked_in_emergency*. That error effects the *Door_actuator*'s execution commands. At the same time, the *Door_controller*'s error, *DoorControllerCommissionError*, is propagated to *Door_actuator* in ports *open_door_a*, *close_door_a* and *door_state_a* through the error propagation path. Now, the *Door_actuator* gets two types of errors: one from the controller of the door and the other generated by the actuator itself. In this situation, we need to provide two safety constraints, SC4 and SC5, to mitigate the effects of the incoming and outgoing errors respectively.



Internal Failure	Error Caused	Probability Value	Probability Thres...	Hazard Level	Component Name	Component Typ
INTF1: door report it is closed on a person in the doorway	DoorSensorCommissionError{door_open_s}	0.02	Probable	Critical	Door_sensor.impl	S
INTF1: door report it is closed on a person in the doorway	DoorSensorCommissionError{door_close_s}	0.02	Probable	Critical	Door_sensor.impl	S
INTF1: door report it is closed on a person in the doorway	DoorSensorCommissionError{door_close_on_person}	0.02	Probable	Critical	Door_sensor.impl	S
INTF2: door report it is opened when the train is not stopped	DoorControllerCommissionError{close_door_cmd}	0.03	Probable	Marginal	Door_controller.i...	C
INTF2: door report it is opened when the train is not stopped	DoorControllerCommissionError{open_door_cmd}	0.03	Probable	Marginal	Door_controller.i...	C
INTF2: door report it is opened when the train is not stopped	DoorControllerCommissionError{door_open_train_not_stopped}	0.03	Probable	Marginal	Door_controller.i...	C
INTF3: door report it is blocked during emergency	DoorActuatorCommissionError{door_blocked_in_emergency}	0.04	Probable	Catastrophic	Door_actuator.i...	A

Component Type	Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component Type
S	Door_sensor.impl	DoorSensorCommissionError{door...	SC1: doors should be opened when the p...	Door_sensor.impl	S
S	Door_sensor.impl	DoorSensorCommissionError{door...	SC1: doors should be opened when the p...	Door_sensor.impl	S
S	Door_sensor.impl->Door_controller.impl	DoorSensorCommissionError{door...	SC2: doors should be opened when the p...	Door_controller.impl	C
C	Door_controller.impl	DoorControllerCommissionError{cl...	SC3: doors should be closed when the tra...	Door_controller.impl	C
C	Door_controller.impl	DoorControllerCommissionError{o...	SC3: doors should be closed when the tra...	Door_controller.impl	C
C	Door_controller.impl->Door_actuator.impl	DoorControllerCommissionError{d...	SC4: doors should be closed before train ...	Door_actuator.impl	A
A	Door_actuator.impl	DoorActuatorCommissionError{do...	SC5: doors should be opened during em...	Door_actuator.impl	A

Figure 4.11: ASAM's Error Mitigation report for Train Automated Door Control System Architecture

Figure 4.11 shows ASAM's generated report about error mitigation analysis by safety constraints for the train automated door control system architecture. ASAM generates the report based on the following information: internal failure for each major component, each internal failure causes error, probability value of error occurrence, probability thresholds to identify probability of which hazard could occur, major component implementation, major component specification, propagation path to identify error mitigation process step by step, error specification propagation collects information about each error, mitigated error by safety constraint, mitigated error in components implementation and mitigated error in components specification. We iteratively explain figure 4.11 below:

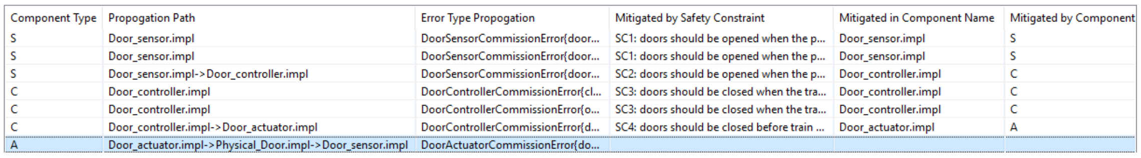
1. The internal failure (INTF1) recorded in the *Door_sensor* which causes *DoorSensorCommisionError* on output port. This error leads to closing the door on a person in the doorway. The probability of occurrence for that error in the sensor mis-reading an object is (0.02). This error gives a *critical* level of hazard for people before train door system is developed because *Door_sensor* does not read objects in the doorway. In the error propagation path, we show that the error is propagated from the sensor to the controller[*Door_sensor.impl* $\rightarrow$ *Door_controller.impl*] through the error propagation path. Also, in the error propagation specification, the controller gathers information about the error such as name of the error, port to propagate, hazard's severity level of the error and p value. To mitigate the propagated error, the *Door_controller* has safety constraints (SC). For example, "*SC2: doors should be opened when the person is in the doorway*" is used to mitigate the effects of misreading objects. This mitigation process has been executed in the *Door_controller.impl*. Finally, we show that the error is propagated from sensor(*S*) or *Door_sensor.impl* as a source of the error and has been mitigated in the controller(*C*) or *Door_controller.impl* as a destination of the error.

2. Previously, we show that how the *Door_controller* mitigates the incoming errors from the *Door_sensor*. But, what will happen if the *Door_controller* itself has an internal failure? How can the *Door_controller* solve this problem? For that purpose, we have recorded an internal failure (INTF2) in the *Door_controller* which causes error, *DoorControllerCommissionError*, in commands. This error leads to letting the controller open the doors before train is stopped or after train is started. The probability of occurrence for that error is (0.03). That p value gives *marginal* severity level to people. In the error propagation path, we show that the error is propagated from the controller to the actuator [*Door_controller.impl* $\rightarrow$ *Door_actuator.impl*] through the error propagation path. Also, in the error propagation information, the controller gathers information about the error and sends it to the actuator such as name of the error, port to propagate, hazard's severity level of the error and p value. Now, to mitigate the propagated error in the actuator, *Door_actuator* provides a safety constraint for it. For example, "*SC4: doors should be closed before train is moving and doors should be opened after train is stopped at the station platform*" is used to mitigate the effects of the error in the controller's command. This mitigation process has been executed in the *Door_actuator.impl* as a destination point. Finally, we show that the source of the error is the door controller(*C*) or *Door_controller.impl* and mitigated in the door actuator (*A*) or *Door_actuator.impl*.

3. Now, we know that how the *Door_controller* handles incoming errors from the *Door_sensor* and we also show that the *Door_actuator* handles the incoming errors from the *Door_controller*. But, what will happen if the *Door_actuator* itself has an internal failure? For that purpose, we have recorded an internal failure (INTF3) in the *Door_actuator* which causes *DoorActuatorCommissionError*. This error leads to blocking the doors during an emergency. The probability of occurrence for that error during execution is (0.04). That p value gives a *catastrophic* severity level. Previously, we show how *Door_actuator* mitigates the incoming propagated errors from *Door_controller*. Now, we need to show how the *Door_actuator* prevents errors that have been produced by itself. In the error propagation path, we show that the source of the error is *Door_actuator.impl* and the actuator has sufficient information about the error such as name of produced error, port to propagate, p value and hazard's severity level. To prevent *DoorActuatorCommissionError*, the *Door_actuator* has a safety constraint for it. For example, "*SC5: doors should be opened during emergency*" is used to mitigate the effects of the error. This mitigation process has been executed in *Door_actuator.impl* as a destination point. Finally, we show that the source of produced error in the actuator (*A*) or *Door_actuator.impl* and has been mitigated in

the (A) or *Door_actuator.impl*.

4. What will happen if the *Door_actuator* is not able to prevent *DoorActuatorCommissionError*? As shown in figure 4.12, the error propagation analysis shows that the error has been propagated to *Door_sensor.impl* through *Physical_Door.impl*. This means *Door_actuator* updates the *Door_sensor*'s information incorrectly. This leads to the *Door_controller* not being able to open the door during an emergency because the sensor continuously provides incorrect values of data to controller and the controller will do incorrect computation for that data which causing it to send inadequate commands that would not lead to opening the physical door. To ensure that the error is propagated and has not been mitigated, the figure 4.12 shows empty cells in some specific parameters such as mitigated error by safety constraints, mitigated in component name, and mitigated by component type.



Component Type	Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component
S	Door_sensor.impl	DoorSensorCommissionError{door...	SC1: doors should be opened when the p...	Door_sensor.impl	S
S	Door_sensor.impl	DoorSensorCommissionError{door...	SC1: doors should be opened when the p...	Door_sensor.impl	S
S	Door_sensor.impl->Door_controller.impl	DoorSensorCommissionError{door...	SC2: doors should be opened when the p...	Door_controller.impl	C
C	Door_controller.impl	DoorControllerCommissionError{cl...	SC3: doors should be closed when the tra...	Door_controller.impl	C
C	Door_controller.impl	DoorControllerCommissionError{o...	SC3: doors should be closed when the tra...	Door_controller.impl	C
C	Door_controller.impl->Door_actuator.impl	DoorControllerCommissionError{d...	SC4: doors should be closed before train ...	Door_actuator.impl	A
A	Door_actuator.impl->Physical_Door.impl->Door_sensor.impl	DoorActuatorCommissionError{do...			

Figure 4.12: ASAM's report for Non-Mitigated Error in Train Door

4.2.4 Error Mitigation Analysis for Medical Embedded Device

We will now provide and discuss error mitigation by safety constraint for the implementation of each individual core component of the feedback control loop architecture for the medical embedded device - pacemaker.

```
system Electrode
    features
        Measure_Sensing: out event data port;
        Heart_Response: in event data port;
end Electrode;

system implementation Electrode.impl
    annex asam{**
        --Identify type of major component
        type => sensor;
        --Electrode is the sensor of the pacemaker. The internal failure in it
            causes electrode error.
        internal failure INTF1:"The electrode does not measure sensing values
            on time" causes [LateMeasureSensing(Critical,p=0.02)] on
            [Measure_Sensing];
    **};
end Electrode.impl;
```

Listing 4.23: ASAM Error Mitigation Analysis for pacemaker

In listing 4.23, we have represented the *Electrode* sensor specification and implementation. From the specification perspective, the interface has two ports *Measure_Sensing* and *Heart_Response*. The *Measure_Sensing* is an out event data port and used to send sensing measured values to controller.*Heart_Response* is an in event data port and used to receive the heart response values whenever the heart is paced. From the implementation perspective, we have identified the *Electrode* major type which is the sensor. Also, we show that the *Electrode* has an internal failure which causes *LateMeasureSensing* error on the out event data port *Measure_Sensing*. Additionally, we have specified the probability of occurrence for that error and hazard's severity level.

```
system Device_Controller_Monitor
    features
        Heart_Monitoring: in event data port;
        Send_Pulse_Command: out event data port;
end Device_Controller_Monitor;

system implementation Device_Controller_Monitor.impl
    annex asam{**
        --Identify type of major component
        type => controller;
        --DCM is the controller of the pacemaker. The internal failure in it
            causes DCM error.
        internal failure INTF2:"The DCM does not send pacing command on time to
            the pulse generator" causes [LatePacingCommand(Critical,p=0.03)] on
            [Send_Pulse_Command];
        --The Electrode error propagates to DCM.
```

```
            [LateMeasureSensing] on [Heart_Monitoring] causes
                [LateMeasureSensing(Critical,p=0.002)] on [Send_Pulse_Command];
            --DCM have a SC to mitigate errors coming from the electrode
            safety constraint SC1:"The DCM should receive heart sensing values on
                time to send the pace command on time as well" handles
                [LateMeasureSensing] on [Heart_Monitoring];
        **};
end Device_Controller_Monitor.impl;
```

Listing 4.24: ASAM Error Mitigation Analysis for pacemaker

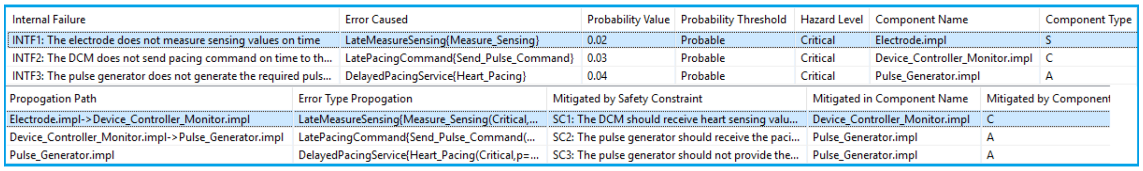
In listing 4.24, we have represented the *Device_Controller_Monitor* specification and implementation. From specification perspective, the interface has two ports *Heart_Monitoring* and *Send_Pulse_Command*. The *Heart_Monitoring* is an in event data port and is used to monitor measured sensing values of the heart. The *Send_Pulse_Command* is an out event data port and is used to send pulse command to pulse generator. From the implementation perspective, we have identified the component major type which is controller. We show the *Device_Controller_Monitor* has an internal failure which causes *LatePacingCommand* on the out event data port *Send_Pulse_Command*. This error effects of the controller's decision by giving pulse command in late. At the same time, *Electrode* sensor error, *LateMeasureSensing* is propagated to *Device_Controller_Monitor* through error propagation path. The sensor's propagated error also effects of the controller's decision as well. In the controller, we need to provide a safety constraint, SC1, to handle the incoming error first.

```
system Pulse_Generator
    features
        Receive_Pulse_Command: in event data port;
        Heart_Pacing: out event data port;
end Pulse_Generator;

system implementation Pulse_Generator.impl
    annex asam{**
        --Identify type of major component
        type => actuator;
        --The PG is the pulse generator. The internal failure in it causes PG
            errors.
        internal failure INTF3:"The pulse generator does not generate the
            required pulse whenever is needed" causes
            [DelayedPacingService(Critical,p=0.04)] on [Heart_Pacing];
        --DCM error from controller causes Delayed Service in PG.
        [LatePacingCommand] on [Receive_Pulse_Command] causes
            [LatePacingCommand(Critical,p=0.1)] on [Heart_Pacing];
        safety constraint SC2:"The pulse generator should receive the pacing
            command on time from DCM" handles [LatePacingCommand] on
            [Receive_Pulse_Command];
        safety constraint SC3:"The pulse generator should not provide the late
            pacing service to the heart" prevents [DelayedPacingService]on
            [Heart_Pacing];
    **};
end Pulse_Generator.impl;
```

Listing 4.25: ASAM Error Mitigation Analysis for pacemaker

In listing 4.25, we have represented the *Pulse_Generator* specification and implementation. From the specification perspective, the interface has two ports *Receive_Pulse_Command* and *Heart_Pacing*. The *Receive_Pulse_Command* is an in event data port and is used to receive pulse command from the controller. The *Heart_Pacing* is an out event data port and is used to execute the received command such as pacing heart according to the controller's decision. From the implementation perspective, we have identified the component's major type which is the actuator. Also, we show that the *Pulse_Generator* has an internal failure which causes *DelayedPacingService* error on the output port *Heart_Pacing*. That error effects the *Pulse_Generator*'s execution commands. At the same time, the *Device_Controller_Monitor* error, *LatePacingCommand*, is propagated to *Pulse_Generator* in port *Receive_Pulse_Command* through error propagation path. That error also effects of the execution commands in the *Pulse_Generator* as well. Now, the *Pulse_Generator* gets two types of errors: one from the *Device_Controller_Monitor* and the other by *Pulse_Generator* itself. In this case, we need to provide two safety constraints, SC2 and SC3, to mitigate the effects of the incoming and outgoing errors respectively.



Internal Failure	Error Caused	Probability Value	Probability Threshold	Hazard Level	Component Name	Component Type
INTF1: The electrode does not measure sensing values on time	LateMeasureSensing{Measure_Sensing}	0.02	Probable	Critical	Electrode.impl	S
INTF2: The DCM does not send pacing command on time to th...	LatePacingCommand{Send_Pulse_Command}	0.03	Probable	Critical	Device_Controller_Monitor.impl	C
INTF3: The pulse generator does not generate the required puls...	DelayedPacingService{Heart_Pacing}	0.04	Probable	Critical	Pulse_Generator.impl	A

Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component
Electrode.impl->Device_Controller_Monitor.impl	LateMeasureSensing{Measure_Sensing(Critical,...	SC1: The DCM should receive heart sensing valu...	Device_Controller_Monitor.impl	C
Device_Controller_Monitor.impl->Pulse_Generator.impl	LatePacingCommand{Send_Pulse_Command(...	SC2: The pulse generator should receive the paci...	Pulse_Generator.impl	A
Pulse_Generator.impl	DelayedPacingService{Heart_Pacing(Critical,p=...	SC3: The pulse generator should not provide the...	Pulse_Generator.impl	A

Figure 4.13: ASAM's Error Mitigation report for pacemaker

Figure 4.13 shows ASAM's generated report about error mitigation analysis by safety constraints for the an embedded medical device is called pacemaker. ASAM generates the report based on the following information: internal failure for each major component, each internal failure causes error, probability value of error occurrence, probability thresholds to identify probability of which hazard could occur, major component implementation, major component specification, propagation path to identify error mitigation process step by step, error specification propagation collects information about each error, mitigated error by safety constraint, mitigated error in components implementation and mitigated error in components specification. We iteratively explain figure 4.13 below:

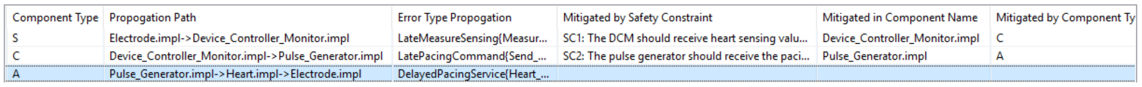
1. The internal failure (INTF1) recorded in the *Electrode* which causes *LateMeasureSensing* error on output port *Measure_Sensing*. This error leads to reading the heart sensing values in late. The probability of occurrence for that error in the sensor late-reading sense value is (0.02). This error gives a *critical* level of hazard for patient before pacemaker is developed because *Electrode* does not read sensing values on time during operational system context. In the error propagation path, we show that the error is propagated from the sensor to controller[*Electrode.impl* $\rightarrow$ *Device_Controller_Monitor.impl*] through the error propagation path. Also, in the error propagation specification, the controller gathers information about the propagated error such as name of the error, port to propagate, hazard's severity level of the error and p value. To mitigate the propagated error, the *Device_Controller_Monitor.impl* has safety constraint (SC). For example, *"SC1:The DCM should receive heart sensing values on time to send the pace command on time as well"* is used to mitigate the effects of late-reading values. This mitigation process has been executed in the *Device_Controller_Monitor.impl*. Finally, we show that the error is propagated from sensor (*S*) or *Electrode.impl* as a source of the error and has been mitigated in the controller *(C)* or *Device_Controller_Monitor.impl* as a destination of the error.

2. Previously, we show that the *Device_Controller_Monitor.impl* mitigates the incoming errors from the *Electrode.impl*. But, what will happen if the *Device_Controller_Monitor.impl* itself has an internal

failure?How can the *Device_Controller_Monitor.impl* solve this problem? For that purpose, we have recorded an internal failure (INTF2) in the *Device_Controller_Monitor.impl* which causes error, *LatePacingCommand*. This error leads to letting the controller send the pace command in a late time because it receives sensing values in a late time as well. The probability of occurrence for that error is (0.03). That p value gives *critical* severity level to the patient. In the error propagation path, we show that the error is propagated from the controller to the actuator [*Device_Controller_Monitor.impl* $\rightarrow$ *Pulse_Generator.impl*] through the error propagation path. Also, in the error propagation information, the controller gathers information about the error and sends it to the actuator such as name of the error, port to propagate, hazard's severity level of the error and p value. Now, to mitigate the propagated error in the actuator, *Pulse_Generator.impl* provides a safety constraint for it. For example, *"SC2: The pulse generator should receive the pacing command on time from DCM"* is used to mitigate the effects of the error in the controller's command. This mitigation process has been executed in the *Pulse_Generator.impl* as a destination point. Finally, we show that the source of the error is the controller *(C)* or *Device_Controller_Monitor.impl* and mitigated in the actuator *(A)* or *Pulse_Generator.impl*

3. Now, we know that how the *Device_Controller_Monitor.impl* handles incoming errors from the *Electrode.impl* and we also show that the *Pulse_Generator.impl* handles the incoming errors from the *Device_Controller_Monitor.impl*. But, what will happen if the *Pulse_Generator.impl* itself has an internal failure? For that purpose, we have recorded an internal failure (INTF3) in the *Pulse_Generator.impl* which causes *DelayedPacingService* error. This error leads to delaying the pace service to the heart. The probability of occurrence for that error during execution is (0.04). That p value gives a *critical* severity level. Previously, we show how *Pulse_Generator.impl* mitigates the incoming propagated errors from *Device_Controller_Monitor.impl*. Now, we need to show how the *Pulse_Generator.impl* prevents errors that have been produced by itself. In the error propagation path, we show that the source of the error is *Pulse_Generator.impl* and the actuator has sufficient information about the error such as name of the produce error, port to propagate, p value and hazard's severity level. To prevent *DelayedPacingService* error, the *Pulse_Generator.impl* has a safety constraint for it. For example, *"SC3: The pulse generator should not provide the late pacing service to the heart"* is used to mitigate the effects of the error. This mitigation process has been executed in the *Pulse_Generator.impl* as a destination point. Finally, we show that the source of the error is the actuator *(A)* or *Pulse_Generator.impl* and has been mitigated in the *(A)* or *Pulse_Generator.impl* itself.

4. What will happen if the *Pulse_Generator.impl* is not able to prevent *DelayedPacingService* error? As shown in figure 4.14, the error propagation analysis shows that the *DelayedPacingService* error has been propagated from the actuator to the sensor [*Pulse_Generator.impl* $\rightarrow$ *Heart.impl* $\rightarrow$ *Electrode.impl*]. This means that the *Pulse_Generator.impl* updates the *Electrode.impl*'s information in a late time. This leads to the *Device_Controller_Monitor.impl* not being able to pace the heart when it is required because of the *Pulse_Generator.impl*'s internal failure. If we assume that the *Device_Controller_Monitor.impl* sends another pulse command to the *Pulse_Generator.impl* in a late time. Also, the *Pulse_Generator.impl* executes the second command in a late time as well. This leads to accumulating time of delay pacing service to the heart. This operation does not help to raise the patient's heart beating because every sensing operation needs pacing action on time. To ensure that the error is propagated and has not been mitigated, the figure 4.14 shows empty cells in some specific parameters such as mitigated error by safety constraints, mitigated in component name, and mitigated by component type.



Component Type	Propogation Path	Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component Ty
S	Electrode.impl->Device_Controller_Monitor.impl	LateMeasureSensing{Measur...	SC1: The DCM should receive heart sensing valu...	Device_Controller_Monitor.impl	C
C	Device_Controller_Monitor.impl->Pulse_Generator.impl	LatePacingCommand{Send_...	SC2: The pulse generator should receive the paci...	Pulse_Generator.impl	A
A	Pulse_Generator.impl->Heart.impl->Electrode.impl	DelayedPacingService{Heart_...			

Figure 4.14: ASAM's report for Non-Mitigated Error in pacemaker

4.2.5 Summary

In this evaluation, we have taken three examples which contain different types of errors such as value errors in ACC example, service errors in train door and timing errors in pacemaker. Each example has been evaluated based on mitigated and non-mitigated errors. The result of each example has been showed based on the ASAM's generated report.

The purpose of this evaluation is to demonstrate that ASAM mitigates errors by safety constraints for different kinds of safety-critical system examples. ASAM generates a report based on the following information: identify internal failure for each major component, each internal failure causes error, probability value of error occurrence, thresholds to identify probability of which hazard could occur, identify major component's implementation & specification parts, propagation path to identify error mitigation process step by step, and mitigate error effects by safety constraints in the component's implementation parts. Finally, in each example, we have also demonstrated the ASAM's ability, to handle incoming errors and prevent outgoing errors by safety constraints, for each specific component in the feedback control loop architecture.

4.3 Demonstrating Safety Constraints Verification

Claim C3: ASAM uses verification, unlike existing safety analysis methods, to further ensure that the safety constraint fully mitigates the hazardous condition improving the quality of the safety constraints.

4.3.1 Evaluation Plan

To demonstrate claim C3, we use ASAM to verify the safety constraints against the system model with injected errors. This allows us to determine that either unsafe behavior occurs (which means the error leads to a hazardous condition in the system) or verify that the error does not lead to unsafe behavior.

To evaluate this claim, we need to add a new piece, *verified by [Guarantee Identifier]*, to the safety constraint statements of ASAM and pass the identifiers to the guarantee statements in XAGREE (eXtended Assume Guarantee REasoning Environment) which is described in section 2.4. The identifier of the guarantee statement is used in the *verified by* statements of ASAM. When we invoke either activity (report or analysis) in ASAM, XAGREE is spawned in the background and as each component of the feedback control loop is encountered, a *verify all* activity in XAGREE is spawned. ASAM waits for XAGREE to complete the analysis and then it matches each safety constraint with the XAGREE results. If XAGREE verified the guarantee(s) associated with the safety constraint statement, then the safety constraint is reported as verified. If any guarantee associated with a safety constraint fails verification, the safety constraint is marked as unverified.

4.3.2 Safety Constraints Verification for Train Automated Door Control System

We will provide and discuss safety constraint verification for the feedback control loop architecture of the train automated door control system. We will verify the safety constraints that have been placed in the implementation of each component. The specification part of the component ensures that the component does not have unsafe behavior by listing the verification conditions using XAGREE. In this example, we will verify three important safety safety constraints of the train door, which are:

1. SC1: The train door should not be blocked when it is opened.
2. SC2: The train door should not be blocked when it is closed.
3. SC3: The train door can not be opened and closed at the same time (i.e, open and then close, close and then open, not both together).

```
system door_sensor
    features
        door_blocked_s: in data port Base_Types::Integer;
        door_open_s: out data port Base_Types::Boolean;
        door_close_s: out data port Base_Types::Boolean;
    annex xagree {**
        --Identify door sensor input states
        assume "Door in block states": door_blocked_s >= 0;
        --Identify door sensor output identifiers
        GAU1: guarantee "Door is blocked only if it opens": (door_blocked_s = 1
            and door_open_s = true) or door_open_s = false;
        GAU2: guarantee "Door is blocked only if it closes": (door_blocked_s =
            0 and door_close_s = true) or door_close_s = false;
    **};
end door_sensor;
```

Listing 4.26: XAGREE Identifiers for Train Door Sensor Specification

In listing 4.26, we have presented the *door_sensor* specification. Notice that identifiers have been annotated in the XAGREE annex. In the specification, the interface has two out data ports, *door_open_s* and *door_close_s*, which are used to send the open and close status of the door to the controller. The interface also has an in data port, *door_blocked_s*, that allows the sensor to receive the blocked status value from the physical door sensors. In the XAGREE annex, we have preconditions for the in data port. Also, we have postconditions and identifiers, GAU1 and GAU2, for the out data ports.

```
system implementation door_sensor.impl
    annex asam {**
        type => sensor;
        Errs => [{
            Type => ValueOpenedError(Critical,p=0.1),
            UCA => UCA1: "Door report it is both opened and blocked",
            Causes => {
                General => "A signal was generated wrong when door is opened"
            },
            --Verify door sensor outputs by safety constraints identifiers.
            SC => SC1: "Door should not be blocked when the door is opened"
                verified by [GAU1]
        },
        {
            Type => ValueClosedError(Critical,p=0.1),
            UCA => UCA2: "Door report it is both closed and blocked",
            Causes => {
                General => "A signal was generated wrong when door is closed"
            },
            --Verify door sensor outputs by safety constraints identifiers.
            SC => SC2: "Door should not be blocked when the door is closed"
                verified by [GAU2]
        }]
    **};
    --Satisfy each status of the door sensor
    annex xagree {**
        assign door_open_s = (door_blocked_s = 1);
        assign door_close_s = (door_blocked_s = 0);
 **};
end door_sensor.impl;
```

Listing 4.27: ASAM and XAGREE for Train Door Sensor Implementation

In listing 4.27, we have presented ASAM and XAGREE in the *door_sensor* implementation. ASAM matches the guarantee identifiers, GAU1 and GAU2, to the guarantee statements in the XAGREE specification shown in list 4.26. These identifiers are used to verify the safety constraints SC1 and SC2 respectively to ensure that the train door is not blocked when it is opened or closed. When we call XAGREE, it checks all activities in the specification and implementation parts of the *door_sensor* and returns the result to ASAM. At that time, ASAM waits for the XAGREE response and then it matches each safety constraint with the XAGREE results. If XAGREE verified the guarantee identifiers, GAU1 and GAU2, associated with the safety constraint, SC1 and SC2, then ASAM reports that the safety constraints have been verified.

```
--The data message comes from the door sensor, it contains two status at the
    same time
data message
```

```
end message;

data implementation message.impl
    subcomponents
        open_door_c : data base_types::Boolean;
        close_door_c : data base_types::Boolean;
end message.impl;

--The door controller specification
system door_controller
    features
        door_open_and_close_msj: in data port message.impl;
        door_blocked_c: in data port Base_Types::Integer;
        open_door_c: out data port Base_Types::Boolean;
        close_door_c: out data port Base_Types::Boolean;
    annex xagree {**
        assume "Door states in": door_blocked_c <= 2 and door_blocked_c >=0;
        GAU3: guarantee "Door cannot be opened and closed at the same time" :
            not (open_door_c and close_door_c);
 **};
end door_controller;
```

Listing 4.28: XAGREE for Train Door Controller Specification

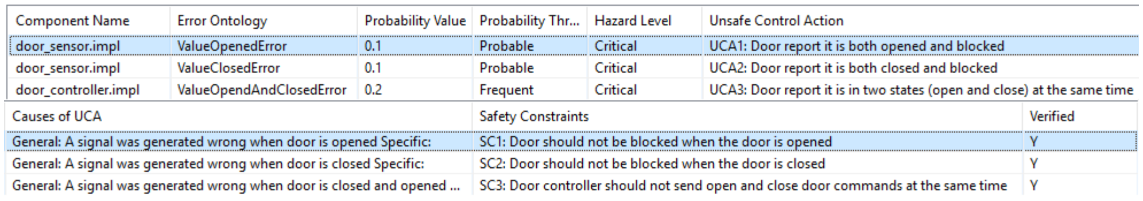
In listing 4.28, we present a message object, read by the *door_sensor*, containing two status at the same time. In this example, only one of the boolean flags of the message will be set at a time; there should not be an instance where both are set. Also, we present the *door_controller* specification. In the specification, the interface has two out data ports, *open_door_c* and *close_door_c*, which are used to send commands to open and close the train door. The interface also has two in data ports, *door_open_and_close_msj* and *door_blocked_c*. The *door_open_and_close_msj* is used to receive the data messages that have been read by the *door_sensor*. The *door_blocked_c* is used to receive the blocked status of the train door. In the XAGREE annex, we have preconditions for the in data port. Also, we have a postcondition ensuring that the *door_controller* does not send both open and close door commands at the same time.

```
system implementation door_controller.impl
```

```
    annex asam {**
        type => controller;
        Errs => [{
            Type => ValueOpendAndClosedError(Critical,p=0.2),
            UCA => UCA3: "Door report it is in two states (open and close) at
                the same time",
            Causes => {
                General => "A signal was generated wrong when door is closed
                    and opened"
            },
            -- Verify the safety constraint identifier
            SC => SC3: "Door controller should not send open and close door
                commands at the same time" verified by [GAU3]
        }]
    **};
    annex xagree{**
        --Satisfy that open and close have the same status
        eq door_open_msj_c : bool = false -> if
            ((door_open_and_close_msj.open_door_c = true) and
            (door_open_and_close_msj.close_door_c = false)) then true else
            false;
        eq door_close_msj_c : bool = false -> if
            ((door_open_and_close_msj.close_door_c = true) and
            (door_open_and_close_msj.open_door_c = false)) then true else false;
        --Satisfy that the data message do not lead to block the door when it
            is opened or closed
        assert (open_door_c = (door_open_msj_c and (door_blocked_c <= 2)));
        assert (close_door_c = (door_close_msj_c and (door_blocked_c >= 0)));
    **};
end door_controller.impl;
```

Listing 4.29: ASAM and XAGREE for Train Door Controller Implementation

In listing 4.29, we have both ASAM and XAGREE in the *door_controller* implementation. ASAM provides an identifier, GAU3, to the guarantee statement in the XAGREE specification of the train door controller, as shown in listing 4.28, to verify the safety constraint SC3. When ASAM calls XAGREE, it checks all activity of the train *door_controller* in the specification / implementation and returns the result to ASAM. As we show that in the *door_controller* implementation, XAGREE checks the status of the open and close flags in the data message *door_open_and_close_msj*. Also, XAGREE has to check that the *door_sensor* data message does not allow the door to enter a state where it is blocked while open or closed.



Component Name	Error Ontology	Probability Value	Probability Thr...	Hazard Level	Unsafe Control Action
door_sensor.impl	ValueOpenedError	0.1	Probable	Critical	UCA1: Door report it is both opened and blocked
door_sensor.impl	ValueClosedError	0.1	Probable	Critical	UCA2: Door report it is both closed and blocked
door_controller.impl	ValueOpendAndClosedError	0.2	Frequent	Critical	UCA3: Door report it is in two states (open and close) at the same time

Causes of UCA	Safety Constraints	Verified
General: A signal was generated wrong when door is opened Specific:	SC1: Door should not be blocked when the door is opened	Y
General: A signal was generated wrong when door is closed Specific:	SC2: Door should not be blocked when the door is closed	Y
General: A signal was generated wrong when door is closed and opened ...	SC3: Door controller should not send open and close door commands at the same time	Y

Figure 4.15: ASAM's report about safety constraints verification for train door

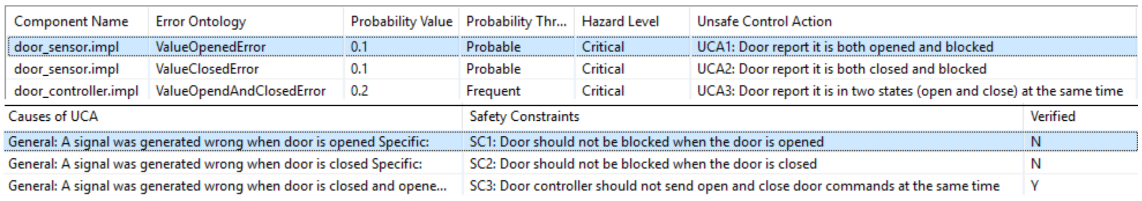
Figure 4.15 shows ASAM's generated report about safety constraint verification for the train automated door control system. ASAM generates the report based on the following information: component's implementation part, error ontology, probability occurrence of the specific error, probability thresholds to identify hazard's severity level, unsafe control action, causes of unsafe control action, safety constraints, and verification of the safety constraints. This stream of information was previously described in other examples. In this section of the report, we will focus on the verification parts of the safety constraints.

1. The unsafe control action (UCA1) in the *door_sensor.impl* tells us that the train door has been blocked when it is opened. The general cause for this unsafe action is a wrong signal generated by the sensor when the door is opened, or an internal failure of the sensor. To solve this problem, we need to provide a safety constraint, SC1, to show that the door should not be blocked when it is opened. At the same time, we need to ensure that the train door is opened correctly/safely and it is not blocked. For that purpose, we provide a guarantee identifier, GAU1, for the safety constraint, SC1. ASAM sends the identifier to XAGREE to verify the activity of the *door_sensor.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (Yes) showing verification of the safety constraint, SC1. This verification means that the *door_sensor.impl* cannot be blocked when the door is opened.

2. The unsafe control action (UCA2) in the *door_sensor.impl* tells us that the train door has been blocked when it is closed. The general cause for this unsafe action is a wrong signal generated by the sensor when the door is closed, or an internal failure of the sensor. To solve this problem, we need to provide a

safety constraint, SC2, to show that the door should not be blocked when it is closed. At the same time, we need to ensure that the train door is closed correctly/safely and it is not blocked. For that purpose, we provide a guarantee identifier, GAU2, for the safety constraint, SC2. ASAM sends the identifier to XAGREE to verify the activity of the *door_sensor.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (Yes) showing verification of the safety constraint, SC2. This verification means that the *door_sensor.impl* cannot be blocked when the door is closed.

3. The unsafe control action (UCA3) in the *door_controller.impl* tells us that the incoming data message from the *door_sensor.impl* contains two status (open and close) at the same time. The general cause for this unsafe action is a wrong signal generated by the sensor when the door is opened or closed, or the controller has an internal failure. To solve this problem, we need to provide a safety constraint, SC3, to show that the *door_controller.impl* should not send both open and close commands at the same time. Also, we need to ensure that the *door_controller.impl* does not send the wrong command. For that purpose, we provide a guarantee identifier, GAU3, for the safety constraint, SC3. ASAM sends the identifier to XAGREE to verify the activity of the *door_controller.impl* and then returns the result (Yes/No) to ASAM. ASAM reports the result (Yes) showing verification of the safety constraint, SC3. This verification means that the *door_controller.impl* does not send inadequate commands, or two status at the same time.



Component Name	Error Ontology	Probability Value	Probability Thr...	Hazard Level	Unsafe Control Action
door_sensor.impl	ValueOpenedError	0.1	Probable	Critical	UCA1: Door report it is both opened and blocked
door_sensor.impl	ValueClosedError	0.1	Probable	Critical	UCA2: Door report it is both closed and blocked
door_controller.impl	ValueOpendAndClosedError	0.2	Frequent	Critical	UCA3: Door report it is in two states (open and close) at the same time

Causes of UCA	Safety Constraints	Verified
General: A signal was generated wrong when door is opened Specific:	SC1: Door should not be blocked when the door is opened	N
General: A signal was generated wrong when door is closed Specific:	SC2: Door should not be blocked when the door is closed	N
General: A signal was generated wrong when door is closed and opene...	SC3: Door controller should not send open and close door commands at the same time	Y

Figure 4.16: ASAM's report about safety constraints are not verified for the train door

Figure 4.16 shows ASAM's generated report about safety constraints of the *door_sensor* status in the train automated door control system are not verified. We will describe the reasons as follows:

1. As a reminder, we need to verify the safety constraint to ensure the door is not blocked when it is opened. For that purpose, we provide a guarantee identifier, GAU1, for the safety constraint, SC1. ASAM sends the identifier to XAGREE to verify the activity of the *door_sensor.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (NO or N) showing the safety constraint, SC1,

is not verified. This verification means that the door is blocked when the it is opened. This means the door will not return to different status because it is blocked in that situation. ASAM verified that the door is blocked when it is opened because ASAM did not obtain the true value from XAGREE to ensure the door is not blocked when it is opened.

2. Also, we need to verify the safety constraint to ensure the door is not blocked when it is closed. For that purpose, we provide a guarantee identifier, GAU2, for the safety constraint, SC2. ASAM sends the identifier to XAGREE to verify the activity of the *door_sensor.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (NO or N) showing the safety constraint, SC2, is not verified. This verification means that the door is blocked when the it is closed. This means the door will not return to different status because it is blocked in that situation. ASAM verified that the door is blocked when it is closed because ASAM did not obtain the true value from XAGREE to ensure the door is not blocked when it is closed.

4.3.3 Safety Constraints Verification for Pacemaker

We will provide and discuss safety constraint verification for the pacemaker feedback control loop architecture. We will verify the safety constraints that have been fed into the implementation part of each individual component do not have unsafe behavior or that the hazard effects have been mitigated. In this example, we will verify three important safety constraints of the pacemaker, which are:

1. SC1: The electrode should not be blocked when it is sensing.

2. SC2: The electrode should not be blocked when the heart is paced.

3. SC3: The heart should not be sensed and paced at the same time (i.e, sense and then pace, not both together).

```
system Electrode
    features
        Electrode_Blocked_s: in data port Base_Types::Integer;
        Measured_Sensing: out data port Base_Types::Boolean;
        Measured_Pacing: out data port Base_Types::Boolean;
    annex xagree {**
        assume "Electrode in blocked states": Electrode_Blocked_s >= 0;
        GAU1: guarantee "Electrode is blocked only if it is in measured
            sensing": (Electrode_Blocked_s = 0 and Measured_Sensing = true) or
            Measured_Sensing = false;
        GAU2: guarantee "Electrode is blocked only if it is heart paced":
            (Electrode_Blocked_s = 1 and Measured_Pacing = true) or
            Measured_Pacing = false;
    **};
end Electrode;
```

Listing 4.30: XAGREE Identifiers for Electrode Specification

In listing 4.30, we show the *Electrode* sensor specification along with preconditions and postconditions for the operation of the sensor. In the specification, the interface has two out data ports, *Measured_Sensing* and *Measured_Pacing*, which are used to send *sensed* and *paced* values of the heart to *Controller_Monitor_Device*. The interface also has an in data port, *Electrode_Blocked_s* that allows the *Electrode* to receive the *blocked* status value from the heart.

```
system implementation Electrode.impl
    annex asam{**
        --Identify type of major component
        type => sensor;
        --Electrode is the sensor of the pacemaker. The internal failure in it
            causes electrode error.
        internal failure INTF1:"The electrode report it is blocked in measured
            sensing" causes [MeasuredSensingError(Critical,p=0.02)] on
            [Measure_Sensing];
        internal failure INTF2:"The electrode report it is blocked when the
            heart is paced" causes [MeasuredPacingError(Critical,p=0.02)] on
            [Measured_Pacing];
        safety constraint SC1: "The electrode should not be blocked when it is
            sensing" prevents [MeasuredSensingError] on [Measure_Sensing]
            verified by [GAU1];
        safety constraint SC2: "The electrode should not be blocked when the
            heart is paced" prevents [MeasuredPacingError] on [Measured_Pacing]
            verified by [GAU2];
    **};
    annex xagree {**
        --Satisfy each status of the electrode
        assign Measured_Sensing = (Electrode_Blocked_s = 0);
        assign Measured_Pacing = (Electrode_Blocked_s = 1);
    **};
end Electrode.impl;
```

Listing 4.31: ASAM and XAGREE for Electrode Implementation

In listing 4.31, we show ASAM and XAGREE in the *Electrode* sensor implementation. ASAM gives the identifiers, GAU1 and GAU2, to the guarantee statements in the XAGREE specification as shown in listing 4.30. These identifiers are used to verify the safety constraints, SC1 and SC2, to ensure that the *Electrode* is not blocked when it is sensing the heart or when the heart is paced. If any guarantee associated with safety constraints fail verification, then ASAM will report that the safety constraint are marked as unverified.

```
--The data message comes from the electrode sensor, it contains two status at
    the same time, sensing and pacing
data Electrode_message
end Electrode_message;

data implementation Electrode_message.impl
    subcomponents
        DCM_sensing_c : data Base_Types::Boolean;
        DCM_pacing_c : data Base_Types::Boolean;
end Electrode_message.impl;

system Device_Controller_Monitor
    features
        DCM_sensing_and_pacing_msj: in data port Electrode_message.impl;
        DCM_Blocked_c: in data port Base_Types::Integer;
        DCM_sensing_c: out data port Base_Types::Boolean;
        DCM_pacing_c: out data port Base_Types::Boolean;
    annex xagree {**
        assume "DCM states in": DCM_Blocked_c <= 2 and DCM_Blocked_c >=0;
        GAU3: guarantee "Heart cannot be sensed and paced at the same time,
            sense and then space": not (DCM_sensing_c and DCM_pacing_c);
    **};
end Device_Controller_Monitor;
```

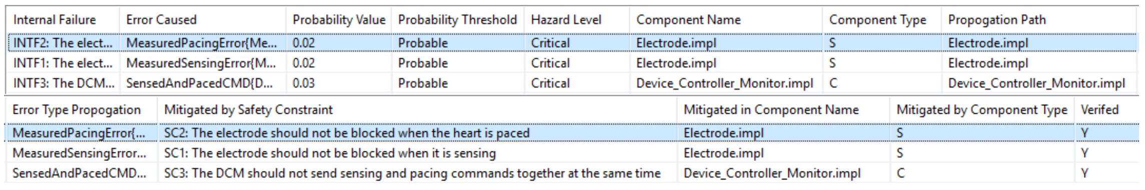
Listing 4.32: XAGREE for Device Controller Monitor Specification

In listing 4.32, we show a data message read by the *Electrode* containing two statuses at a time. This means the message could contain both sensing and pacing information simultaneously. In reality, the pacemaker should first sense and then pace the heart with respect to sensing values. We also show the *Device_Controller_Monitor* specification with identifiers in the XAGREE part. In the specification of the controller, the interface has two out data ports, *DCM_sensing_c* and *DCM_pacing_c*, to send sensing and pacing commands. The interface also has two in data ports, *DCM_sensing_and_pacing_msj* and *DCM_Blocked_c* to receive different types of messages. The *DCM_sensing_and_pacing_msj* is used to receive the data messages that have been read and sent by the *Electrode* sensor. The *DCM_Blocked_c* is used to receive the blocked status values of the *Electrode* sensor. In the XAGREE part, we show a precondition for the in data port to identify blocked status values of the *Device_Controller_Monitor*. Also, we show a postcondition identifier or guarantee identifier, GAU3, for the out data ports of the controller, *DCM_sensing_c* and *DCM_pacing_c*, to identify that the *Device_Controller_Monitor* does not send sensing and pacing commands together at the same time.

```
system implementation Device_Controller_Monitor.impl
    annex asam{**
        --Identify type of major component
        type => controller;
        --DCM is the controller of pacemaker. The internal failure in it causes
            DCM error.
        internal failure INTF3:"The DCM report that heart sensed and paced at
            the same time" causes [SensedAndPacedCMD(Critical,p=0.03)] on
            [DCM_sensing_c];
        safety constraint SC3:"The DCM should not send sensing and pacing
            commands together at the same time" prevents [SensedAndPacedCMD] on
            [DCM_sensing_c] verified by [GAU3];
    **};
    annex xagree{**
        --Satisfy that sensing and pacing have the same status
        eq DCM_sensing_msj_c : bool = false -> if
            ((DCM_sensing_and_pacing_msj.DCM_sensing_c = true) and
            (DCM_sensing_and_pacing_msj.DCM_pacing_c = false)) then true else
            false;
        eq DCM_pacing_msj_c : bool = false -> if
            ((DCM_sensing_and_pacing_msj.DCM_pacing_c = true) and
            (DCM_sensing_and_pacing_msj.DCM_sensing_c = false)) then true else
            false;
        --Satisfy that electrode data message should not lead to block the DCM
            when it is the sensing and pacing have the same status
        assert (DCM_sensing_c = (DCM_sensing_msj_c and (DCM_Blocked_c <= 2)));
        assert (DCM_pacing_c = (DCM_pacing_msj_c and (DCM_Blocked_c >=0)));
    **};
end Device_Controller_Monitor.impl;
```

Listing 4.33: ASAM and XAGREE for Device Controller Monitor Implementation

In listing 4.33, we show ASAM and XAGREE in the *Device_Controller_Monitor* implementation. ASAM gives an identifier, GAU3, to the guarantee statement in the XAGREE specification of the controller as shown in listing 4.32, to verify the safety constraint, SC3, to ensure that the controller of the pacemaker does not send sensing and pacing commands together at the same time. For that purpose, when we call the XAGREE, it checks all activity of the *Device_Controller_Monitor* in the specification and implementation parts and returns the result to ASAM.



Internal Failure	Error Caused	Probability Value	Probability Threshold	Hazard Level	Component Name	Component Type	Propogation Path
INTF2: The elect...	MeasuredPacingError{Me...	0.02	Probable	Critical	Electrode.impl	S	Electrode.impl
INTF1: The elect...	MeasuredSensingError{M...	0.02	Probable	Critical	Electrode.impl	S	Electrode.impl
INTF3: The DCM...	SensedAndPacedCMD{D...	0.03	Probable	Critical	Device_Controller_Monitor.impl	C	Device_Controller_Monitor.impl

Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by Component Type	Verifed
MeasuredPacingError{...	SC2: The electrode should not be blocked when the heart is paced	Electrode.impl	S	Y
MeasuredSensingError...	SC1: The electrode should not be blocked when it is sensing	Electrode.impl	S	Y
SensedAndPacedCMD...	SC3: The DCM should not send sensing and pacing commands together at the same time	Device_Controller_Monitor.impl	C	Y

Figure 4.17: ASAM's report about safety constraints verification for pacemaker

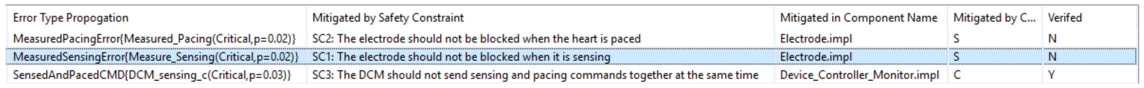
Figure 4.17 shows ASAM's report about the safety constraints verification for the pacemaker. ASAM generates the report based on the following information: component's implementation part, error ontology, probability occurrence of the specific error, probability thresholds to identify hazard's severity level, unsafe control action, causes of unsafe control action, safety constraints, and verification of the safety constraints. This stream of information is described previously in detail for different types of examples. Now, we need to focus on the verification part of the safety constraints. We iteratively explain figure 4.17 below:

1. The internal failure (INTF1) in the *Electrode.impl* tells us that the electrode sensor has been blocked when it is sensing the heart. This failure gives *MeasuredSensingError* to the *Electrode.impl*. We need a safety constraint to mitigate the effects of the error. For that purpose, we provide a safety constraint, SC1, to show that the electrode sensor should not be blocked because of the error when it is sensing the heart. This is not a sufficient mechanism to mitigate the effects of the error only. Also, we need to ensure that *Electrode.impl* is sensing the heart correctly/safely and it is not blocked. For this case, we provide a guarantee identifier, GAU1, for the safety constraint, SC1. ASAM sends the guarantee identifier to XAGREE to verify the activity of the *Electrode.impl* specification / implementation and returns the result (Yes/No) to ASAM. Then, ASAM reported (Yes) as a verification of the safety constraint, SC1. This verification means that the *Electrode.impl* is not blocked when it senses the heart.

2. The internal failure (INTF2) in the *Electrode.impl* tells us that the electrode sensor has been blocked when the heart is paced. This failure gives *MesuredPacingError* to the *Electrode.impl*. We need a safety constraint to mitigate the effects of the error. For that purpose, we provide a safety constraint, SC2, to show that the electrode sensor should not be blocked because of the error when the heart is paced. That is not sufficient mechanism to mitigate the effects of the error only. Also, we need to ensure that the *Electrode.impl* sensor has not been blocked when the heart is paced. For this case, we provide a guarantee identifier, GAU2, for the safety constraint, SC2. ASAM sends the guarantee identifier to XAGREE to verify the activity of the *Electrode.impl* specification / implementation and returns the result (Yes/No) to ASAM. Then, ASAM reported (Yes) as a verification of the safety constraint, SC2. This verification means that the *Electrode.impl* is not blocked when the heart is paced.

3. The internal failure (INTF3) in the *Device_Controller_Monitor.impl* tell us that the heart is sensed and paced together at the same time. This failure gives *SensedAndPacedCMD* error to the controller. We need a safety constraint to mitigate the effects of the error. For that purpose, we provide a safety constraint, SC3, to show that the controller should not send sensing and pacing commands together at the same time. That is not sufficient mechanism to mitigate the effects of the error only. Also, we need to ensure that the controller do not send the wrong command to the heart. For this case, we provide a

guarantee identifier, GAU3, for the safety constraint, SC3. ASAM sends the identifier to XAGREE to verify the activity of the *Device_Controller_Monitor.impl* specification / implementation and returns the result (Yes/No) to ASAM. Then, ASAM reported (Yes) as a verification of the safety constraint, SC3. This verification means that the *Device_Controller_Monitor.impl* do not send the inadequate command to the heart such as sensing and pacing commands together to the heart.



Error Type Propogation	Mitigated by Safety Constraint	Mitigated in Component Name	Mitigated by C...	Verifed
MeasuredPacingError{Measured_Pacing(Critical,p=0.02)}	SC2: The electrode should not be blocked when the heart is paced	Electrode.impl	S	N
MeasuredSensingError{Measure_Sensing(Critical,p=0.02)}	SC1: The electrode should not be blocked when it is sensing	Electrode.impl	S	N
SensedAndPacedCMD{DCM_sensing_c(Critical,p=0.03)}	SC3: The DCM should not send sensing and pacing commands together at the same time	Device_Controller_Monitor.impl	C	Y

Figure 4.18: ASAM's report about safety constraints are not verified for the pacemaker

Figure 4.18 shows ASAM's generated report about safety constraints of the *Electrode.impl* status in the pacemaker system are not verified. We will describe the reasons as follows:

1. As a reminder, we need to verify the safety constraint of the *Electrode.impl* to ensure that the electrode is not blocked when it is sensing. For that purpose, we provide a guarantee identifier, GAU1, for the safety constraint, SC1. ASAM sends the identifier to XAGREE to verify the activity of the *Electrode.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (NO or N) showing the safety constraint, SC1, is not verified. This verification means that the electrode sensor is blocked when it is sensing. This means the electrode will not sense any more because it is blocked in that status. ASAM verified that the electrode sensor is blocked when it is sensed because ASAM did not obtain the true value from XAGREE to ensure the electrode is not blocked when it is sensed.

2. Also, we need to verify the safety constraint of the *Electrode.impl* to ensure that the electrode is not blocked when the heart is paced. For that purpose, we provide a guarantee identifier, GAU2, for the safety constraint, SC2. ASAM sends the identifier to XAGREE to verify the activity of the *Electrode.impl* and then returns the result (Yes/No) to ASAM. ASAM reports that result (NO or N) showing the safety constraint, SC2, is not verified. This verification means that the electrode sensor is blocked when the heart is paced. This means the electrode will not pace the heart any more because it is blocked in that situation. ASAM verified that the electrode sensor is blocked when the heart is paced because ASAM did not obtain the true value from XAGREE to ensure the electrode sensor is not blocked when the heart is paced.

4.3.4 Summary

In this evaluation, we have taken two important safety-critical system examples, the train automated door control system and a medical embedded device (i.e. pacemaker). In each example, we have multiple safety constraints, (SC1, SC2, and SC3) and multiple guarantee identifiers (GAU1, GAU2, and GAU3) which are used to verify the safety constraints. The result of each example has been evaluated based on XAGREE analysis and ASAM's generated report.

The purpose of this evaluation is to demonstrate that ASAM uses verification, unlike existing safety analysis methods, to prove that the safety constraint fully mitigates the hazardous condition improving the quality of the safety constraints.

The result tells us that ASAM verified the safety requirements for different kinds of examples in the safety-critical domains. We provided safety constraints in the implementation part for each individual core component in the feedback control loop architecture. We also provided guarantee identifier for each safety constraint within the component implementation. We adopted XAGREE to check the activity of the identifiers in the specification and implementation of the component and then it returns the result to ASAM. In each example, we have also demonstrated that ASAM generate a report to verify the result. In summary, we show how to ensure the error effects have been mitigated by verifying safety the constraints within the architecture model.

4.4 Demonstrating Notations and Expressions for Error Flows and Safety Constraints

In this section, we will show that how to identify the notations in the ASAM's implementation. Essentially, we are going to say that the tooling shown in claims 1 - 3 is an implementation of the mathematical framework. We described in detail the theoretical foundation of notation and expression in section 3.3.

As a reminder, previously, we provide the mathematical notation for the feedback control loop architecture and provide formal specification for the error ontology in the loop. By doing so, we allow stakeholders to find more unsafe possibilities in the operational system context. Additionally, we provide an expression for each component in the feedback control loop and use error types in the form of formal specifications to find more possibilities for how the system could behave incorrectly. Also, we identify unsafe functional behavior for each component, and we provide safety constraints for possibilities of unsafe functions in the system because if the system knows about those possibilities, it can take action to avoid incorrect behavior.

We will show the notations in ASAM's implementation to identify the locations of the variables in the feedback control loop architecture such as measured variables (*Vme*), control actions (*CA*), manipulated variables (*Vma*), and controlled variables(*Vc*). We also show the other elements of the notation in ASAM such as safety constraints (*SC*), internal failures (*INTF*), errors (*Err*), the error ontology (*EO*), guarantee identifiers (*GAU*), unsafe control actions (*UCA*), probability values (*P*), hazards types (*H*), major component types such as sensors (*S*), controllers (*C*), actuators (*A*) and controlled processes (*CP*). These elements help the stakeholders use our notation in the safety-critical examples.

4.4.1 Identify Notations in ASAM's Implementation

When the safety analysts or stakeholders run ASAM, they will see our notations that have been executed within the architecture model to generate a report. We have presented the notations in the form of executed safety analysis statements in ASAM such as error statements, error propagation statements, internal failure statements, safety constraint statements that either handle or prevent hazards, and safety constraint verification statements. In this section, we will provide and discuss the notations that we have implemented in ASAM.

We have presented different types of variables in the feedback control loop architecture such as measured variables (*Vme*), control actions (*CA*), manipulated variables (*Vma*) and controlled variables (*Vc*). We also identify major component types such as sensors (*S*), controllers (*C*), actuators (*A*), and controlled processes (*CP*). We have also specified the location of the variables among the components. For example, we have presented (*Vme*) between (*S* and *C*), (*CA*) between (*C* and *A*), (*Vma*) between (*A* and *CP*), (*Vc*) between (*CP* and *S*).

Each component in the feedback control loop could has an internal failure (*INTF*) which leads to an error (*Err*) in the variable. In this case, the stakeholders can select any of the six (*Type*) errors in the error ontology (*EO*) such as (*Service Errors, Value Errors, Timing Errors, Replication Errors, Concurrency Errors, Access Control Errors*) as an effect of the internal failure. In the our application examples, we have focused on the first three. Also, each error causes an unsafe control action (*UCA*) in the component. We identify the general and specific cause for that error. To mitigate the effects of the (*Err*), we provide safety constraints (*SC*), to handle and prevent incoming and outgoing (*Err*)s. We also have represented the probability value (*P*) of error (*Err*)'s occurrence as well as the severity level of the the (*Err*) such as (*Catastrophic, Critical, Marginal, Negligible*). Finally, we show the notation in the components implementation within the architecture model. Also, XAGREE helped ASAM to verify the guarantee identifiers (*GAU*) in the specification of the components. ASAM wants to verify the safety constraint (*SC*) identifiers that have been fed in the implementation by specification using XAGREE to ensure that the (*Err*) effects have been mitigated or hazardous condition could not occur. In the following example, we just remind the reader how to identify our notations in ASAM:

```
system S
    features
        --Controlled Variables (Vc) of the Sensor (S)
        Vc_S: in data port Base_Types::Integer;
        -- Measured Variable (Vme) of the Sensor (S)
        Vme_S: out data port Base_Types::Boolean;
        --Check the sensor's specification input and output
    annex xagree {**
        assume "Controlled variable of the sensor's input'": Vc_S >= 0;
        GAU1: guarantee " Check measured variable with respect to controlled
            variable ": (Vc_S = 1 and Vme_S = true) or Vme_S = false;
    **};
end S;
```

Listing 4.34: Notations in ASAM

In listing 4.34, we show controlled variable (*Vc*) as an in data port and also have represented measured variable (*Vme*) as an out data port. We also have used XAGREE to check the sensor, (*S*)'s, input and output specifications respectively. XAGREE provides conditions for the controlled variable (*Vc*) and provides a guarantee identifier (*GAU1*) for the measured variable (*Vme*).

```
system implementation S.impl
    annex asam {**
        type => sensor;
        internal failure INTF1: "Sensor internal failure statement" causes
            [EO_S(Critical,p=0.02)] on [Vme_S];
        safety constraint SC1: "Sensor safety constraint statement for the
            sensor's measured variable" prevents [EO_S] on [Vme_S] verified by
            [GAU1];
    **};
    annex xagree {**
        assign Vme_S = (Vc_S = 1); --Return true when the Vc is correct
    **};
end S.impl;
```

Listing 4.35: Notations in ASAM

In listing 4.35, in the sensor's implementation part (*S.impl*), we show the type of the component (*type => sensor*), an internal failure statement (*INTF1*) which gives an error (*Err*) in the error ontology (*EO_S*) to the sensor (*S*). The error has a probability of occurrence (*P*) and has a severity level, (*Critical*). We also a safety constraint statement (*SC1*) to prevent the effects of the error on the measured variable of the sensor (*Vme_S*), which also has a guarantee identifier (*GAU1*) to that the effect has been mitigated. Also, we have use XAGREE to return true or (Yes) to ASAM if the safety constraint is correct.

```
system C
    features
        --Measured Variables (Vme) of the Controller (C)
        Vme_C: in data port Base_Types::Boolean;
        --Control Action (CA) of the Controller (C)
        CA_C: out data port  Base_Types::Integer;
end C;

system implementation C.impl
    annex asam {**
        type => controller;
        internal failure INTF2: "Controller internal failure statement" causes
            [EO_C(Marginal,p=0.04)] on [CA_C];
        --Error comes from the measured variable and causes error on control
            action.
        [EO_S] on [Vme_C] causes [EO_S(Catastrophic,p=0.002)] on [CA_C];
        --Controller have a SC to handle error(s) in measured variable.
        safety constraint SC2:"Controller safety constraint for controller's
            measured variable" handles [EO_S] on [Vme_C] verified by [GAU2];
    **};
```

```
end C.impl;
```

Listing 4.36: Notations in ASAM

In listing 4.36, we have represented the controller's (*C*) specification and implementation. In the specification, we have represented measured variables as in data port to the controller (*Vme_C*) and have represented control action as an out data port of the the controller (*CA_C*). In the implementation, we have identified the major type of the component (*type => controller*), internal failure statement (*INTF2*) of the controller which gives an error (*Err*) in the error ontology to the controller (*EO_C*). Then, the error (*Err*) will propagate to the actuator (*A*) through control action of the controller (*CA_C*). At the same time, the incoming error from the sensor (*EO_S*) arrives to the controller (*C*) through measured variable of the controller (*Vme_C*), it causes an error on the control action (*CA_C*) of the controller's output. We also have represented a safety constraint, (*SC2*), to handle the incoming error from the sensor (*EO_S*) on controller's measured variable (*Vme_C*), and we have an identifier (*GAU2*) to ensure that it is mitigated.

```
system A
    features
        -- Control Action(CA) of the Actuator (A)
        CA_A: in data port Base_Types::Integer;
        -- Manipulated Variables (Vma) of the Actuator (A)
        Vma_A: out data port Base_Types::Integer;
end A;

system implementation A.impl
    annex asam {**
        type => actuator;
        internal failure INTF3: "Actuator internal failure statement" causes
            [EO_A(Catastrophic,p=0.2)] on [Vma_A];
        --Error comes from the controller and causes error on manipulated
            variables.
        [EO_C] on [CA_A] causes [EO_C(Critical,p=0.1)] on [Vma_A];
        safety constraint SC3:"Actuator safety constraint to handle incoming
            error from control action" handles [EO_C] on [CA_A] verified by
            [GAU3];
        safety constraint SC4:"Actuator safety constraint to prevent outgoing
```

```
            error on manipulated variable" prevents [EO_A]on [Vma_A] verified
            by [GAU4];
    **};
end A.impl;
```

Listing 4.37: Notations in ASAM

In listing 4.37, we show the actuator's, (*A*), specification and implementation. In the specification, we show the control action of the actuator (*CA_A*) as an in data port and show the manipulated variables of the actuator (*Vma_A*) as an out data port. In the implementation, we have identified the type of the component (*type => actuator*) and we show an internal failure statement (*INTF3*) of the actuator which gives an error (*Err*) in the error ontology (*EO_A*). The error will propagate through the manipulated variables of the actuator (*Vma_A*) to the controlled process (*CP*). At the same time, an incoming error from controller (*EO_C*) arrives to the actuator (*A*) through the control action of the actuator (*CA_A*), it causes an error on the manipulated variables of the actuator (*Vma_A*). Also, we show a safety constraint, (*SC3*), to handle incoming errors from controller (*EO_C*) on the control action of the actuator (*CA_A*) and we have an identifier (*GAU3*) to ensure that it is mitigated. We also represent another safety constraint, (*SC4*) to prevent errors in the actuator itself (*EO_A*) and we have an identifier (*GAU4*) to ensure that they are mitigated.

```
system CP
    features
        --Manipulated Variables (Vma) of the Controlled Process (CP)
        Vma_CP: in data port Base_Types::Integer;
        --Controlled Variables (Vc) of the Controlled Process (CP)
        Vc_CP: out data port Base_Types::Integer;
end CP;

system implementation CP.impl
    annex asam {**
        type => controlled_process;
        --Pass the incoming errors from manipulated variables to controlled
            variables
        [EO_C] on [Vma_CP] causes [EO_C(Marginal,p=0.01)] on [Vc_CP];
        [EO_A] on [Vma_CP] causes [EO_A(Negligible,p=0.00001)] on [Vc_CP];
    **};
```

```
end CP.impl;
```

Listing 4.38: Notations in ASAM

In listing 4.38, we show the controlled process (*CP*) specification and implementation. In the specification, we show the manipulated variables of the controlled process (*Vma_CP*) as an in data port and show the controlled variable of the controlled process (*Vc_CP*) as an out data port. In the implementation, we have identified the type of the component (*type => controlled_process*). The errors of the controller (*C*) and actuator (*A*), (*EO_C*) and (*EO_A*), will propagate through the manipulated variable of the controlled process (*Vma_CP*) to the controlled variable of the controlled process (*Vc_CP*).

```
system Top_System
end Top_System;

system implementation Top_System.impl
    subcomponents
        S: system S.impl;
        C: system C.impl;
        A: system A.impl;
        CP: system CP.impl;
    connections
        conn_1: port S.Vme_S -> C.Vme_C;
        conn_2: port C.CA_C -> A.CA_A;
        conn_3: port A.Vma_A -> CP.Vma_CP;
        conn_4: port CP.Vc_CP -> S.Vc_S;
end Top_System.impl;
```

Listing 4.39: Notations in ASAM

In listing 4.39, we show the top level specification and implementation. In the specification, we do not have any features to describe. In the implementation, we show the subcomponents and their connections. In the subcomponents, we have called the component implementations such as sensor implementation (*S.impl*), controller implementation (*C.impl*), actuator implementation (*A.impl*) and controlled process implementation (*CP.impl*). In their connections, we have connected the components together through the ports. For example, the measured variable of the sensor (*S.Vme_S*) is connected to the measured variable of the controller (*C.Vme_C*) under the connection name (*conn_1*). Also, the control action of the controller (*C.CA_C*) is connected to the control action of the actuator (*A.CA_A*) under the connection name (*conn_2*). Then, the manipulated variables of the actuator (*A.Vma_A*) are connected to the manipulated variables of the controlled process (*CP.Vma_CP*) under the connection name (*conn_3*). Finally, the controlled variable of the controlled process (*CP.Vc_CP*) is connected to the controlled variable of the sensor (*S.Vc_S*) under the name the connection (*conn_4*).

4.4.2 Summary

In this evaluation, we have made two different types of analysis, theoretical and practical. In the theoretical analysis, described in section 3.3.1, we have applied our notations and expressions for the train automated door control system to identify more safety constraints for the hazardous possibilities. In the practical analysis, described in section 4.4.1, we have an architectural description for the stakeholders or safety analysts to know how to use our notations in the feedback control loop architecture to run ASAM successfully and generate a report for the architecture model.

The purpose of this evaluation is to demonstrate that ASAM uses system mathematical notation, unlike existing safety analysis methods, to describe the systems error flows allowing it more rigorously audit the model.

In the theoretical analysis, the result of the notation and expression tell us that the safety analysts are able to identify unsafe behavior of the components, specify input/output/function for each component, identify unsafe functional behavior/find more hazardous possibilities using error ontology, identify mathematical notation for each error flow / scenario and provide the safety constraint to mitigate the effects of the error flow.

In the practical analysis, the result tells us that the ASAM's implementation satisfied the notations that have been used in the theoretical analysis.

Chapter 5

Discussion

In this section, we will first discuss the limitations of the techniques used to evaluate the framework presented as part of this research. We will also justify the choices made during the evaluation, and we will discuss the contributions of ASAM.

5.1 Difference Between Two-Way and Three-Way Communication Formats

In this section, we will describe the difference between two-way and three-way communication formats. The two-way communication format is the interaction between two components only. The stakeholders know the source of the hazard and impact destination. But, the three-way communication format is the interaction among the three components. The safety analysts need to focus on the system behavior when the data is sent by the first component, crosses into the second component and is received by the third component. This allows us to identify different kinds of hazard sources and different types of impact destinations. We show the difference between the two formats in figure 5.1.

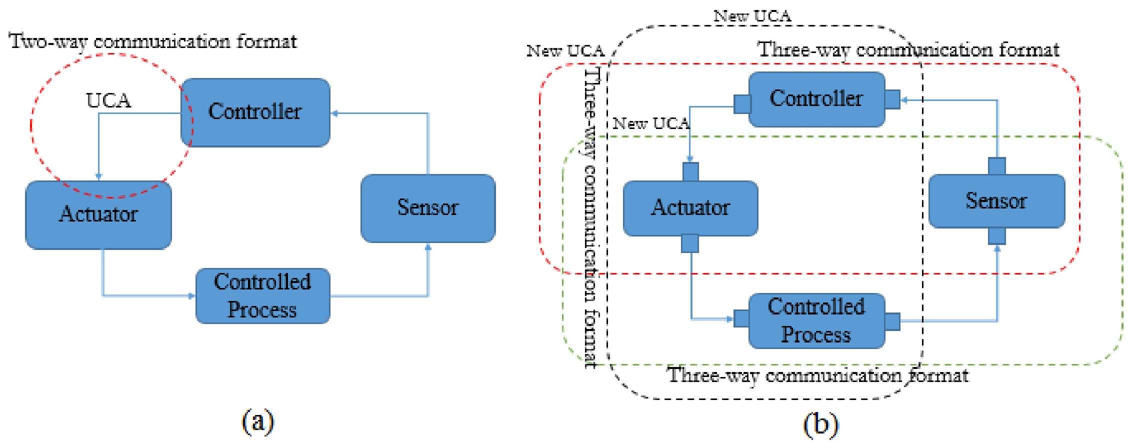

Two-way communication format
UCA
Controller
Actuator
Sensor
Controlled Process
(a)
New UCA
New UCA
Three-way communication format
Three-way communication format
New UCA
Controller
Actuator
Sensor
Controlled Process
Three-way communication format
(b)

Figure 5.1: Two-way versus Three-way communication format

Figure 5.1(a), the two-way communication format, depends on the interaction between controller and actuator to identify unsafe control actions (UCA). This allows the stakeholders / safety analysts to focus on the system controller as a source of the hazard because they are continuously monitoring the system behavior when the controller sends inadequate commands to the actuator. Then, they will provide the safety requirements to mitigate that hazard during the operational system context.

Figure 5.1(b), the three-way communication format, focuses on the interaction among multiple components, usually three in a sequence such as $[sensor \rightarrow controller \rightarrow actuator]$, $[controller \rightarrow actuator \rightarrow controlled\_process]$, and $[actuator \rightarrow controlled\_process \rightarrow sensor]$. In each sequence, we have identified the new unsafe control action for the system. Each element in the sequence has a different role of (source, path, sink) in the context of the error propagation information. This allows the stakeholders to identify different types of hazard sources because each of three major component can have different types of hazards.

5.2 Justification of & Limitations of Evaluation

In the first evaluation (for claim C1), we chose to use only the three-way communication format rather than do a comparison against the two-way format. STPA is a long process that requires extensive description, and STPA analyses of each of our examples are already available in existing literature so repeating the analysis for each example was deemed unnecessary. For those wishing to see one of our examples analyzed in STPA, please see appendix B.

In the second evaluation (for claim C2), we chose error flows to identify safety constraints instead of using control tables, used in two-way communication format analysis. The rational behind this choice was two fold. First, the error flows allow the stakeholders to identify the safety constraints and mitigate the effects of errors that have been propagated from a component to another. Second, the error flows also allow the identification of general and specific causes of the unsafe control actions. In figure 5.2, we show the difference between safety analysis using control tables and safety analysis using error flows:

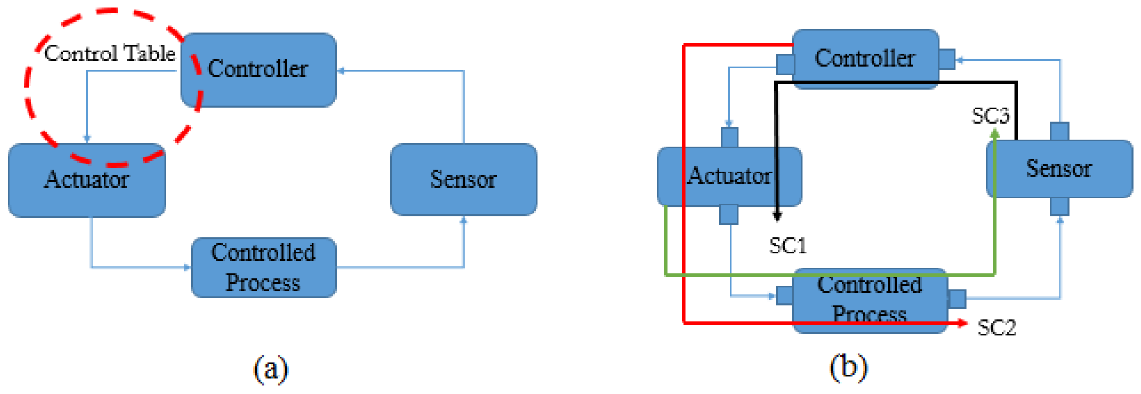


Figure 5.2: Control Table versus Error Flows

As shown in the figure 5.2(a), the control table has been created to hold the safety requirements to eliminate system hazards or unsafe interaction between the controller and actuator. This allows the safety analysts to provide the safety requirements between the two components to mitigate the unsafe system behavior. Also, as shown in the figure 5.2(b), we show that error flow analysis allows us to specify safety constraints and describe hazards between components other than the actuator and controller. The error flows also allow us to specify the exact source, propagation path and sink of each error, allowing us to specify both the general and specific causes of each hazard.

In the third evaluation (for claim C3), we extended safety analysis by adding verification. Most current standards do verify the safety constraints resulting from the analysis instead relying on the experience of the analyst to ensure the correctness of the constraint. Due to our use of formal notation and exact specification, it is possible to perform verification of the constraint against the architecture model of the system. The rational behind this was to introduce the safety constraint verification to ensure the hazard was completely mitigated.

5.3 ASAM Field Contributions

ASAM makes three primary contributions or advancements to the field of safety-critical systems and error propagation information. Briefly, we summarized ASAM's contributions:

First, ASAM uses an architectural model rather than a mental process model. This provides several benefits:

1. It allows us to use defined error ontologies to describe hazards.
2. It allows us to define an unsafe control action's propagation path rather than just describing its effect.
3. It allows us to define more precise safety constraints focusing on the general and / or specific cause of an error identified through the propagation path.

Second, ASAM is based a formal expression notation in addition to the architectural model. This provides several benefits:

1. Formalizing the error ontology allows us to find more possibilities of incorrect system behavior.
2. It allows us to formally specify the behavior of each component, and through rigorous analysis uncover the unsafe functional behavior of each component in the feedback control loop.
3. It allows us to discover additional safety constraints as well as improve existing constraints to more thoroughly mitigate potential hazards.

Finally, ASAM allows verification of its safety constraints in the architectural model through the use of its formal notation. This provides several benefits:

1. It allows us to better ensure that safety constraints completely cover all hazardous possibilities.
2. It allows us to discover weak constraints so that they can be improved.
3. It allows us to discover areas of the architectural model that have missing constraints when verification is done not just against the safety constraints but also against the system requirements, a possibility that exists due to our use of a rigorous architecture specification format and formal notation.

Figure 5.3 shows the contributions in the road map of the ASAM's implementation:

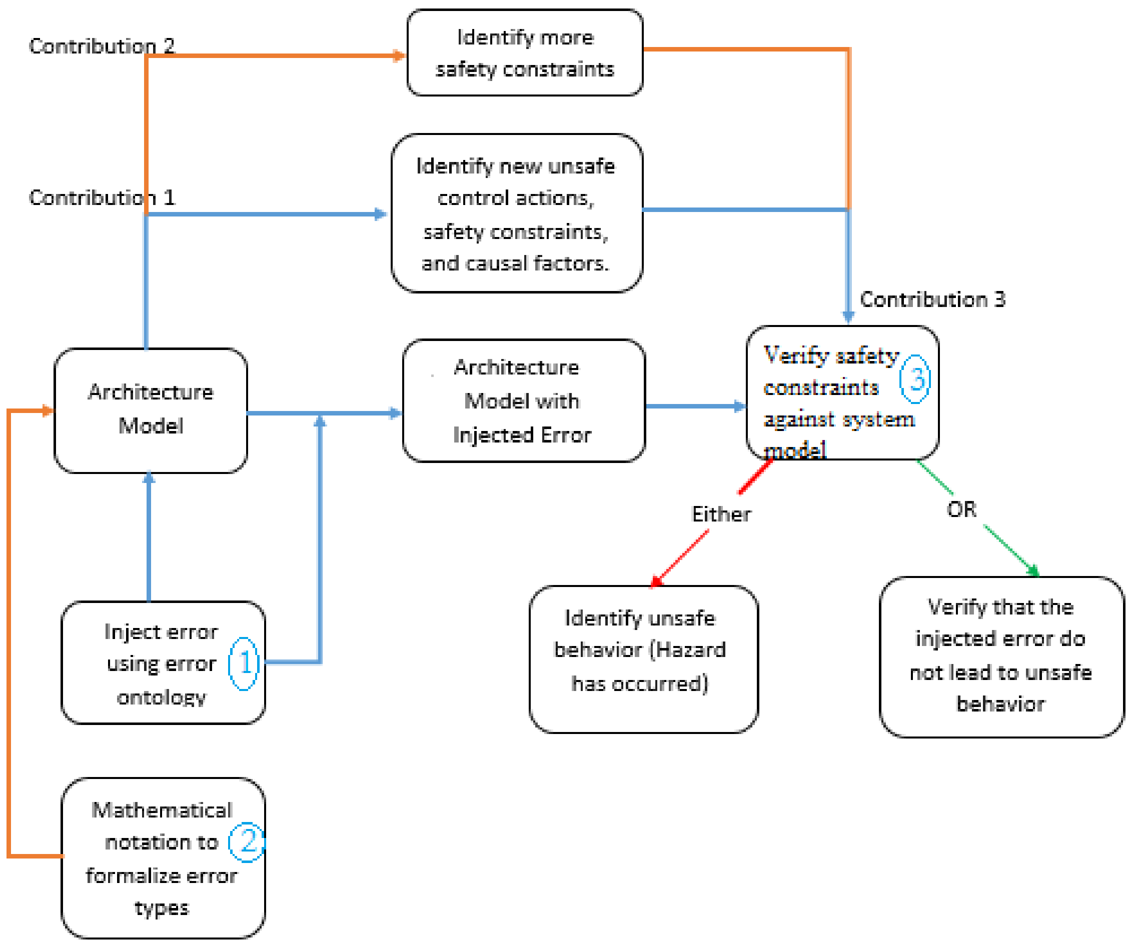

Contribution 2
Identify more safety constraints
Contribution 1
Identify new unsafe control actions, safety constraints, and causal factors.
Contribution 3
Architecture Model
Architecture Model with Injected Error
Verify safety constraints against system model
3
Either
OR
Inject error using error ontology
1
Identify unsafe behavior (Hazard has occurred)
Verify that the injected error do not lead to unsafe behavior
Mathematical notation to formalize error types
2

Figure 5.3: ASAM's contributions

Chapter 6

Related Work

In this section, we will discuss works related to ASAM. We will also discuss work that is loosely related to our method, including research that is likely to be incorporated into our method at a later date.

6.1 Works Related to ASAM

STPA is the current accepted standard used to analyze safety of systems. It is the latest top-down hazard analysis method. It has been used successfully in many safety-critical systems outside of AADL. Some examples of this are [22] and [45]. Our work differs from these groups in where safety analysis is applied during the development process of safety-critical systems. We augment STPA with error propagation information to finding hazards and additional information in the system that other groups miss such as unsafe control actions, safety constraints and specific causes of the unsafe actions. We apply safety analysis to the system architecture during the design phase of the software development life cycle while these other projects or groups apply hazard analysis during the requirements phase.

Our work is most similar to the work performed by the following groups, particularly [37] and [35] both of which use AADL. Additional architecture-based techniques exist, such as [10], [36], [34] and [12]. Our work differs from these groups in that we are explicitly focused on allowing the error ontology assets to be reused in the same manner as the safety-critical assets, and we also add and exploit new several statements in the AADL language such as error statements, error propagation statements, internal failure statements, safety constraint statements that either handle or prevent hazards and verify safety constraints statements.

ASAM is a model-based software safety analysis tool which works with AADL models annotated with an annex (i.e. annex asam) and supported by OSATE. It is used to analyze and generate a report based on information attached to each component in the feedback control loop architecture. There are many other safety and hazard analysis techniques, for example, STPA [22], SAFE [36], FTA [3], HAZOP [30], FMEA [40], FPTN [13], Hip-HOPS [48], FPTC [47]. We differ from each of these primarily in focus.

Our work is focused on mitigating the effects of the errors through safety constraints and verifying the constraints within the system model which are unique to safety-critical systems. Additionally, ASAM is a modified version of the previously existing STPA that has been made more rigorous, and ASAM effectively analyzes hardware & software components in safety-critical systems. Also, ASAM is different than SAFE in handling incoming errors / preventing outgoing errors via the implementation of the safety constraints, and verifies the safety constraint statements to ensure that the error effect or unsafe action has been mitigated.

Our work is different from FTA, HAZOP and FMEA in where faults are inserted into the architecture model. Our generated report also provides how the fault produces an error in the system & how the error propagates through the model & gives the probability value of the occurrence for each error in the model & records the hazard's severity level for each error.

Also, our work is different than FPTN, Hip-HOPS and FPTC in that we show how the error propagation effects are mitigated via the implementation of safety constraints instead of just showing how errors propagate through the model. Another difference is that we focus on introducing compositional reasoning in ASAM, unlike other safety analysis methods, to ensure that the hazardous condition of the propagated error will not happen.

6.2 Additional Related Works

According to our experience in applying hazard analysis methods on different types of systems, these hazard analysis techniques cannot describe the dynamic error behavior of the system, system states, the transitions of the system, and error propagations among system components. Because of this, the traditional hazard analysis techniques depend on decomposition of the system with respect to the hierarchy of failure effects instead of the systems architectural model. Based on this fact, we have decided to develop procedures to augment the existing hazard analysis techniques (such as STPA) with error propagation information and state machines to support modeling and analyzing dynamic error behavior. To prove that the current hazard analysis methods like STPA still needs development, we have several references:

This work [1] shows that the STPA requires suitable diagrammatic notations to represent the relationship among hazards, process model variables in the controller and control actions to the actuator. Our work is different from this group in that we use formal notations to augment STPA to find hazardous conditions in a design. Also, we investigated how a formalized error ontology could assist in identifying unsafe behavior. The formal specification of the error ontology can aid in identifying mathematical expressions for each error flow in the canonical feedback control loop architecture.

This work [2] describes software safety verification based on recent hazard analysis methods. They found the safety requirements using STPA. Then, they changed the requirements to formal specifications using theorem proving such as CTL/LTL (Computation Tree Logic / Linear-Time Temporal Logic). Then, they used a model checker such as SMV (Symbolic Model Verifier) to verify the safety requirements. This work has been done by using several tools, but our work is different than this group, we make it easier by putting the safety requirements within the architecture model and then verify the requirements within in the model as well, using less tools.

This work [4] represents a formal framework for the hazard identification step in STPA. Our work is different from this project, in that we provide guidance, [39], concerning mathematical notations to formalize an error ontology used in the architecture descriptions of systems represented in AADL to improve the rigor of STPA. The results of our work have shown that providing a formal notation for the feedback control loop and providing formal specification for the error ontology lead to finding hazards in the operational system context that other methods miss. By augmenting STPA with an error ontology described in a formal notation, we are able to find more hazards.

Chapter 7

Future Work

As a reminder, ASAM has been created based on three primary contributions that have been completed and evaluated as part of the work for this dissertation, which are:

- Augmenting a hazard analysis method with error propagation information for safety-critical systems [38]
- Using formal notations to augment a hazard analysis method [39]
- Verifying safety constraints in a compositional safety analysis method (i.e. ASAM) [39]

As mentioned in section 2.6, ASAM's future work is the implementation of False Positive (FP) / False Negative (FN) identification for each safety-critical application that we described in the dissertation. Shown below are partially complete FP/FN identification in some of the safety-critical example systems used in this dissertation. Now, we will show that how false positive and false negative could occur and be identified.

7.1 ASAM's Future Work in Adaptive Cruise Control System

As a reminder, a false positive (FP) means the result is positive when in reality it is not. For example, when a warning or automatic braking happens before the critical distance. Also, a false negative (FN) means the result is negative when in reality it is present. For example, if a warning or an automatic braking happens too late.

Figure 7.1 shows ASAM's future work in the adaptive cruise control system to warn a driver if the minimum distance between your own vehicle and the vehicle's front is passed. The obstacle is that the

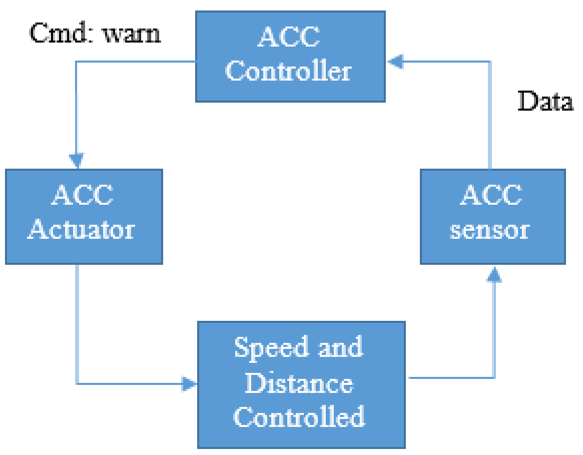


Figure 7.1: ASAM's future work in ACC example

ACC system warns the driver to the apply the brakes when the minimum distance is not presented to the system. Another obstacle is that the ACC sensor presents the minimum distance, but the ACC controller does not send the warn command to the driver. For that purpose, we need to analyze the ACC sensor data availability according to warning / not warning the driver to identify false positives (FP), false negatives (FN), true positives (TP), and true negatives (TN) in ACC system. The analysis is shown in a table below 7.1:

Table 7.1: Identify FP/FN/TP/TN in ACC system

	Min. Distance presented	**Min. Distance not presented**
Positive Result (Warn)	ACC_TP*(1)	ACC_FP*(2)
Negative Result (Not Warn)	ACC_FN*(3)	ACC_TN*(4)

As shown in table 7.1, we have identified a true positive which is ACC_TP*(1): *ACC controller warns the driver when the minimum distance is presented*. Also, we identified a false positive which is ACC_FP*(2): *ACC controller warns the driver when the minimum distance is not presented.*. It means ACC_controller made a wrong decision because the warning command has been sent based on incorrect data. Also, we identified a false negative which is ACC_FN*(3): *ACC controller does not warn the driver when the minimum distance is presented.*. It means ACC_controller does not make a decision when data is available. That's a hazard because the data is available but the controller missed sending the warning command, potentially leading to a crash. Finally, we have identified a true negative which is ACC_TN*(4): *ACC controller does not warn the driver when the minimum distance is not presented*. It means the ACC_controller does not make a decision when the data is not available, normal operation for the system.

7.2 ASAM's Future Work in Medical Embedded Device - Pacemaker

In this example, we will identify FP/FN/TP/TN for the pacemaker. The device_controller_monitor (DCM) sends the pace command to the pulse generator (PG) to pace the heart with respect to sensing data.

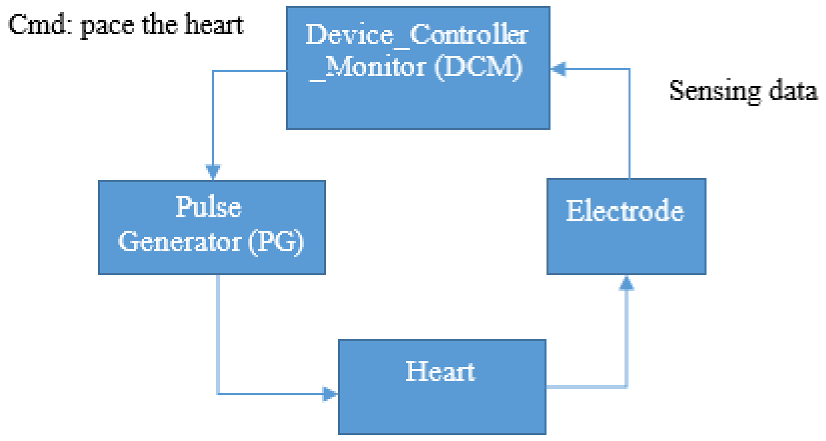


Figure 7.2: ASAM's future work in pacemaker example

Figure 7.2 shows ASAM's future work in the pacemaker system to pace the heart with respect to availability of sensing data which provides by the electrode sensor. The problem is that the DCM paces the heart when the sensing data is not available. Another problem is that the DCM does not pace the heart when the sensing data is available. For that purpose, we need to analyze the electrode's sensing data according to pacing / not pacing the heart to identify false positives (FP), false negatives (FN), true positives (TP) and true negatives (TN) in the pacemaker system. We will show that in the table 7.2:

Table 7.2: Identify FP/FN/TP/TN in pacemaker

	Sensing Data Available	**Sensing Data Not Available**
Positive Result (Pace)	PM_TP*(1)	PM_FP*(2)
Negative Result (Not Pace)	PM_FN*(3)	PM_TN*(4)

As shown in table 7.2, we have identified a true positive which is PM_TP*(1): *DCM paces the heart when the sensing data is available*. In addition, we identified a false positive PM_FP*(2): *DCM paces the heart when the sensing data is not available*. It means DCM made a wrong decision because the pace command has been sent based on incorrect sensing data. Also, we have identified a false negative which is PM_FN*(3): *DCM does not pace the heart when the sensing data is available*. That is a hazard because the sensing data is available but the controller does not send the command to the pulse generator, potentially causing a life-threatening situation. Finally, we have identified a true negative which is PM_TN*(4): *DCM does not pace the heart when the sensing data is not available*. It means that DCM does not make a decision when the data is not available, normal operation for the system.

7.3 Summary

In this section, we have focused on how to identify the false positives (FP) and false negatives (FN) in each safety-critical application example that described in the dissertation. Specifically, we focused on availability of the data which is provided by the sensor and then analyzed by the controller to make a decision. This step can be considered as an initial identification step to start the future work of ASAM to identify FP/FN.

According to the identification step, false negatives (FN) can be assessed as a hazard for the safety-critical systems. As part of our future work we will analyze the following questions:

- RQ6) How false positives and false negatives could occur in the safety-critical systems?
- RQ7) What can be done to reduce false positive and false negative probabilities in the feedback control loop architecture?
- RQ8) How to handle false positives and false negatives in the architecture to identify some margins of system safety?

Chapter 8

Conclusion

In the proposal for this dissertation, we promised to provide a new architecture safety analysis method (ASAM) to analyze system safety in a novel way. In this dissertation, we have presented the outline of a new method as well as an implementation of of that method for the analysis of safety-critical systems. Our method has been built based on STPA but it has the following additions:

- ASAM is capable of finding more hazards by using a three-way communication format as opposed to the two-way format used by STPA. This allows the identification of potential internal failures of the major components in the feedback control loop architecture. The result of each internal failure is an error. The error propagates and cuts across components based on the three-way interaction format. This allows stakeholders to identify new unsafe control actions and allows the reduction of the potential effects of residual hazards in the operational system context.

- ASAM also provides a mathematical notation/expression and formal specification of error ontology for the feedback control loop architecture to find more safety constraints during hazard analysis. This allows stakeholders to use formal methods to find more hazardous possibilities and mitigate them using safety constraints.

- ASAM provides safety verification procedures to verify the safety constraints to ensure that hazardous conditions cannot occur. ASAM verifies the safety constraints against the system model with injected errors. This allows us to determine that either unsafe behaviors occur (which means the error leads to a hazardous condition in the system) or verifies that the error does not lead to unsafe behaviors of the system.

In conclusion, ASAM allows system stakeholders / safety analysts to find new hazards and new safety constraints that STPA does not find. This does not mean that our method is a replacement for STPA; however, it supports more effective and rigorous analyses of the elements of safety-critical systems. Our method is an augmentation of existing safety analysis methods, including STPA, to be used alongside current industry standards to uncover more difficult latent errors that current methods are not capable of discovering.

Appendices

Appendix A Error Ontology

- Service Errors
  - ItemOmission / ItemCommission
  - ServiceOmission / ServiceCommission
  - SequenceOmission / SequenceCommission
    - LateServiceStart
    - EarlyServiceTermination
    - BoundedOmissionInterval
    - EarlyServiceStart
    - LateServiceTermination
- Timing Errors
  - ItemTimingErro
    - EarlyDelivery
    - LateDelivery
  - SequenceTimingError
    - HighRate
    - LowRate
    - RateJitter
  - ServiceTimingError
    - DelayedService
    - EarlyService
- Value Errors
  - ItemValueError
    - UndetectableValueError
    - DetectableValueError
      - OutOfRange

      - BelowRange
      - AboveRange
      - OutOfBounds
  - SequenceValueError
    - BoundedValueChange
    - StuckValue
    - OutOfOrder
  - ServiceValueError
    - OutOfCalibration
- Replication Errors
- Concurrency Errors
- Access Control Errors

Appendix B STPA Example

In [45], an automated door control system for a train is described. We want to know how the train door can harm people. We simplified the example according to the steps:

1. The stakeholder establishes fundamental analyses to identify accidents and the hazards associated with those accidents. Here, we have: **Accident**: Hit by closing door, falling out of the train door, people trapped inside the train during emergency evacuation. **Hazards:** The hazards related to the accident could be: H-1: Door closes on a person in the doorway. H-2: Door opens when the train is not in the station / it is not aligned with a station platform. H-3: People are not able to exit during emergency.

2. The stakeholder designs a feedback control loop for the system to identify major components such as sensors, controllers, actuators, and the controlled process. For the train door, we have the following structure as shown in figure 1

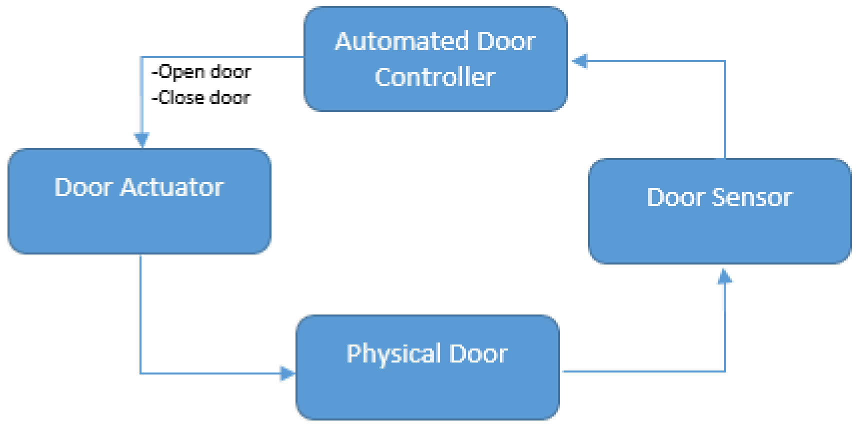
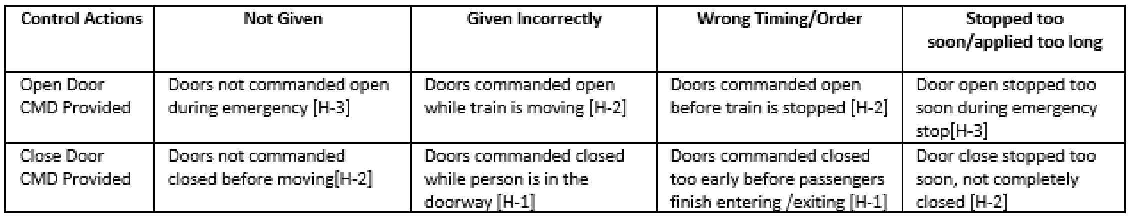


Figure 1: Feedback control loop for automated door control system for train

3. The stakeholder identifies unsafe control actions that could lead to hazardous states. The figure 2 shows the relationship among hazards that we have found previously and the provided control actions.

4. The stakeholder identifies causal factors for the unsafe control actions. For example, the doors are commanded closed while a person in the doorway. One of the potential causes of that action is an incorrect process model (i.e, the controller incorrectly believes that the doorway is clear). This is the result of inadequate / ineffective feedback which is provided by a failing sensor.

   The figure 3 will help to identify causal factors of the unsafe control actions. For that purpose, we provide table 4 to clarify the context of each control action. The name of the columns are the variables



Control Actions	Not Given	Given Incorrectly	Wrong Timing/Order	Stopped too soon/applied too long
Open Door CMD Provided	Doors not commanded open during emergency [H-3]	Doors commanded open while train is moving [H-2]	Doors commanded open before train is stopped [H-2]	Door open stopped too soon during emergency stop[H-3]
Close Door CMD Provided	Doors not commanded closed before moving[H-2]	Doors commanded closed while person is in the doorway [H-1]	Doors commanded closed too early before passengers finish entering /exiting [H-1]	Door close stopped too soon, not completely closed [H-2]

Figure 2: Unsafe control actions for train door controller

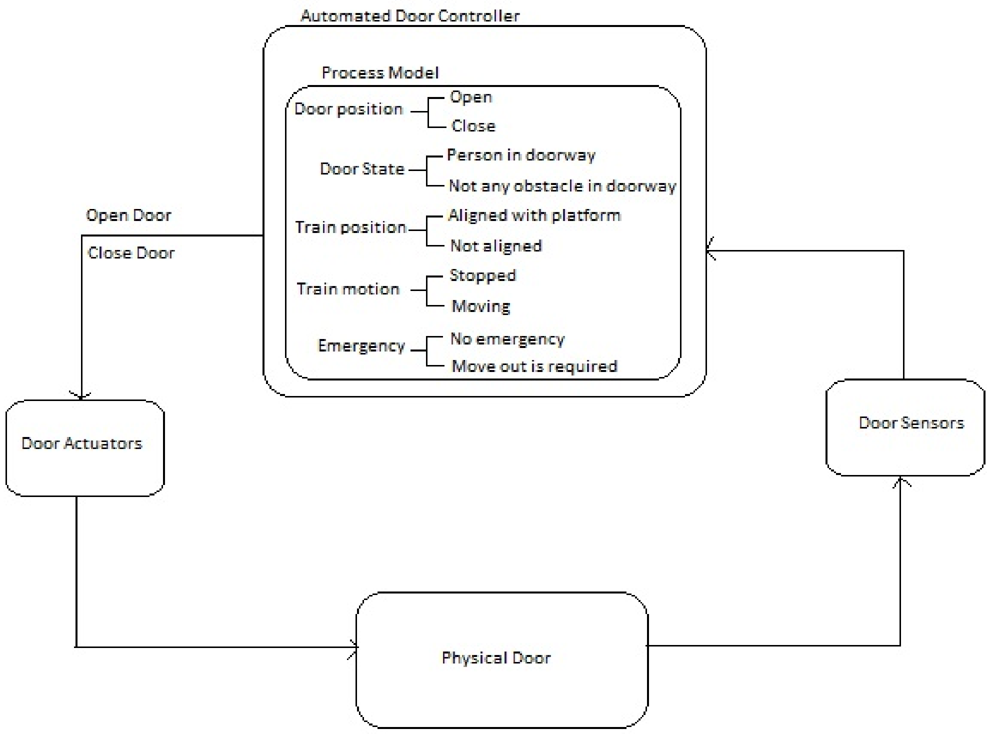
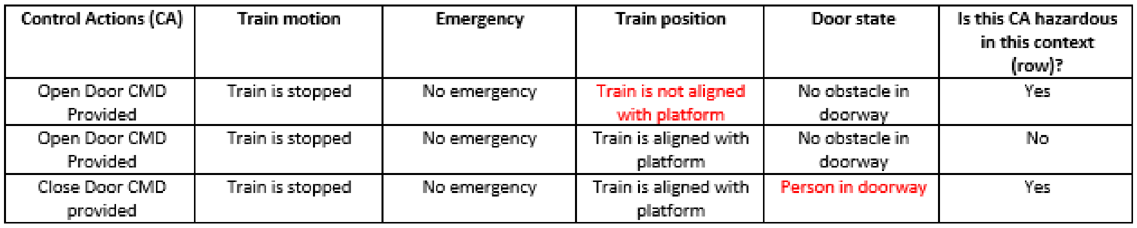


Figure 3: Process model of the controller

of the process model and the cells are the values of the variables. We know that the controller makes a decision based on the context of the control action. For example, the open door command consists of the values: the train is stopped, no emergency, train is not aligned with platform, and no obstacle in doorway.



Control Actions (CA)	Train motion	Emergency	Train position	Door state	Is this CA hazardous in this context (row)?
Open Door CMD Provided	Train is stopped	No emergency	Train is not aligned with platform	No obstacle in doorway	Yes
Open Door CMD Provided	Train is stopped	No emergency	Train is aligned with platform	No obstacle in doorway	No
Close Door CMD provided	Train is stopped	No emergency	Train is aligned with platform	Person in doorway	Yes

Figure 4: Context of the control actions

Appendix C Safety-Critical System Terminologies

- **Fault** - 1) A manifestation of an error in software 2) An incorrect step/ process /data definition in a computer program 3)A defect in a component / hardware device [17].
- **Error** - 1) The difference between a measured value and the specified value. 2)Omission of a requirement/constraints in the design specification. 3)A human action which produces an incorrect result such as software containing a fault [17].
- **Failure** - 1) Termination of the capability of a product to do a required function 2) Can be described as an event in a system / component which does not perform a required function within specified limits [17].
- **Accident** - Can be described as a (loss) which results from inadequate enforcement of the behavioural safety requirements on the process [21].
- **Hazard** - 1) A source of potential harm which leads to human injury, damage to health, property/the environment 2) An essential property or condition which has potential to cause harm. 3) System state / set of conditions that will lead to an accident [17, 21].
- **Unsafe Control Actions** - They are hazardous scenarios that may occur in the system due to provided/not provided control action when the system need it [21].
- **Safety Constraints** - They are the safeguards which prevent the system from leading to accident/loss [21].
- **Process model** - A model is required to determine system variables states and their values that affect the controller decision and it can be updated through feedback [44].
- **Process model variables** - They are the safety-critical system variables of the controller in the feedback control loop. They have an effect on the safety of issuing the control actions [44].
- **Causal Factors** - They are the accident scenarios which explain how unsafe control actions may occur and how safe control actions may not occur [44].
- **Software Safety** - Can be described as a field of study of software assurance, it is a systematic method to identifying, analyzing, mitigating, and controlling software hazards and hazardous functions to assure that the system is safe within operational context [27].

- **Internal failure** - 1) Can be described as an error detection condition. Specifically, it is used to identify source of the error because result of internal failure is a propagation [33].

  2) given a correct input, a failure happens during the execution which leads to produce an erroneous output [8].

  3) Can be interpreted as a probability of component failure per demand and accumulated with the failure rate which occurs during interaction with other components [18]

Appendix D List of Abbreviations

- AADL - Architecture Analysis and Design Language
- STPA - System Theoretic Process Analysis
- STAMP - Systems Theoretic Accident Model and Processes
- EMV2 - Error Model Annex Version 2
- AGREE - Assume Guarantee REasoning Environment
- SAE - Society of Automotive Engineers
- ACC - Adaptive-Cruise Control
- UCA - Unsafe Control Action
- MBE - Model-Based Engineering
- FTA - Fault Tree Analysis
- FMEA - Failure Modes and Effects Analysis
- SAFE - Systematic Analysis of Faults and Errors
- HAZOP - Hazard and Operability Study
- FPTN - Failure Propagation and Transformation Notation
- FPTC - Fault Propagation and Transformation Calculus
- Hip-HOPS - Hierarchically Performed Hazard Origin and Propagation Studies
- LTL - Linear-Time Temporal Logic
- CTL - Computation Tree Logic
- SMV - Symbolic Model Verifier
- FP - False Positive
- FN - False Negative
- TN - True Negative
- TP - True Positive

Appendix E   ASAM Annex Language Reference

E.1   Error Statement

E.1.1   Format

```
Errs => [{
      Type => <ERROR_1_TYPE>(<ERROR_1_SEVERITY>, p=<ERROR_1_PROB>),
      UCA => <UCA_1_ID>: "<UCA_1_DESCRIPTION>",
      Causes => {
            General => "<ERROR_1_GENERAL_CAUSE_DESCRIPTION>",
            Specific => "<ERROR_1_SPECIFIC_CAUSE_DESCRIPTION>"
      },
      SC => <SC_1_ID>: "<SC_1_DESCRIPTION>" verified by [GAU1]
}, {
      Type => <ERROR_2_TYPE>(<ERROR_2_SEVERITY>,p=<ERROR_2_PROB>),
      UCA => <UCA_2_ID>: "<UCA_2_DESCRIPTION>",
      Causes => {
            General => "<ERROR_2_GENERAL_CAUSE_DESCRIPTION>",
            Specific => "<ERROR_2_SPECIFIC_CAUSE_DESCRIPTION>"
      },
      SC => <SC_2_ID>: "<SC_2_DESCRIPTION>" verified by [GAU2]
},...,{
      Type => <ERROR_N_TYPE>(<ERROR_N_SEVERITY>,p=<ERROR_N_PROB>),
      UCA => <UCA_N_ID>: "<UCA_N_DESCRIPTION>",
      Causes => {
            General => "<ERROR_N_GENERAL_CAUSE_DESCRIPTION>",
            Specific => "<ERROR_N_SPECIFIC_CAUSE_DESCRIPTION>"
      },
      SC => <SC_N_ID>: "<SC_N_DESCRIPTION>" verified by [GAU3]
}];
```

E.1.2   Description

The errors statement allows known errors that determined causes and known safety constraints to be attached to a component of the architecture. A report of all components and their mitigated errors can then be produced. ERROR_SEVERITY can be one of 4 values: Catastrophic, Critical, Marginal, or Negligible. The ERROR_PROB represents how likely the error is to occur. This is a value between 0 and 1 (inclusive). The verified by portion requires the XAGREE annex to be installed. Once installed, you can a mathematical proof of your safety constraint associating XAGREE guarantees to each constraint. Provided that those guarantees can be proven, the safety constraint will be reported as verified.

E.1.3 Example

```
annex asam {**
    Errs => [{
        Type => ValueOutOfRangeError(Catastrophic,p=0.001),
        UCA => UCA1: "That error gives the unsafe control action to the system",
        Causes => {
            General => "Why this unsafe action happened?"
            Specific => "That error propagated and crossed three components"
        },
        SC => SC1: "Provide the safety constraint to mitigate the effects of
            the error" verified by [GAU1]
    }]
};
```

E.2 Error Propagation Statement

E.2.1 Format

```
[<INCOMING_ERROR_1>, <INCOMING_ERROR_2>, ... <INCOMING_ERROR_N>] on
    [<INCOMING_PORT_1>, <INCOMING_PORT_2>, ... <INCOMING_PORT_N>] causes
    [<OUTGOING_ERROR_1>(<ERROR_1_SEVERITY>, p=<ERROR_1_PROB>),
    <OUTGOING_ERROR_2>(<ERROR_2_SEVERITY>,
    p=<ERROR_2_PROB>),... <OUTGOING_ERROR_N>(<ERROR_N_SEVERITY>,
    p=<ERROR_N_PROB>)] on [<OUTGOING_PORT_1>, <OUTGOING_PORT_2>,
    ...<OUTGOING_PORT_N>];
```

E.2.2 Description

Error propagation statements allow incoming errors to be transformed within the current component to outgoing errors. Errors that do not have a transformation rule are assumed to not propagate beyond the current component. ERROR_SEVERITY can be one of 4 values: Catastrophic, Critical, Marginal, or Negligible. The ERROR_PROB represents how likely the error is to occur. This is a value between 0 and 1 (inclusive).

E.2.3 Example

```
annex asam {**
    [ValueOutOfRange] on [door_open_c, door_blocked_c] causes
    [ValueOutOfRange(Marginal,p=0.5)] on [open_door_c, close_door_c];
};
```

E.3 Internal Failure Statement

E.3.1 Format

```
internal failure <ID>: "<DESCRIPTION>" causes [<ERROR_1>(<ERROR_1_SEVERITY>,
    p=<ERROR_1_PROB>), <ERROR_2>(<ERROR_2_SEVERITY>, p=<ERROR_2_PROB>), ...
    <ERROR_N>(<ERROR_N_SEVERITY>, p=<ERROR_N_PROB>)] on [<OUT_PORT_1>,
    <OUT_PORT_2>, ... <OUT_PORT_N>];
```

E.3.2 Description

The internal failure statement allows internal failures to be documented as well as what errors they cause and on which ports they are caused. ERROR_SEVERITY can be one of 4 values: Catastrophic, Critical, Marginal, or Negligible. The ERROR_PROB represents how likely the error is to occur. This is a value between 0 and 1 (inclusive).

E.3.3 Example

```
annex asam {**
    internal failure INTF2: "something bad happened" causes
        [ValueOutOfRange(Negligible,p=0.002)] on [open_door_c,close_door_c];
};
```

E.4 Safety Constraint Handles Statement

E.4.1 Format

```
safety constraint <ID>: "<DESCRIPTION>" handles [<ERROR_1>, <ERROR_2>, ...
    <ERROR_N>] on [<IN_PORT_1>, <IN_PORT_2>, ... <IN_PORT_N>] verified by
    [GAU1, GAU2, .... GAU_N];
```

E.4.2 Description

The safety constraint handles statements allows you to specify safety constraints for handling errors on in ports. The verified by portion requires the XAGREE annex to be installed. Once installed, you can a mathematical proof of your safety constraint associating XAGREE guarantees to each constraint. Provided that those guarantees can be proven, the safety constraint will be reported as verified.

E.4.3 Example

```
annex asam {**
    safety constraint SC1: "handle an error" handles [ValueOutOfRange] on
        [in_data] verified by [GAU1];
};
```

E.5 Safety Constraint Prevents Statement

E.5.1 Format

```
safety constraint <ID>: "<DESCRIPTION>" prevents [<ERROR_1>, <ERROR_2>, ...
    <ERROR_N>] on [<OUT_PORT_1>, <OUT_PORT_2>, ... <OUT_PORT_N>] verified by
    [GAU1, GAU2, ... GAU_N];
```

E.5.2 Description

The safety constraint handles statements allows you to specify safety constraints for preventing errors on out ports. The verified by portion requires the XAGREE annex to be installed. Once installed, you can a mathematical proof of your safety constraint associating XAGREE guarantees to each constraint. Provided that those guarantees can be proven, the safety constraint will be reported as verified.

E.5.3 Example

```
annex asam {**
    safety constraint SC1: "handle an error" prevents [ValueOutOfRange] on
        [out_data] verified by [GAU1];
};
```

Appendix F XText Grammar

```
grammar edu.clemson.asam.Asam with org.osate.xtext.aadl2.properties.Properties

generate asam "http://www.clemson.edu/ASAM"

import "http://aadl.info/AADL/2.0" as aadl2

AnnexLibrary returns aadl2::AnnexLibrary:
 AsamLibrary;

AnnexSubclause returns aadl2::AnnexSubclause:
 AsamSubclause;

AsamLibrary:
 {AsamContractLibrary} contract=AsamContract;

AsamSubclause:
 {AsamContractSubclause} contract=AsamContract;

AsamContract returns Contract:
 {AsamContract} (statement+=AsamStatement)*;

AsamStatement:
 ErrsStatement | InternalFailureStatement | SafetyConstraintHandlesStatement |
     SafetyConstraintPreventsStatement | ErrorPropagationRuleStatement |
     TypeStatement;

InternalFailureStatement:
 'internal' 'failure' internalFailureId=ID ':'
     internalFailureDescription=STRING 'causes' '[' errors=ErrorsList ']' 'on'
     '[' ports=PortsList ']' ';';

SafetyConstraintHandlesStatement:
 'safety' 'constraint' safetyConstraintId=ID ':'
     safetyConstraintDescription=STRING 'handles' '['
     errors=ErrorsListNoProbability ']' 'on' '[' ports=PortsList ']' 'verified'
     'by' '[' matchingXagreeStatements=GuaranteeIds ']' ';';

SafetyConstraintPreventsStatement:
 'safety' 'constraint' safetyConstraintId=ID ':'
     safetyConstraintDescription=STRING 'prevents' '['
     errors=ErrorsListNoProbability ']' 'on' '[' ports=PortsList ']' 'verified'
     'by' '[' matchingXagreeStatements=GuaranteeIds ']' ';';

ErrorPropagationRuleStatement:
 '[' inErrorsList=ErrorsListNoProbability ']' 'on' '[' inPortsLists=PortsList
     ']' 'causes' '[' outErrorsList=ErrorsList ']' 'on' '['
     outPortsLists=PortsList ']' ';';

TypeStatement:
 'type' '=>' type=('sensor' | 'controller' | 'controlled_process' | 'actuator')
     ';';

ErrorsList:
 firstError=Error (',' restErrors+=Error)*;

ErrorsListNoProbability:
 firstError=ErrorNoProbability (',' restErrors+=ErrorNoProbability)*;

PortsList:
 firstPort=ID (',' restPorts+=ID)*;

ErrsStatement:
 {ErrsStatement} 'Errs' '=>' '[' firstError=ErrorStatement (','
     restErrors+=ErrorStatement)* ']';

```

```
ErrorStatement:
 '{' type=ErrorTypeStatement ','
   uca=UnsafeControlActionStatement ','
   cause=CausesStatement ','
   sc=SafetyConstraintStatement

  '}';

ErrorTypeStatement:
 'Type' '=>' (errorType=Error);

UnsafeControlActionStatement:
 'UCA' '=>' id=ID ':' description=STRING;

CausesStatement:
 'Causes' '=>' '{' (general=GeneralCauseStatement |
     specific=SpecificCauseStatement | general=GeneralCauseStatement ','
     specific=SpecificCauseStatement) '}';

GeneralCauseStatement:
 'General' '=>' description=STRING;

SpecificCauseStatement:
 'Specific' '=>' description=STRING;

SafetyConstraintStatement:
 'SC' '=>' id=ID ':' description=STRING 'verified' 'by' '['
     matchingXagreeStatements=GuaranteeIds ']';

GuaranteeIds:
 first=ID (',' rest+=ID)*;

Error:
```

```
error=('ServiceError' | 'ItemOmission' | 'ServiceOmission' |
    'SequenceOmission' | 'TransientServiceOmission' | 'LateServiceStart' |
    'EarlyServiceTermination' | 'BoundedOmissionInterval' | 'ItemCommission' |
    'ServiceCommission' | 'SequenceCommission' | 'EarlyServiceStart' |
    'LateServiceTermination' | 'TimingRelatedError' | 'ItemTimingError' |
    'EarlyDelivery' | 'LateDelivery' | 'SequenceTimingError' | 'HighRate' |
    'LowRate' | 'RateJitter' | 'ServiceTimingError' | 'DelayedService' |
    'EarlyService' | 'ValueRelatedError' | 'ItemValueError' |
    'UndetectableValueError' | 'DetectableValueError' | 'OutOfRange' |
    'BelowRange' | 'AboveRange' | 'OutOfBounds' | 'SequenceValueError' |
    'BoundedValueChange' | 'StuckValue' | 'OutOfOrder' | 'ServiceValueError' |
    'OutOfCalibration' | 'ReplicationError' | 'AsymmetricReplicatesError' |
    'AsymmetricValue' | 'AsymmetricApproximateValue' | 'AsymmetricExactValue'
    | 'AsymmetricTiming' | 'AsymmetricOmission' | 'AsymmetricItemOmission' |
    'AsymmetricServiceOmission' | 'SymmetricReplicatesError' |
    'SymmetricValue' | 'SymmetricApproximateValue' | 'SymmetricExactValue' |
    'SymmetricTiming' | 'SymmetricOmission' | 'SymmetricItemOmission' |
    'SymmetricServiceOmission' | 'ConcurrencyError' | 'RaceCondition' |
    'ReadWriteRace' | 'WriteWriteRace' | 'MutExError' | 'Deadlock' |
    'Starvation' | ID)
'('
severityLevel=('Catastrophic' | 'Critical' | 'Marginal' | 'Negligible')
',p='
probability=('0' | '1' | FLOAT)
')';

ErrorNoProbability:
error=('ServiceError' | 'ItemOmission' | 'ServiceOmission' |
    'SequenceOmission' | 'TransientServiceOmission' | 'LateServiceStart' |
    'EarlyServiceTermination' | 'BoundedOmissionInterval' | 'ItemCommission' |
    'ServiceCommission' | 'SequenceCommission' | 'EarlyServiceStart' |
    'LateServiceTermination' | 'TimingRelatedError' | 'ItemTimingError' |
    'EarlyDelivery' | 'LateDelivery' | 'SequenceTimingError' | 'HighRate' |
    'LowRate' | 'RateJitter' | 'ServiceTimingError' | 'DelayedService' |
    'EarlyService' | 'ValueRelatedError' | 'ItemValueError' |
    'UndetectableValueError' | 'DetectableValueError' | 'OutOfRange' |
    'BelowRange' | 'AboveRange' | 'OutOfBounds' | 'SequenceValueError' |
    'BoundedValueChange' | 'StuckValue' | 'OutOfOrder' | 'ServiceValueError' |
    'OutOfCalibration' | 'ReplicationError' | 'AsymmetricReplicatesError' |
    'AsymmetricValue' | 'AsymmetricApproximateValue' | 'AsymmetricExactValue'
    | 'AsymmetricTiming' | 'AsymmetricOmission' | 'AsymmetricItemOmission' |
    'AsymmetricServiceOmission' | 'SymmetricReplicatesError' |
    'SymmetricValue' | 'SymmetricApproximateValue' | 'SymmetricExactValue' |
    'SymmetricTiming' | 'SymmetricOmission' | 'SymmetricItemOmission' |
    'SymmetricServiceOmission' | 'ConcurrencyError' | 'RaceCondition' |
    'ReadWriteRace' | 'WriteWriteRace' | 'MutExError' | 'Deadlock' |
    'Starvation' | ID);

terminal FLOAT: '0.' INTEGER_LIT;
```

Appendix G ASAM's plug-in

G.1 Install OSATE

`http://osate.org/download-and-install.html`

G.2 Install ASAM

`https://bitbucket.org/strategicsoftwareengineering/asam-update-site/`

G.3 ASAM Examples

G.3.1 Adaptive Cruise Control System - (ACC)

Example 1: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/ACCExample1.aadl`

Example 2: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/ACCExample2.aadl`

G.3.2 Medical Embedded Device - Pacemaker (PM)

Example 1: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/PMExample1.aadl`

Example 2: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/PMExample2.aadl`

Example 3: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/PMExample3.aadl`

G.3.3 Train Automated Door Control System - (TADCS)

Example 1: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/TADCSExample1.aadl`

Example 2: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/TADCSExample2.aadl`

Example 3: `https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/TADCSExample3.aadl`

G.3.4 Infant Incubator Temperature Control System - Isolette (ISO)

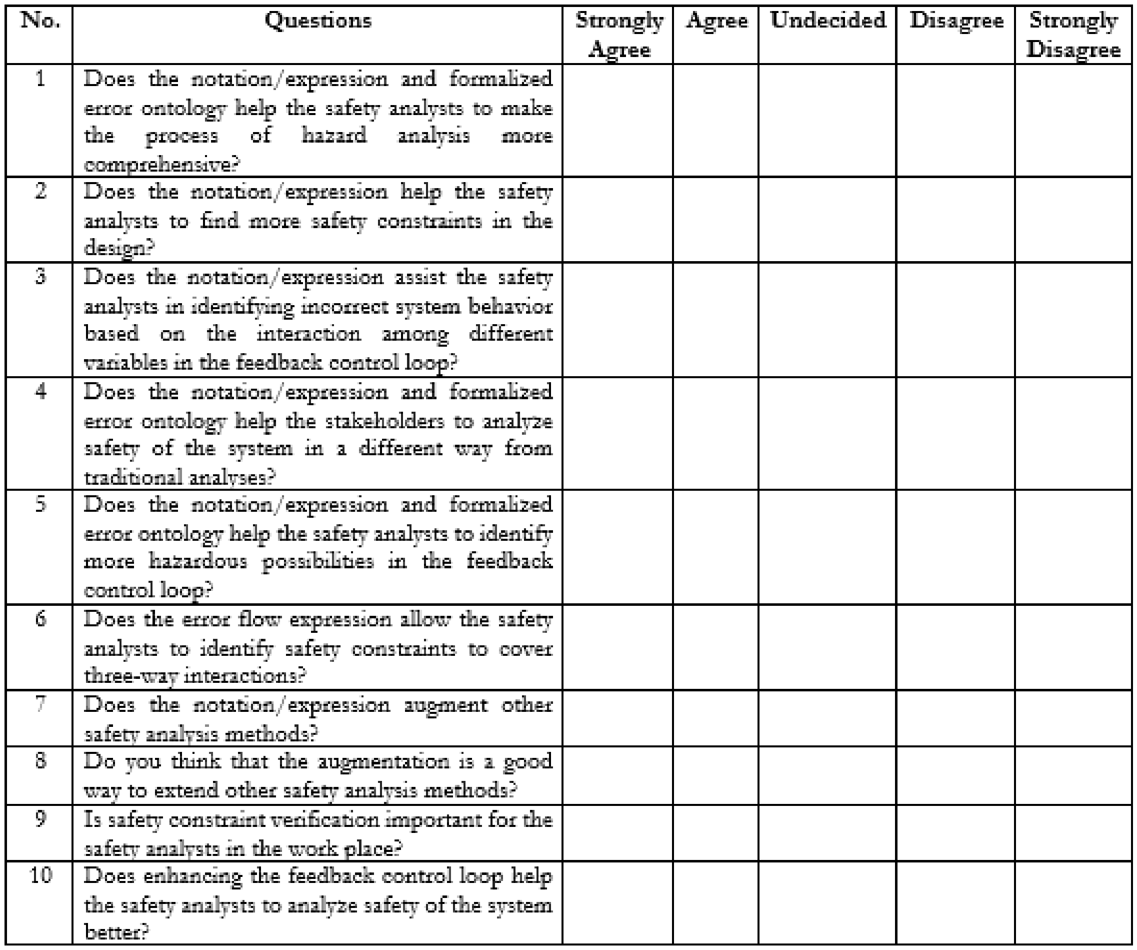
https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/ISOLETTEExample1.aadl

G.3.5 Notation Example

https://bitbucket.org/strategicsoftwareengineering/asam-examples-2/src/master/NotationExample.aadl

Appendix H Feedback Results Representation

In this section, we present the results of the feedback questionnaire from section 3.3 which was published as a technical paper in the 36th international system safety conference (ISSC). In that session, after the presentation, 21 system safety specialists voted for 10 questions based on different criteria such as strongly agree, agree, undecided, disagree and strongly disagree. Questions are listed in figure 5.



No.	Questions	Strongly Agree	Agree	Undecided	Disagree	Strongly Disagree
1	Does the notation/expression and formalized error ontology help the safety analysts to make the process of hazard analysis more comprehensive?					
2	Does the notation/expression help the safety analysts to find more safety constraints in the design?					
3	Does the notation/expression assist the safety analysts in identifying incorrect system behavior based on the interaction among different variables in the feedback control loop?					
4	Does the notation/expression and formalized error ontology help the stakeholders to analyze safety of the system in a different way from traditional analyses?					
5	Does the notation/expression and formalized error ontology help the safety analysts to identify more hazardous possibilities in the feedback control loop?					
6	Does the error flow expression allow the safety analysts to identify safety constraints to cover three-way interactions?					
7	Does the notation/expression augment other safety analysis methods?					
8	Do you think that the augmentation is a good way to extend other safety analysis methods?					
9	Is safety constraint verification important for the safety analysts in the work place?					
10	Does enhancing the feedback control loop help the safety analysts to analyze safety of the system better?					

Figure 5: Feedback questionnaire

Also, figure 6 shows the results of their votes with respect to each question.

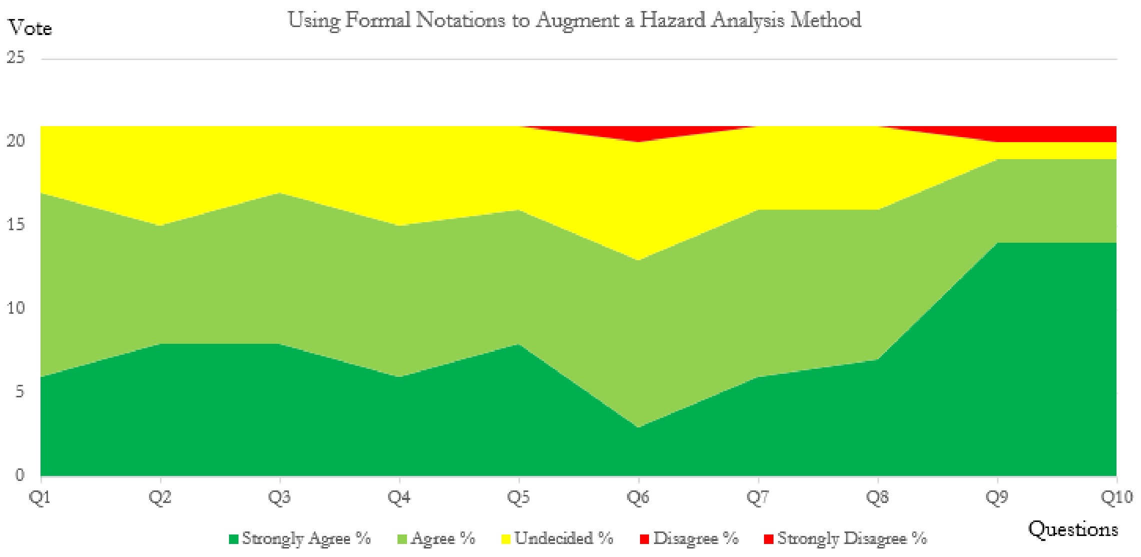


Figure 6: Results of the votes

Bibliography

[1] Asim Abdulkhaleq and Stefan Wanger. Integrating state machine analysis with system-theoretic process analysis. *LNI Proceeding of Software Engineering Workshop Band, Springer*, P(215):501–514, December 2013.

[2] Asim Abdulkhaleq and Stefan Wanger. A software safety verification method based on system-theoritic process analysis. In *International Conference on Computer Safety, Reliability, and Security,(SAFECOMP)*. Springer, 2014.

[3] Risza Ruzli Ahmed Baig and Azizul Buang. Reliability analysis using fault tree analysis: A review. *International Journal of Chemical Engineering and Applications*, 4(3):169–173, June 2013.

[4] Philip Asare, John Lach, and John A. Stankovic. Fstpa-i: A formal approach to hazard identification via system theoretic process analysis. In *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems*, ICCPS '13, pages 150–159, New York, NY, USA, 2013. ACM.

[5] Mordechai Ben-Ari. The bug that destroyed a rocket. *Special Interest Group on Computer Science Education, SIGCSE Bulletin*, 33(2):58–59, June 2004.

[6] Ronal Berger and et al. Rhythm management product performance report: Q2 edition. *Boston Scientific:Advancing Science for Life*, June 2016.

[7] Darren Cofer and et al. Compositional verification of architectural models. In *NFM 2012, LNCS 7226, pp. 126140*. Springer-Verlag Berlin Heidelberg, 2012.

[8] V. Cortellessa and V. Grassi. Role and impact of error propagation in software architecture reliability. Technical Report TRCS 007/2006, 2006.

[9] Julien Delange. *AADL in Practice: Design and Validate the Architecture of Critical Systems*. Reblochon Development Company, May 2017.

[10] Julien Delange and Peter Feiler. Supporting safety evaluation process using aadl. SEI, 7th Layered Assurance Workshop, 2013.

[11] Peter Feiler and David Gluch. *Model-Based Engineering with AADL*. Addison-Wesley, 2013.

[12] Peter H. Feiler, David Gluch, and John D. McGregor. An Architecture-Led Safety Analysis Method. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016.

[13] Peter Fenelon and John A McDermid. An integrated toolset for software safety analysis. *Journal of Systems and Software*, 21(3):279–290, June 1993.

[14] Jeremy Hsu. Toyota recalls 1.9 million prius hybrids over software flaw. IEEE Spectrum, Feb. 2014.

[15] John Hudak and Peter Feiler. Developing aadl models for control systems: A practitioners guide. CMU/SEI-2007-TR-014, July 2007.

[16] Carnegie Mellon University/Software Engineering Institute(SEI). Open source aadl tool environment: http://osate.org/, 2005.

[17] International Electrotechnical Commission International Organization for Standardization, Institute of Electrical, and Electronics Engineers. Iso/iec/ieee 24765:2010 systems and software engineeringvocabulary, December 2010.

[18] Kishor S. Trivedi Katerina Goeva-Popstojanova. Architecture-based approach to reliability assessment of software systems. *An International Journal of Performance Evaluation*, 45(1):179–204, April 2001.

[19] John C. Knight. Safety critical systems: challenges and directions. In *The 24 rd International Conference on Software Engineering, ICSE*. IEEE, May 2005.

[20] Nancy Leveson. A new accidental model for engineering safer systems. *Safety Science*, 42(4):237–270, April 2004.

[21] Nancy Leveson. *Engineering a safer world: Systems Thinking Applied to Safety*. MIT Press, 2011.

[22] Nancy Leveson. An stpa primer:version 1. MIT publications, http://sunnyday.mit.edu/STPA-Primer-v0.pdf, August 2013.

[23] Nancy Leveson and Clark Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, July 1993.

[24] Jacques-Louis Lions. Ariane 5:flight 501 failure. In *Technical Report: Report by the Inquiry Board*, July 1996.

[25] Ethan Mcgee and John McGregor. A decision support system for selecting between designs for dynamic software product lines. PhD Dissertation in Computer Science, Clemson University, August 2018.

[26] Anitha Murugesan and et al. Compositional verification of a medical device system. In *n ACM SIGAda Ada Letters, volume 33, pages 5164*. ACM, 2013.

[27] NASA-GB-8719.13. *NASA Software Safety Guidebook: TECHNICAL STANDARD*. NASA, March 2004.

[28] Michael Maile Natalya An and Daniel Jiang. Balancing the requirements for a zero false positive/negative forward collision warning. 10th IEEE Annual Conference on Wireless On-demand Network Systems and Services (WONS), Banff, Canada, March 2013.

[29] National Heart, Lung, and Blood Institute. ”explore pacemakers”. https://www.nhlbi.nih.gov/health/health-topics/topics/pace, Feb.28 2012.

[30] Dennis Nolan. *Safety and Security Review for the Process Industries Application of HAZOP, PHA, What-If and SVA Reviews, 4th Edition*. Elsevier, 2015.

[31] Standard Pacemaker. Pacemaker system specification, software quality research laboratory (sqrl),. Boston Scientific Publisher, Jan.3 2007.

[32] David Gluch Peter Feiler and John Hudak. The architecture analysis and design language(aadl):an introduction. SEI, CMU/SEI-2006-TN-011, February 2006.

[33] David Gluch Peter Feiler, John Hudak and Julien Delange. Architecture fault modeling and analysis with the error model annex, version 2. SEI, CMU/SEI-2016-TR-009, June 2016.

[34] David Gluch Peter Feiler, Julien Delange and John D. McGregor. Architecture-led safety process. Technical Report: CMU/SEI-2016-TR-012, Carnegie Mellon University/Software Engineering Institute, Pittsburgh, 2016.

[35] Julien Delange Peter Feiler, John Hudak and David Gluch. Architecture fault modeling and analysis with the error model annex, version 2. Technical Report: CMU/SEI-2016-TR-009, Carnegie Mellon University/Software Engineering Institute, Pittsburgh, 2016.

[36] Sam Procter. A development and assurance process for medical application platform apps. Ph.D. Dissertation, Kansas State University, 2016.

[37] Sam Procter and John Hatcliff. An architecturally-integrated, systems-based hazard analysis for medical applications. In *Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign, MEMOCODE*. IEEE, November 2014.

[38] Fryad M. Rashid and John D. McGregor. Augmenting a hazard analysis method with error propagation information for safety-critical systems. 35th International System Safety Conference, Albuquerque, New Mexico, USA, 21-25 August 2017.

[39] Fryad M. Rashid and John D. McGregor. Using formal notations to augment a hazard analysis method. 36th International System Safety Conference, Phoenix, Arizona, USA, 13-17 August 2018.

[40] Michael Beauregard Raymond Mikulak, Robin McDermott. *The Basics of FMEA, 2nd Edition*. CRC Press, 2008.

[41] ATSB Transport Safety Investigation Report. In-flight upset event 240km north-west of perth, wa boeing company 777-200, 9m-mrg. *Australian Transport Safety Bureau*, August 2005.

[42] NIST Planning report 02-3. The economic impacts of inadequate infrastructure for software testing. May 2002.

[43] Felix Salfner and Miroslaw Malek. *Reliability Modeling of Proactive Fault Handling*. Humboldt-Universitt zu Berlin, Mathematisch-Naturwissenschaftliche Fakultt II, Institut fr Informatik, 2006.

[44] John Thomas. Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis. Ph.D. Dissertation. MIT., 2013.

[45] John Thomas and Nanacy Leveson. Performing hazard analysis on complex, software and human-intensive systems. The 29th International System Safety Conference, Las Vegas, NV, August 2011.

[46] Nigel J. Tracey and et al. Integrating safety analysis with automatic test-data generation for software safety verification. Proceedings of the 17th International Conference on System Safety, 1999.

[47] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3):53–71, December 2005.

[48] R. Sasseb G. Heinerb Y. Papadopoulosa, J. McDermida. Analysis and synthesis of the behaviour of complex programmable electronic systems in conditions of failure. *Reliability Engineering and System Safety*, 71(3):229–247, 2001.