Supporting model based safety and security assessment of high assurance systems

by

Hariharan Thiagarajan

B.Tech, Anna University, 2009

M.S., Kansas State University, 2012

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2022

Abstract

Modern embedded systems are more complex than ever due to intricate interaction with the physical world in a system environment and sophisticated software in a resource-constrained context. Cyber attacks in software-reliant and networked safety-critical systems lead to consideration of security aspects from the system's inception. Model-Based Development (MBD) is one approach that has been an effective development practice because of the abstraction mechanism that hides the complicated lower-level details of software and hardware components. Standards play an essential role in embedded development to ensure the safety of the users and environment. In safety-critical domains like avionics, automotive, and medical devices, standards provide best practices and consistent approaches across the community.

The Analysis and Design Language (AADL) is a standardized modeling language that includes patterns that reflect best architectural practices inspired by multiple safety-critical domains. The work described in this dissertation comprises numerous contributions that support a model analysis framework for AADL that aims to help developers design and assure safety and security requirements and demonstrate system conformance to specific categories of standards.

This first contribution is Awas - an open-source framework for performing reachability analysis on AADL models annotated with information flow annotations at varying degrees of detail. The framework provides highly scalable interactive visualizations of flows with dynamic querying capabilities. Awas provide a simple domain-specific language to ease posing various queries to check information flow properties in the model.

The second contribution is a process for integrating risk management tasks of ISO 14971 - the primary risk management standard in the medical device domain — with AADL modeling, specifically with AADL's error modeling (EM) of fault and error propagations. This

work uses an open-source patient-controlled analgesic (PCA) pump - the largest open-source AADL model to illustrate the integration of risk management process with AADL and provides the first mapping of AADL EM to ISO 14971 concepts. It also provides industry engineers, academic researchers, and regulators with a complex example that can be used to investigate methodologies and methods of integrating MBD and risk management.

The third contribution is a technique to model and analyze security properties such as confidentiality, authentication, and resource partitioning within AADL models. This effort comprises an AADL annex language to model multi-level security domains along with classification of system elements and data using those domains and a tool to infer security levels and check information leaks. The annex language and the tools are evaluated and integrated into the AADL development environment for a seamless workflow.

Supporting model based safety and security assessment of high assurance systems

by

Hariharan Thiagarajan

B.Tech, Anna University, 2009

M.S., Kansas State University, 2012

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computer Science
Carl R. Ice College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2022

Approved by:

Major Professor
John Hatcliff

Copyright

Abstract

Modern embedded systems are more complex than ever due to intricate interaction with the physical world in a system environment and sophisticated software in a resource-constrained context. Cyber attacks in software-reliant and networked safety-critical systems lead to consideration of security aspects from the system's inception. Model-Based Development (MBD) is one approach that has been an effective development practice because of the abstraction mechanism that hides the complicated lower-level details of software and hardware components. Standards play an essential role in embedded development to ensure the safety of the users and environment. In safety-critical domains like avionics, automotive, and medical devices, standards provide best practices and consistent approaches across the community.

The Analysis and Design Language (AADL) is a standardized modeling language that includes patterns that reflect best architectural practices inspired by multiple safety-critical domains. The work described in this dissertation comprises numerous contributions that support a model analysis framework for AADL that aims to help developers design and assure safety and security requirements and demonstrate system conformance to specific categories of standards.

This first contribution is Awas - an open-source framework for performing reachability analysis on AADL models annotated with information flow annotations at varying degrees of detail. The framework provides highly scalable interactive visualizations of flows with dynamic querying capabilities. Awas provide a simple domain-specific language to ease posing various queries to check information flow properties in the model.

The second contribution is a process for integrating risk management tasks of ISO 14971 - the primary risk management standard in the medical device domain — with AADL modeling, specifically with AADL's error modeling (EM) of fault and error propagations. This

work uses an open-source patient-controlled analgesic (PCA) pump - the largest open-source AADL model to illustrate the integration of risk management process with AADL and provides the first mapping of AADL EM to ISO 14971 concepts. It also provides industry engineers, academic researchers, and regulators with a complex example that can be used to investigate methodologies and methods of integrating MBD and risk management.

The third contribution is a technique to model and analyze security properties such as confidentiality, authentication, and resource partitioning within AADL models. This effort comprises an AADL annex language to model multi-level security domains along with classification of system elements and data using those domains and a tool to infer security levels and check information leaks. The annex language and the tools are evaluated and integrated into the AADL development environment for a seamless workflow.

Contents

List of Figures

List of Tables

Acknowledgments

First and foremost, I thank my major professor, Dr. John Hatcliff, for his consistent support, guidance, patience and kindness throughout my graduate studies. His insightful comments and constructive criticisms shaped my problem-solving and technical communication skills. I would also like to thank Dr. Robby for grooming my software engineering skills and for the opportunity to work on cutting-edge industrial technologies. I immensely enjoyed working with Dr. Eugene Vasserman in developing the security aspects of a safety-critical system.

I am thankful to Dr. David Schmidt and Dr. Torben Amtoft for the engaging lectures and discussions that encouraged me to work on static analysis. I am grateful to Jason Belt and Dr. Sam Procter for all the feedback and collaboration through the years. Thanks to all students and other members of SAnToS Laboratory for the exciting discussions and lively workspace.

I want to thank Rand Whillock, Dr. Robert Edman, Todd Carpenter from Adventium Labs, and Dr. Peter H. Feiler and Lutz Wrage from Software Engineering Institute for their collaboration on several projects.

I am grateful to my wife, I could not have completed this dissertation without her love and support. Finally, my sincere gratitude goes to my mother, father, sisters, and both my brothers-in-law for their patience and belief during my long pursuit of Ph.D.

Dedication

This dissertation is dedicated to my family, without whom this endeavor would not be possible.

Chapter 1

Introduction

> *"There is a race between the increasing complexity of the systems we build and our ability to develop intellectual tools for understanding their complexity. If the race is won by our tools, then systems will eventually become easier to use and more reliable. If not, they will continue to become harder to use and less reliable for all but a relatively small set of common tasks. Given how hard thinking is, if those intellectual tools are to succeed, they will have to substitute calculation for thought."*
>
> – *Leslie Lamport*

In recent times, critical systems are larger and more complex than ever due to sophisticated needs from the market. An important factor in the growing complexity of systems is the increased use of software to implement system's behaviors. Moreover, critical software components and the computing platform execute in a resource-constrained environment.

A common approach to tackle large systems is to decompose systems into sub-systems and utilize off-the-shelf components where ever possible. In this approach, multiple organizations can concurrently develop sub-systems independently and thus significantly reduce time-to-market. This contrasts with the development of many legacy systems, in which a single organization developed the entire system as a monolithic unit. An ideal goal is to develop interoperable and reusable sub-systems that are usable in varying contexts.

Aerospace, military, and medical are some of the major industries that develop critical systems. Failures in these systems can result in huge losses. Therefore hazard analysis, risk assessment, and reliability estimates are vital activities performed in developing these systems to avoid accidents and improve trust.

One approach to developing systems of systems is to use a model-based system engineering (MBSE) methodology. Developing a system model captures of the overall architecture and assigns responsibilities among stakeholders. In a distributed development with multiple stakeholders, integration failure is a common concern. Identification and rectification of integration failures are expensive due to encountering integration failures late in the system integration process, typically after developing individual components. Architecture and Analysis Definition Language (AADL) is a standardized architecture description language[1] enabling: (a) engineers to define a vocabulary and modeling elements for common architecture and coding patterns used in real-time embedded systems and, (b) modeling, analysis, and code generation tools to help engineers design and implement a system. These capabilities enable developers to design and address important engineering challenges before considerable monetary investments are associated with implementing and testing the system.

In the medical device domain, risk management is a crucial activity in the development and certification. The international standard ISO 14971 describes the risk management process for medical devices. The 14971 process includes identifying hazards (things associated with the device and its use that might cause harm), performing risk analysis (including hazard analysis) to identify hazardous situations (causal chains leading from root causes to device-user / device-patient interactions that might cause harm), developing risk controls (mitigations of hazard situations), verifying risk controls, and determining if residual risks are acceptable.

A vital element of the interoperability vision is that reuse is not just limited to subsystem implementations — the component's risk management and assurance results should also be reusable when sub-systems are integrated into a new system context. The interoperability goal also introduces network capability to devices that are traditionally independent and monolithic. In interconnected systems, safety concerns resulting from security issues

can often be overlooked as traditional risk management processes were developed when safety critical systems were not networked and thus not as susceptible to security problems. However, more recent standards and guidance documents such as AAMI TIR57 suggest an interleaved safety and security risk management process[2] for monolithic medical systems.

In modern military systems, manufacturers need to consider security aspects from the early design of the system. The MILS approach to designing secure systems decomposes the system to identify security critical parts and develop them as formally verified trusted components/processes in a separation kernel. A verified security kernel offers isolation of resources, faults and provides communication with other components through specialized trusted components. In distributed systems, the trusted components are physically isolated in single-user machines and share resources through trusted mediation components. MBSE provides a robust structure to model and analyzes security aspects much earlier in the system development.

Although distributed development copes up with the growing scale of the system. In practice, it poses the following challenges:

- In a distributed development context, safety and security aspects of the system often span multiple organizations and many sub-systems. Thus, it is essential to have a common understanding of various dependencies in the system and the respective responsibilities of the involved vendors.

- In a multi-vendor development context, various notions of dependence analysis are key to gaining system understanding and supporting safety and security audits and assurance cases. In hazard analysis, understanding how faults propagate through the system due to dependencies helps develop better hazard mitigation strategies. Analyzing security aspects of the system requires comprehending the flow of information through data and control channels. Additionally, over the system's life cycle, upgrading and re-integrating components require an understanding of the impact of the changes.

- The risk management standards such as ISO 14971 are developed for medical systems developed by a single vendor. With the advent of distributed development, adapting

existing risk management standards to distributed context is complicated. The Medical device plug-and-play (MD PnP) and Integrated Clinical Environment (ICE) standard for developing interoperable systems provide a wealth of resources on compositional risk management, security concerns, and providing safety arguments derived from multiple stakeholders. However, applying them to a novel device is not a straightforward task.

We address these challenges by providing a general dependence analysis framework that can be used in a model-based development to support system understanding, security analyses, and analyses used in risk management activities. While the proposed approach is not restricted to a particular modeling language, we use the industry-standard AADL modeling language to illustrate and evaluate the proposed approach. The specific contributions of this dissertation are as follows:

- A general-purpose dependence analysis infrastructure, called Awas[1], supports model-based specification of dependencies, automatic derivation of dependencies from system architecture models, and analyses of these different types of dependence information.

- Tool for creating interactive visualizations of Awas dependence information and analysis results. The tools create projections of dependence information from models, enabling developers and auditors to better focus on dependence-related concerns without being overwhelmed by the scale of the model or the details in the model that are not relevant to the task at hand. The generated visualizations are independent of the platform and modeling tools, enabling both technical and non-technical staff to understand and communicate aspects of the model.

- A query language for interrogating information flows and causal relations in systems at varying stages of modeling. This query language and supporting analysis can help developers explore and understand causality relations and "what if" scenarios in Awas-support security and safety analysis activities and help design and verify security and risk controls.

[1]Awas means "caution" in Indonesian

- A validation and assessment of Awas using one of the largest and most complex medical device examples considered in the academic/industry literature to date. This work includes an approach for (a) developing model annotation libraries that instantiate the AADL EMv2 framework to support ISO 14971 medical device risk management and (b) auto-generating risk management reports that relate sub-system failures to system-level hazards.

This work developed/applied in the following projects:

- **Security & Safety Co-Analysis Tool Environment (SSCATE):**
  This project focused on incorporating hazard analysis into AADL system models. We developed the initial version of the Awas analysis and visualization tool to query and identify hazardous paths in a system model.

- **Cyber Assured Systems Engineering (CASE):**
  This program's objective was to develop the design, analysis, and verification tools for engineering cyber resilient embedded computing systems. We applied Awas to visualize and support assurance arguments concerning Unmanned Aerial Vehicle's (UAV) security properties.

- **Microkernel Application Information fLow with Logic-based Enforcement:**
  This project aims to provide an integrated model and code-level information flow analysis tools to increase aircraft survivability. We developed a secure modeling annex for AADL and extended Awas to infer security types and check the system's security information flow policies.

- **Integrating Safety and Security Engineering for Mission-Critical Systems:**
  With the maturity of Awas tools, Software Engineering Institute (SEI) applied Awas in their safety and security hazard analysis and report generation platform for modern mission critical embedded systems.

The primary contribution of this work is the open-source implementation of the dependency analysis framework for AADL available under an open license[3]. Additionally, we

provide models, tutorial materials, and user documents for all the tools.

The rest of this dissertation is organized as follows:

Chapter 2 provides the context of system engineering and some of the aspects of developing safety critical system using MBSE. The following section describes popular modeling languages, risk management standards for medical devices, and existing hazard analysis techniques.

Chapter 3 describes the AADL modeling language and its information flow aspects with the help of a security critical system. *AAMI/ANSI/IEC TIR80001* risk management standard, an extension of *ISO 14971* to support networked systems and demonstrate the process using interoperable medical devices. Subsequently, demonstrated iterative development of a safety critical system.

Chapter 4 reviews the formal theories behind the analysis and modeling of the systems and describing some of the existing tools designed to perform model-level safety and security analysis.

Chapter 5 presents Awas - a tool for transforming of AADL EMv2 models into dependency graphs and performing reachability analysis on varying levels of model details. The content of this chapter have been published in the journal – *Innovations in System and Software Engineering*[4].

Chapter 6 describes the Awas tool architecture, visualization of AADL models and dynamic interaction, and querying capability. The content of this chapter have been presented in the conference – *moDeling, vErification and Testing of dEpendable CriTical systems*[5].

Chapter 7 demonstrates applications of the information flow analysis specifically, automating parts of *ISO 14971* medical device risk management process presented in the conference – *International Symposium on Model-Based Safety and Assessment*[6]. The second application is developing an AADL annex and analysis for modeling and analyzing MILS-based security critical systems.

Chapter 8 provides a performance evaluation of Awas using a collection of publicly available AADL models and, based on the feedback from the industrial partners, various integration of Awas with OSATE - an Eclipse-based AADL development environment.

Chapter 9 conclude by summarizing the work and providing direction to extend this work to develop distributed critical systems.

Chapter 2

Literature Review

To provide a concrete illustration of risk management standards for critical systems, this chapter summarizes some key medical device risk management standards, specifically ISO14971, and focus on risk analysis. Since hazard analysis is a key component of risk analysis, a survey of hazard analysis techniques is provided that includes Fault Tree Analysis (FTA), Failure Mode Effect Analysis (FMEA), and System Theoretic Process Analysis (STPA). Finally, challenges are identified related to critical system communication and security as well as the overall development process.

2.1 System

With the miniaturization of the computer processor, computing became *anytime* and *anywhere*. This was predicted by Mark Weiser who coined the term "ubiquitous computing" [7]. Embedded systems are an important class of computers that control the physical environment in a feedback loop. A device senses a certain aspect of its environment, a controller performs a series of calculations to determine if the environment should be acted on. Based on the result the controller sends commands to actuators to modify the environment [8].

Embedded systems such as modern cars, smart toasters, washers, even mobile devices share common characteristics, for example, real-time constraints, safe operation, reliability,

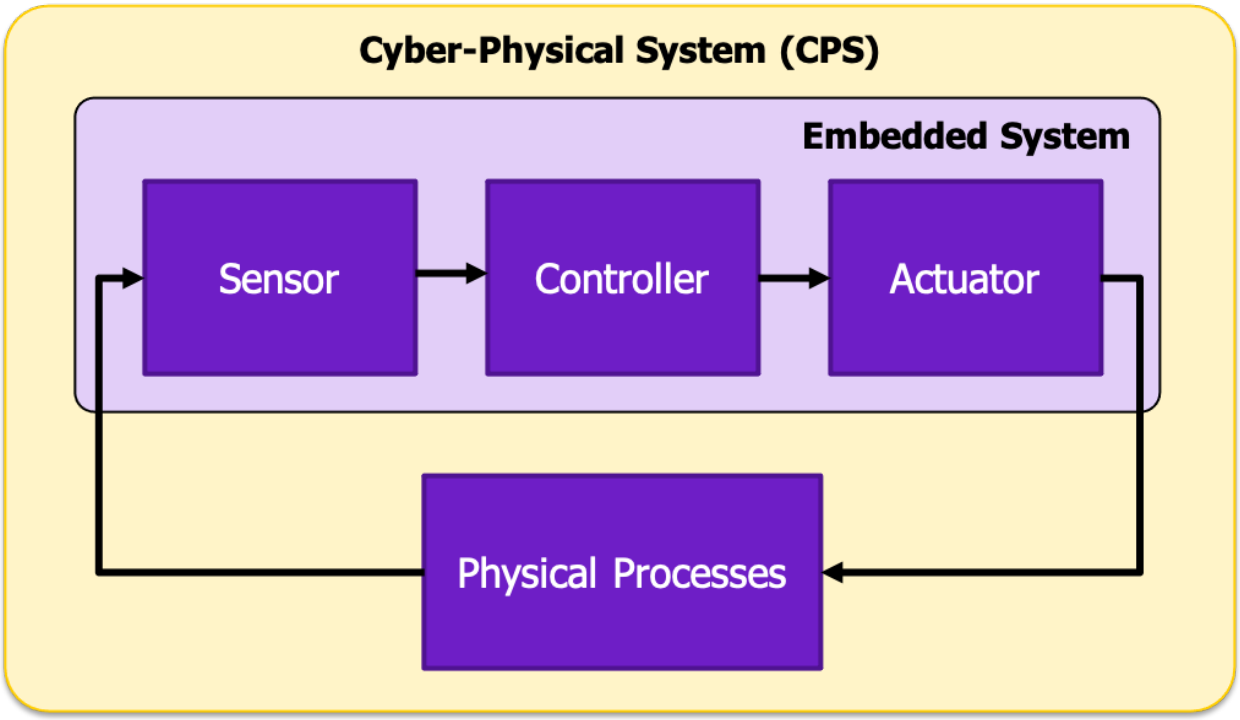


Figure 2.1: Cyber-Physical System - Control Structure

and efficiency requirements. Along with these common properties, they also share a strong link to their physical environment. This common attribute of manipulating the environment led to the introduction of the term "cyber-physical systems" (CPS), defined as the integration of computation and physical processes[9]. In such systems, the physical processes affect computations and vice versa. Figure 2.1 captures the relation between the computer controller and the physical process in a feedback loop.

In practice, a system consists of multiple sensors, controllers, and actuators. Furthermore, each can be atomic (not further decomposed) or a system (with subordinate system elements) working together to achieve a common purpose. The term *controlled process* is used to address both physical processes or a sub-system that the controller controls. In this way, a hierarchy among the system elements is established along with the system boundary at each level. If each system element can also be deployed as an independent system, then such a composition of system elements is termed as a 'system of systems'.

"Internet of Things (IoT)" refers to the pervasive presence of system elements (*things*) such as sensors and actuators, and are connected to the controller through the internet with

unique addressing schemes[10]. Most often, IoT systems employ heterogeneous sensors/actuators and varying connectivity. A key characteristic is interoperability, enabling heterogeneous devices to communicate and collaborate to achieve the system's intended purpose.

In recent times, automobiles, aircraft, nuclear power plants, medical devices, financial systems, manufacturing and industrial automation systems, and home automation rely on software to provide improved and sophisticated services. Complex software components in critical systems pose unique challenges in system engineering, risk management, hazard analysis, and certification of embedded systems.

2.2 System Engineering

System engineering is an iterative process of design, develop and operation of a system that satisfies the requirements in an efficient manner[11]. Irrespective of the system domain and its use-case, engineering an embedded system is challenging due to the myriad design choices, specialized development process, and higher degree of assurance requirements to ensure the system's dependability.

Traditionally, system engineers followed the software development life cycle (SDLC) model such as the waterfall model. However, over the years, the V-model[12] of system development became the de facto process for developing embedded systems and systems of systems. Figure 2.2 shows the various stages of the engineering process, where the left-wing is the project definition, the bottom of the "V" consists of implementation, and the right-wing is the verification process.

As shown in Figure 2.2, this approach defies the project requirements before technology choices and implementation. In the left-wing, the understanding of the system progresses from the high-level concept of operation to a well-defined set of system requirements. In the subsequent stages, a system design is developed to meet the requirements and further decomposed into sub-systems and sub-systems into components. A large system is decomposed into several layers until the system's design consists only of atomic components or third-party components (Commercially of the shelf (COTS) components).

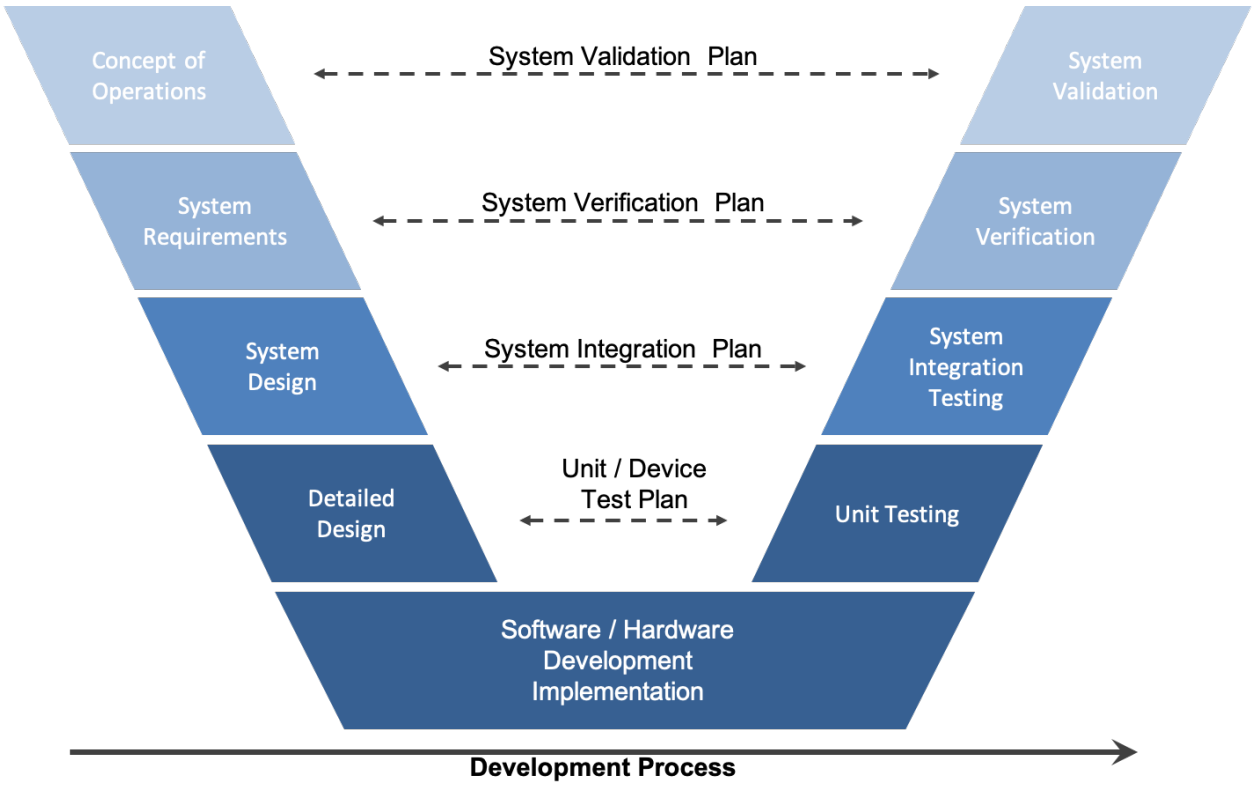


Figure 2.2: V-Model System Development Process

At the end of each stage, the outcome of the stage is discussed with the customer, and an agreement is reached before proceeding to the next stage. The documents generated at the end of each stage guide the development of the following stage. The bottom of the "V" consists of the implementation of hardware and software components. In the right-wing, the system components are then integrated as per the system design and verified against the system requirements. Finally, the validation process ensures that the system satisfies the user's needs and system goals.

The linear waterfall model is bent to form the "V-model" to establish the correlation between the system definition (left-wing) and the assurance (right-wing) stages. The final system is validated against the concept of operation developed as the first stage. Similarly, the system is verified against requirements. The arrows connecting the left and right-wings of the "V" shows the relationship between the stages and ensures that the focus of the development is always within the goals of the system.

2.2.1 Challenges

Over the years engineering embedded system evolved and engineers learned from experience. However, the demand for novel and complex systems is ever growing. Here are some of the challenges faced by engineers in developing embedded systems.

Safety

One definition of safety used by many experts is "freedom from harm" [13;14]. At the boundary of the embedded system, it interacts with its environment. If there is a system failure, the system often performs an adverse action on its environment, including humans, expensive equipment, nature, or intellectual property. Due to the magnitude of the loss stemming from an accident, system safety is factored in from the first principles of system engineering. However, it is not possible to identify and eliminate all possible failure scenarios efficiently.

Security

Security is becoming an increasingly important topic in the field of system engineering. For successful operation, it is important to address security issues comprehensively. Networked systems and IoT devices should account for outside attacks and failures. Anytime a device is connected to the internet, the system engineer has to take into account unauthorized personal accessing sensitive information. A system engineer should consider the flow of information through various system parts, specifically between secure and public domains. Through out the life of a system, a system engineer should consider who gets to access the system based on the following properties.

- Confidentiality: Only authorized users can gain information
- Integrity: Authorized users access accurate and complete information
- Availability: Authorized personals can access information provided by the system

Safety of the critical system may be compromised when the system fails to provide its service to its authorized users. For example, a networked medical device fails to alert its

authorized users when a patient's vitals are deteriorating due to Denial of Service(DOS) attack on its network. On the other hand, when an unauthenticated user with malicious intent gains access to the system, then they may add or block sensed information, block or modify commands to the actuator to cause damage to the users and environment or disable the system to deliver services.

Reliability

Reliability is the probability of lack of failure that deviates a system from its goals. The reliability of a system is only based on the durability of components and lack of internal failures during regular operation. Various factors affect the system's reliability, including its operating environment, component choices, redundancy, and design. Engineers make a careful trade-off between cost and reliability.

There is subtle difference between safety and reliability of the system. In reliability engineering, engineers tries to reduce the failure rate. On the other hand, safety engineers tries to avoid hazards. Failures and hazards themselves corresponds to the violation of functional requirements and safety requirements. For example, an aircraft that fails to airborne is safe yet it is unreliable. Similarly, in laser tonsillectomy, if the system is not designed to avoid oxygen tube, it can cause fire accident in patient's windpipe. Such a system is reliable to deliver its service but the service is not evaluated for safety.

Efficiency

Efficient use of resources is paramount in embedded systems. Often it dictates the design choices and the success of a product. The following are some of the areas where the efficiency is focused.

- Energy: With the widespread use of portable embedded devices, choosing energy-efficient hardware and software components is crucial for performing its task.
- Cost: Consumer electronics are mass-produced, and the cost-effective device always edges out in the competitive market.

- Weight: Low weight is essential for portable devices and devices used in avionics and space exploration, and striking a balance among the three is a challenge in developing an embedded system.

Heterogeneity

Sifakis *et al.* describes heterogeneity as the property of the embedded system to be built from components with different characteristics[15]. Two challenges come with heterogeneity: a) Modeling systems with a composition of heterogeneous components without the loss of interoperability, and: b) Designing and integrating heterogeneous components that are compatible and safe.

Technical

With the rapid improvement in technology, keeping pace in other areas such as adaption rate, standardization, development throughput, and testing needs is difficult. Embedded system hardware consists of limited resource availability and developing software that avoids inefficient hardware resource use is challenging.

2.3 Model Based System Engineering (MBSE)

MBSE is a methodology for developing complex and large systems that emphasizes system model as central engineering artifact. In MBSE, a system model acts as the center point where the other phases of system development, such as requirements engineering, the concept of operation, design, analysis, verification, and validation, are captured in it. MBSE reduces cost by early detection of design issues and maintaining a consistent structure.

2.3.1 Document-Centric System Engineering

Prior to MBSE, engineers emphasized text-based documents to capture key aspects of system development. Each stage in the "V-model" and the tractability between stages are docu-

mented. Different teams/stakeholders developing components use documents to capture the interface specifications called Interface Control Documents (ICDs). In a large system with hundreds of components, engineers spend considerable amount of time in developing and maintaining ICDs for each component. However, there is some clear advantage to this engineering approach.

- Both technical and non-technical personal can author and consume these documents
- Sharing documents between teams and stakeholders do not require any additional steps
- Documents are written in natural language and provide greater flexibility to support any situations

2.3.2 Model-Centric System Engineering

In a model-centric approach, developing models is the focus of the engineering effort. A model is a simplified version of something–a graphical, mathematical, or physical representation that abstracts reality to eliminate some complexity[16]. Engineers Contribute to a common model or set of models to capture information about the system. These model artifacts are progressively refined and analyzed until the models represents a system.

Although the document-centric approach is simpler, there are some apparent downsides to it. The top on the list is the sheer number of ICDs generated and the lack of linkage between them. Additionally, artifacts in the documents, such as system architecture diagrams are static. If a component's interfaces are updated, engineers have to manually perform the change impact analysis *i.e.* finding the impact of the change throughout other documents and their changes. This process can quickly become very time-consuming with the growing system's size. Finally, manually checking for integration issues in a critical system can lead to inadequate detection and investigation of hazards.

The model-centric approach is a product of the digital world. A model captures several important system engineering functions. However, engineers document the summary report at the end of each stage and the communication artifacts outside the engineering team. The

key advantage of the model-centric approach is the consistency of the model throughout the engineering process. A model captures dependencies across teams, and the effect of a change made in one part of the model is immediately reflected in the rest. Additionally, a system can be hierarchically refined and decomposed into sub-systems, and a model consistently translates the structure and behavior to the sub-system. Overall, in a model-centric approach, integration failures are detected and mitigated at the design stage in the "V-model" rather than at the integration testing phase, thus eliminating the cost of redevelopment and testing the failed component.

A system engineered using MBSE suffers from higher initial cost due to the cost associated with defining the process, developing infrastructure, training engineers, and modeling[17]. Systems with a longer operational lifespan benefit from adopting the MBSE approach. Industries such as transportation, aerospace, defense, medical device, energy, and industrial equipment see a higher return on investment using MBSE . On the other hand, finance, business services, retail, and high-tech sectors do not seem to benefit as much from MBSE .

2.4 Safety-Critical System

Embedded systems, by definition, interact with their environment, and if a failure could lead to an unacceptable consequence, then such a system is called safety critical system. John Knight defined safety critical systems as those systems whose failure could result in loss of life, significant property damage, or damage to the environment[18]. The distinguishing factor in safety-critical system engineering is the additional effort to ensure the system's safe operation. Risk management effort, strict adherence to safety standards, and certification are key aspects of the safety-critical systems development process.

2.4.1 Interoperable Medical Devices

Modern medical devices are typically specialized computers. For example, a pacemaker can be understood as a computer that controls the pacing pulse, timing, and intensity. Many

surgeries such as hip replacement, spinal surgery, and ophthalmic surgery are assisted by computerized equipment. In recent times, with the increasing sophistication of medical devices, the manufacturers are incentivized to develop devices with interoperability and platform support.

Medical Application Platform (MAP) is an emerging research focus for developing the system of systems medical devices, and its associated standards and risk management[19;20]. A MAP is a safety and security critical real-time computing platform for (a) integrating heterogeneous devices, medical IT systems, and information displays via a communication infrastructure and (b) hosting application programs ("apps") that provide medical utility via the ability to both acquire information from and update/control inte- grated devices, IT systems, and displays[21]. Developing devices for a platform can lead to an ecosystem of manufacturers that collaborate to develop reusable components encouraging device manufacturers to develop interoperable ecosystem-based plug-and-play medical devices. Integrated Clinical Environment (ICE)[22] is a ASTM F2761-2009 standardized architecture realization of MAP. The CIMIT[23] Medical Device Plug-and-Play program (MD PnP)[24] at Massachusetts General Hospital is one research group that made a significant effort to use the ICE architecture to showcase the benefits of interoperable and interconnected medical devices.

2.4.2 Stakeholders

Kim *et al.*[25] identified the following list of stakeholders in developing interoperable medical devices based in the context of ICE platform.

Consortium

The consortium is a central organizational authority that provides architecture standards, interface requirements, and compliance processes. Consortium also provides reference devices and modeling vocabulary that other device manufacturers can use to communicate with stakeholders. Finally, the consortium provides risk management guidelines and processes for component compliance evaluation[24;25].

Component Vendors

Component vendors include device manufacturers, interoperable apps authors, and platform providers. Irrespective of their role, all component vendors must provide interface specifications of their component according to the guidelines provided by the consortium, submit their component for compliance testing, and submit for regulatory review.

Third-Party Certification Authority

Certification authorities ensure that the component complies with the consortium's architecture standards. Certification authorities perform interface testing, ICE complainant testing, and issue certificates to components based on the results. A system integrator can compose certified components with confidence.

Regulatory Authority

Authorities ensure safe and effective use of the interoperable components. The safety of each component is evaluated in a broad set of contexts instead of a system instance. Regulatory authorities make sure that the certification authorities are competent and produce enough evidence to substantiate their certificate.

Health-care Delivery Organizations (HDO)

HDO comprises hospitals and other health care providers. HDOs procure and deploy ICE systems to improve the quality and effectiveness of the health care provided to the patients. HDOs compose the system based on the clinical needs from the collection of compatible components.

2.4.3 Medical Device: PCA Pump

A PCA infusion pump is a medical device intended to administer intravenous (IV) infusion of pain medication to the patient in various clinical settings. During clinical use, a caregiver (typically nurse) first prepares the PCA pump by loading a vial of medication, priming the

pump's infusion set (tubing and needle), and connecting the pump to the patient via the infusion set. The caregiver then configures infusion parameters (e.g., infusion volume, rate, and duration) on the pump's operator interface and initiates the infusion.

Pain medication (opioid) is prescribed by a licensed physician and dispensed by the hospital pharmacy. A clinician loads the vial into the pump, and attaches the pump's drug dispensing tube to the patient's IV line.

A PCA pump can deliver medication in either a basal or bolus mode, where the former continuously delivers medication at a low rate, and the latter delivers a bulk of medication in a short period of time. The patient can request additional boluses for further pain relief by pressing a hand-held button provided by the pump. Too many bolus request can pose severe overdosing risks to the patient.

Figure 2.3: Example PCA Pump

While PCA pumps (and infusion pumps in general) have allowed for a greater level of control and accuracy in drug delivery, they have been associated with persistent safety problems[26]. Through a study of adverse events and device recalls related to infusion pumps, the US Food and Drug Administration (FDA) concluded that many of these problems appear

to be related to deficiencies in device design and engineering[27]. The increased safety concern led FDA to develop *infusion pump improvement initiative* to enhance infusion pump safety[27], including additional scrutiny of risk controls and supplementary documents such as assurance cases over infusion pumps coming to the market[28].

2.4.4 PCA Pump Interlock Scenario

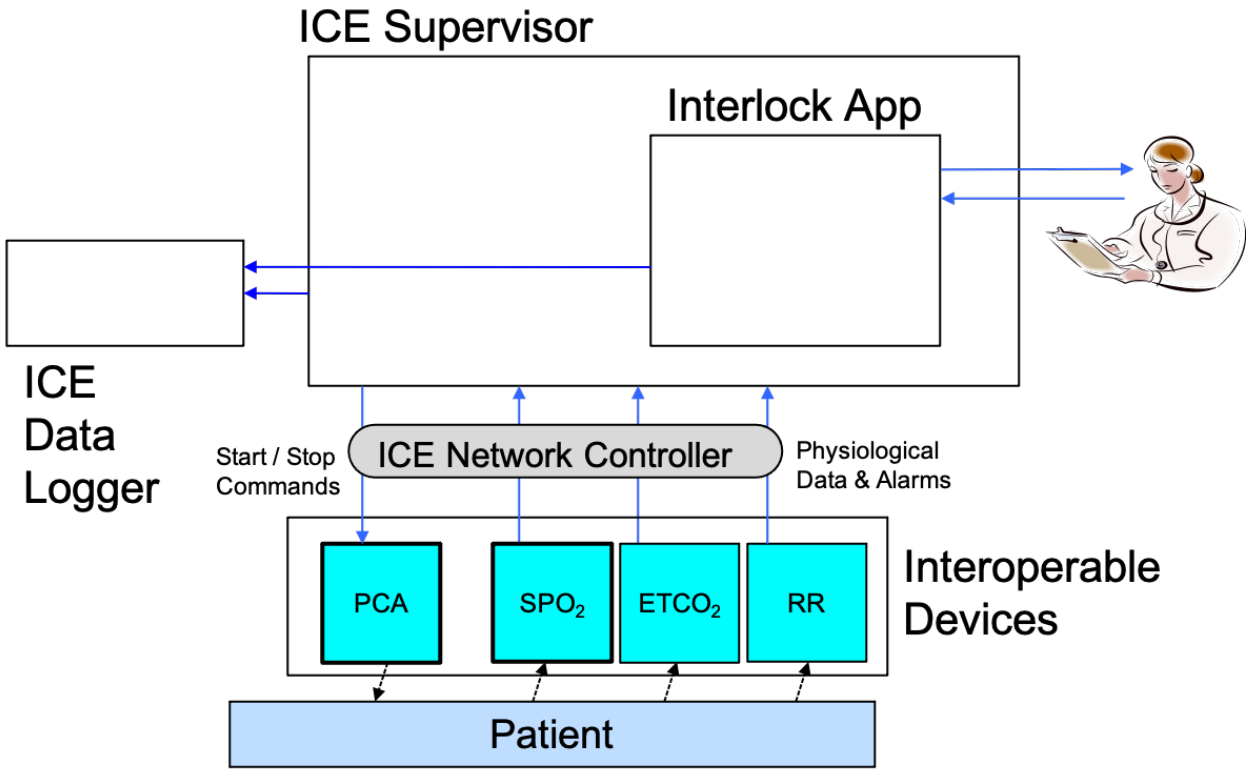


Figure 2.4: ICE Instantiation of PCA Safety Interlock

The most common adverse event in a PCA Pump therapy is drug overdose[29]. Over infusion of opioids may lead to respiratory depression and eventually respiratory distress, which may cause death. May different scenarios can cause over-infusions such as visitors pressing the bolus(PCA-by-proxy), incorrect drug, incorrect dosage, drug interaction, and device malfunction. Patients receiving pain medication are usually also connected to patient monitoring devices. In case of respiratory depression, these monitoring devices sound an alarm and summons a caregiver. However, diagnosis and adequate action may time a while

steamin which damage may have already been done.

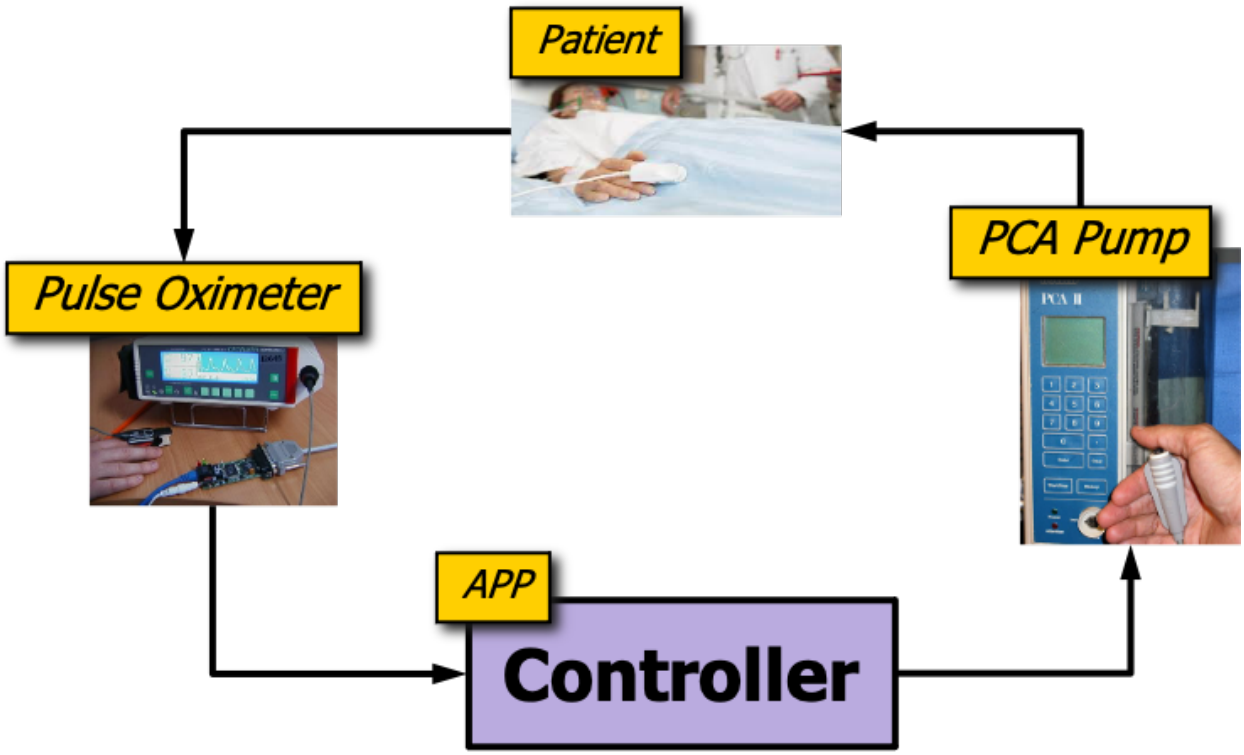


Figure 2.5: Simple PCA Interlock with PulseOx

Researchers proposed the PCA pump interlock scenario to improve the safety of the PCA pumps[21;30;31]. Figure 2.4 illustrates an ICE-based MAP implementation of PCA interlock scenario with interoperable patient monitoring devices, interlock app, PCA pump, and the network controller. In this scenario, different manufacturers develop the components of the PCA interlock system, and the system integrator composes the components as per the model architecture.

Figure 2.5 illustrates a simplified version of the PCA interlock scenario, where a pulse oximeter connected to a patient monitors the patient's current health status. Based on the patient's health condition, an interlock app as described Hatcliff *et al.*[21] halts the infusion when halting conditions (*e.g.* when the patient's respiratory health is deteriorating) are satisfied. Subsequently, if the App determines the patient's vitals to be healthy, it may allow the pump to resume respecting the bolus commands. The interlock App accomplishes this by issuing *START* and *STOP* commands to the PCA Pump. For example, if the last

received command is *STOP*, the bolus activation command is ignored. Similarly, when the PCA Pump receives *START* command, the bolus requests are respected.

2.4.5 Challenges

MAPs are distinct from existing medical systems and safety critical systems due to their unique way of interacting with humans and the environment, assurance requirements, and the involvement of varying stakeholders presents several challenges.

- *Communication process and responsibilities among stakeholders:* With multiple organizations involved, there is a need for a contract to establish communication and trust among the stakeholders. Additionally, a methodology for assigning responsibilities among the stakeholders.

- *Distribution of development process:* With multiple stakeholders, it is efficient for each organization to work in parallel. However, techniques to identify the dependencies and the temporal order of required artifacts are insufficient.

- *Reuse existing components:* It is inefficient for a device vendor or an app developer to rebuild a component for a different platform or a different system. Currently, there is a lack of appropriate guidelines for adapting and reusing existing components without compromising the system's safety.

- *Risk management for interoperable devices:* It is efficient to perform risk management for a broader class of devices from a system integrator point of view. Therefore, the App can be compatible with a wide range of devices. However, who defines the device class? How to reach consensus on the device requirements? Who evaluates for the conforment?

- *Development and automation tools:* To meet the market demands and to incorporate rapid development cycle with higher safety standards, a lot of the development process must be automated. Currently, there is a lack of a unified development environment where the system can be designed and analyzed for safety.

2.5 Modeling Languages and Tools

Engineers develop models to understand better and test a certain aspect of the system. They develop physical models, mock-ups, and abstract models. This work focus on abstract models developed using modeling tools typically running on a computer providing a modeling language to express modeling constructs. The tool checks the model and ensures the construction of a well-formed model. The following sections provide a brief overview of popular modeling languages in industry and academia.

2.5.1 OMG SysML (Object Management Group System Modeling Language)[32]

SysML is a graphical modeling language supporting system engineering phases such as specification, design, analysis, verification, and validation. SysML is a modified version of the Unified Modeling Language (UML) by removing some software-centric diagrams and adding diagrams related to the system engineering lifecycle. SysML consists of nine different diagrams to capture information regarding a system. Among them, the requirement diagram and the parametric diagram are not part of the UML profile. The activity diagram and the block definition diagram are modified from UML to support system engineering tasks. All the other diagrams are the same as the UML diagrams.

Figure 2.6 is the block definition diagram for the PCA interlock system. A block is a fundamental unit for describing the system structure. A block can be a hardware, software, person, or abstract entity. A block diagram defines the structural relation with other blocks.

Figure 2.7 is the internal block diagram for the PCA interlock system. This diagram shows the connections between the ports of a block.

2.5.2 Simulink[33]

Simulink is a commercial toolbox for Matlab[34] from Mathworks. It is a high-fidelity graphical modeling language excelling at modeling, simulation, and analysis of dynamic systems.

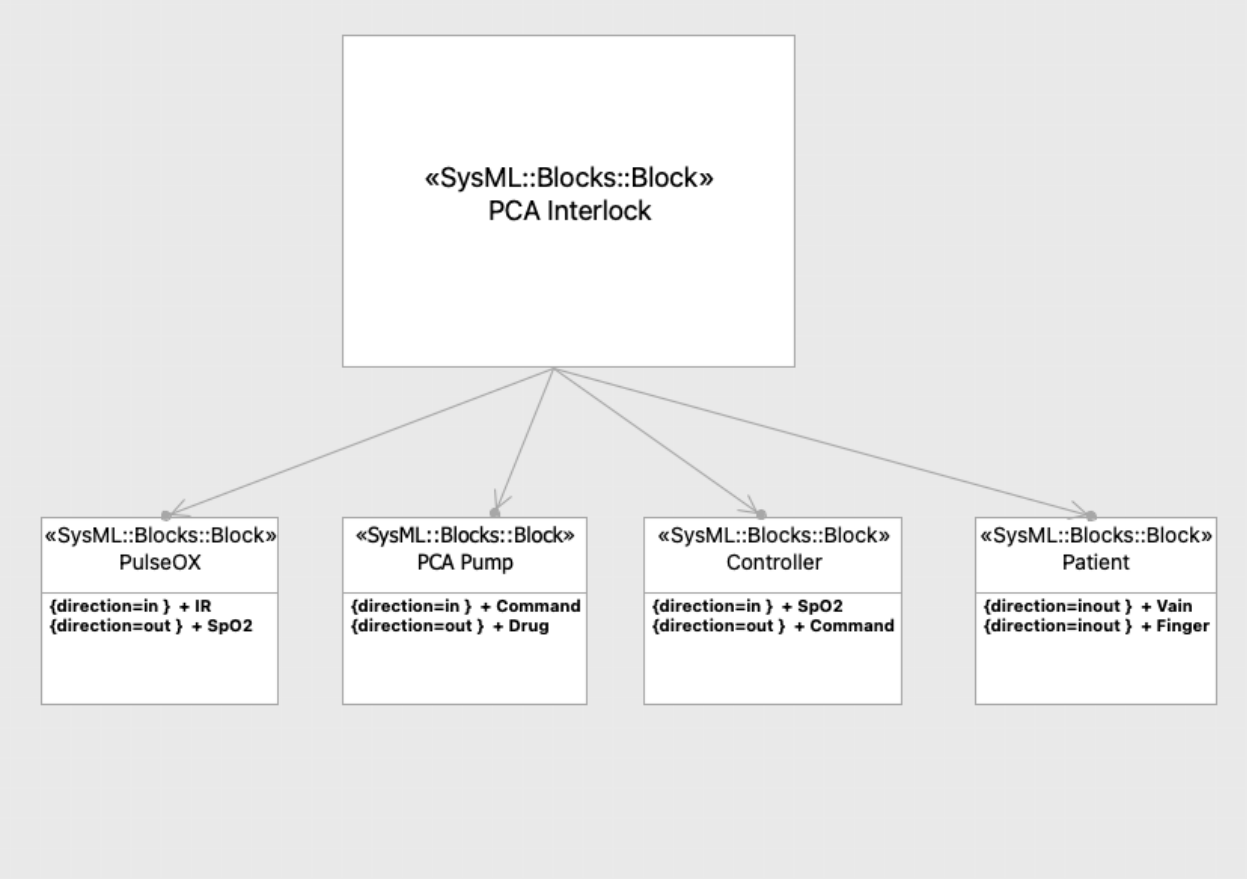


Figure 2.6: SysML block definition diagram for PCA interlock system

Simulink provides a graphical user interface for building block diagrams, and it also provides a library of predefined blocks representing both hardware and software entities. With this collection of blocks, an engineer can build a rapid prototype of a system and analyze the merits of the model design.

Figure 2.8 presents the patient's drug absorption function. Simulink is capable of providing a graphical view to a mathematical function with input and output ports. The patient can be wrapped by a subsystem and used in the PCA interlock system.

2.5.3 Architecture Analysis & Design Language (AADL)

AADL is a SAE standardized architecture description language for modeling real-time embedded systems. AADL has both textual and graphical representations/views as illustrated in Figure 3.1. AADL modeling elements include software, middleware, and hardware compo-

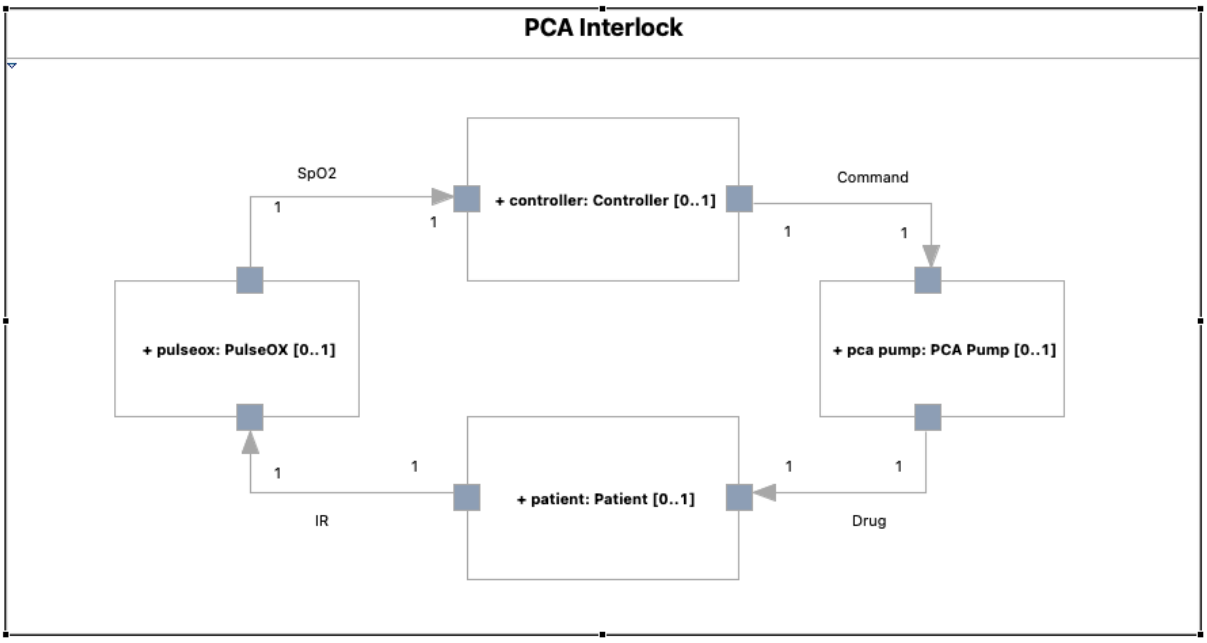


Figure 2.7: SysML internal block diagram for PCA interlock system

nents along with various types of dependency relationships *between* them (inter-component dependencies) and *within* them (intra-component dependencies). Each modeling element can have a variety of *properties* for modeling attributes that specify important characteristics of the elements that may subsequently be leveraged for model analysis or code generation. AADL is an extensible language with an *annex* mechanism to support additional modeling and tool capabilities such as runtime behavior analysis, code generation, error modeling, and user-defined annex.

Figure 2.10 illustrates the instance model generated through the instantiation process. This model captures the simple PCA interlock scenario with components *PulseOx, App, PCA_Pump* and *Patient* represented as a block. The edges between them represent the logical connection in the system. Each component block captures the component's ports and the flow of information within the component between the input and output ports.

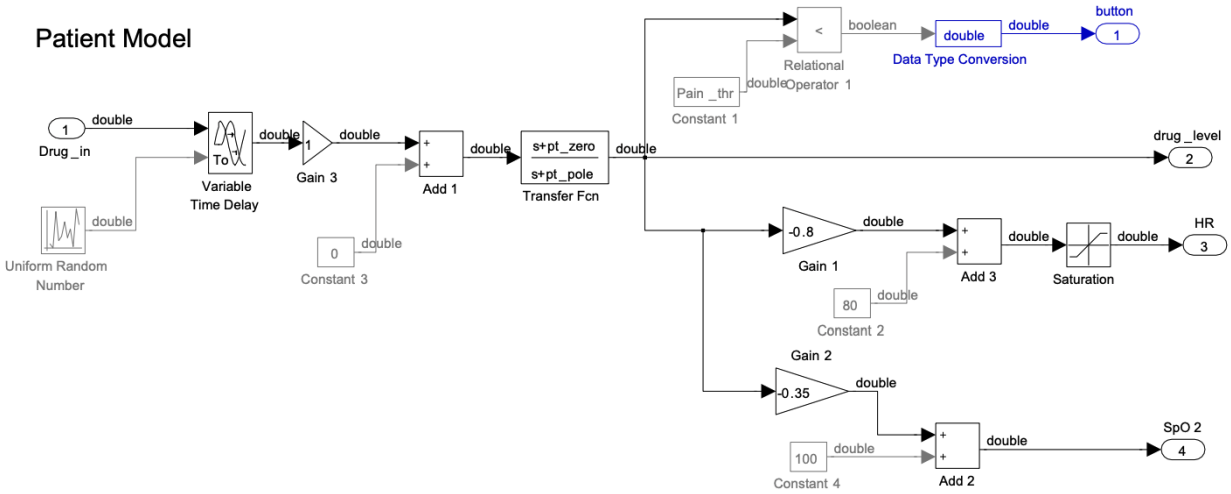


Figure 2.8: Patient model in Simulink developed by Arney *et al.*[30]

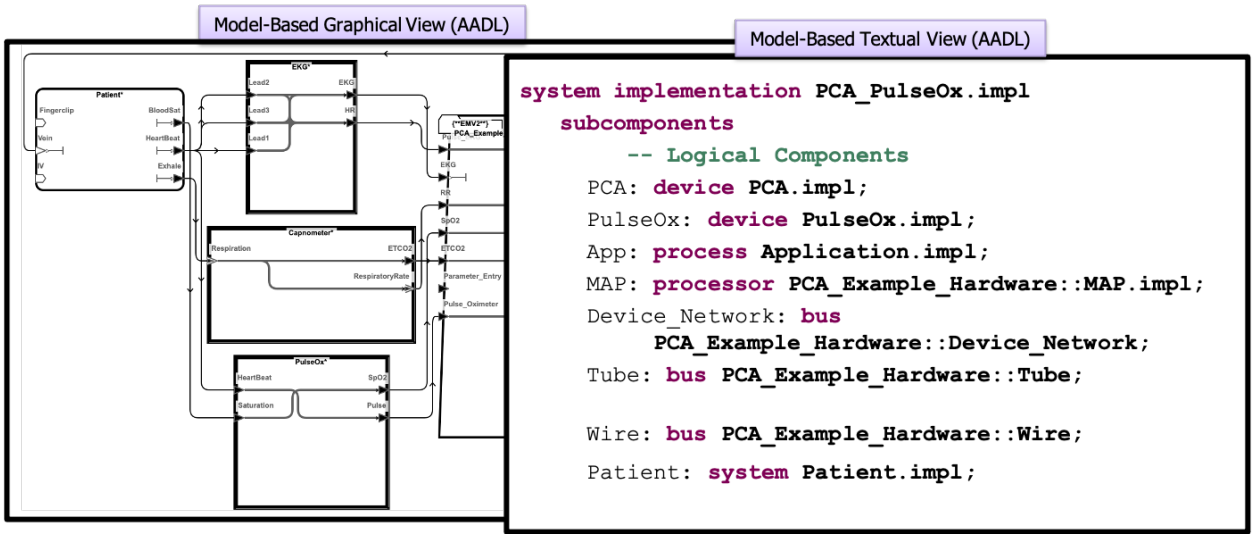


Figure 2.9: AADL Graphical and Textual view

Inter-component dependencies

The most prominent inter-component relationships are *connections*, which capture data and control flows between software components such as threads and processes (e.g., the flow of SpO2 data between the PulseOx and App). Connections associate *ports* on sending and receiving components. AADL includes different port categories to specify communication patterns between components (e.g., asynchronous message passing, synchronous shared memory). Relationships between middleware components can be captured by specifying

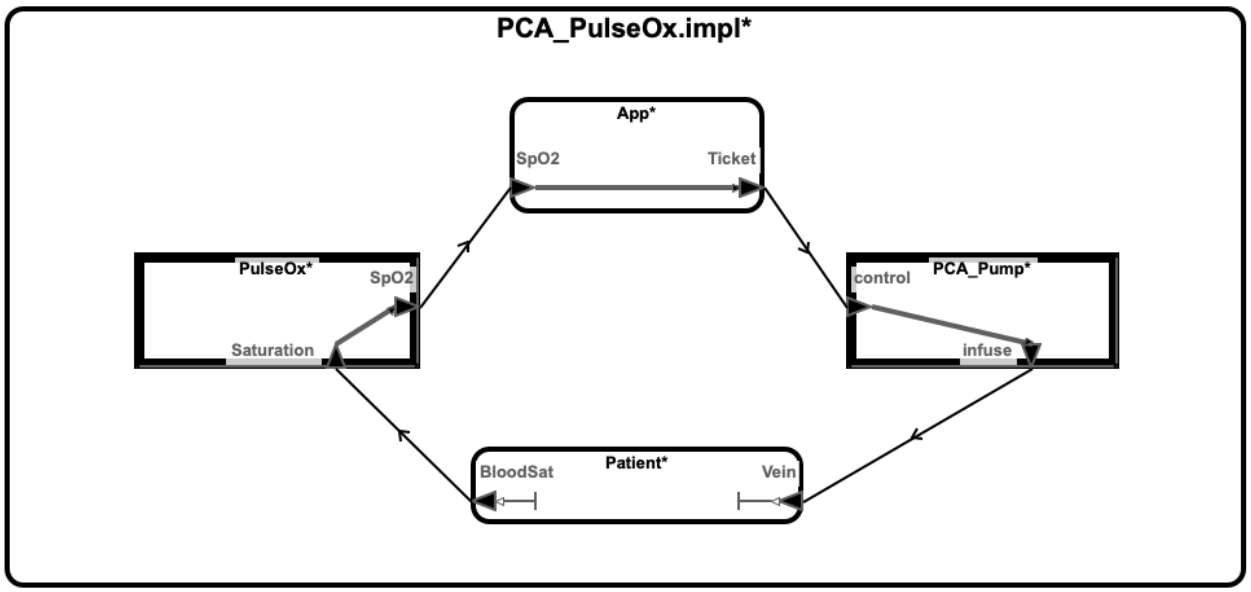


Figure 2.10

connections via *bus accesses* (intuitively, bus access is a feature on a software component indicating that it utilizes a communication substrate). Finally, software elements such as threads/processes and connections can be allocated to middleware and hardware resources such as processors and buses using *bindings*. These dependencies can have multiple layers. For example, a process can first be bound to a *virtual processor* used to model a partition in a hypervisor, and then the hypervisor partitions can be bound to a *processor*. Similarly, a communication *connection* can be bound (transitively) through *virtual buses* representing layers of abstraction and associated protocols in a protocol stack.

Intra-component dependencies

AADL also provides multiple notions of intra-component dependencies. The most basic of these are *flow specifications*, which model data and control flow relationships between a component's input and output ports. AADL does not define precise semantics for flows nor explicitly distinguish between data and control flows. Different analysis tools may give flow annotations different interpretations. For example, a latency analysis tool may consider a flow to model a single or a collection of execution paths through the component source code, with an associated worst-case execution time for the path. A security analysis tool

may interpret the flow as a specification of information flow (e.g., a combination of data and control flow).

2.6 Risk Management

In this section, I will discuss the overall risk management process as per the *ISO 14971* . However, *ISO 14971* does not address interoperability or platform based system. In the work presented in section 3.4 I will present the *ISO 14971* concepts in the context of MAP apps.

To following steps capture the progression of *ISO 14971* risk management process as show in figure 2.11.

Risk Analysis

1. Identify the intended use of a medical device and the possible incorrect or improper use. This is also a good place to identify the safety properties

2. Identify the hazards associated with the medical device in both normal and fault conditions when operated according to its intended use

3. Identify cause and resulting hazardous situation for each hazard identified in the previous step

4. Calculate the associated risk(s) for each of the identified hazardous situations. If the probability of occurrence of harm cannot be estimated with the available information, document the consequence and its severity

Risk Evaluation

1. For each of the identified hazardous situations, determine the foreseeable sequence of events from the root cause and compute the probability along the sequence of events that leads to the hazardous situation

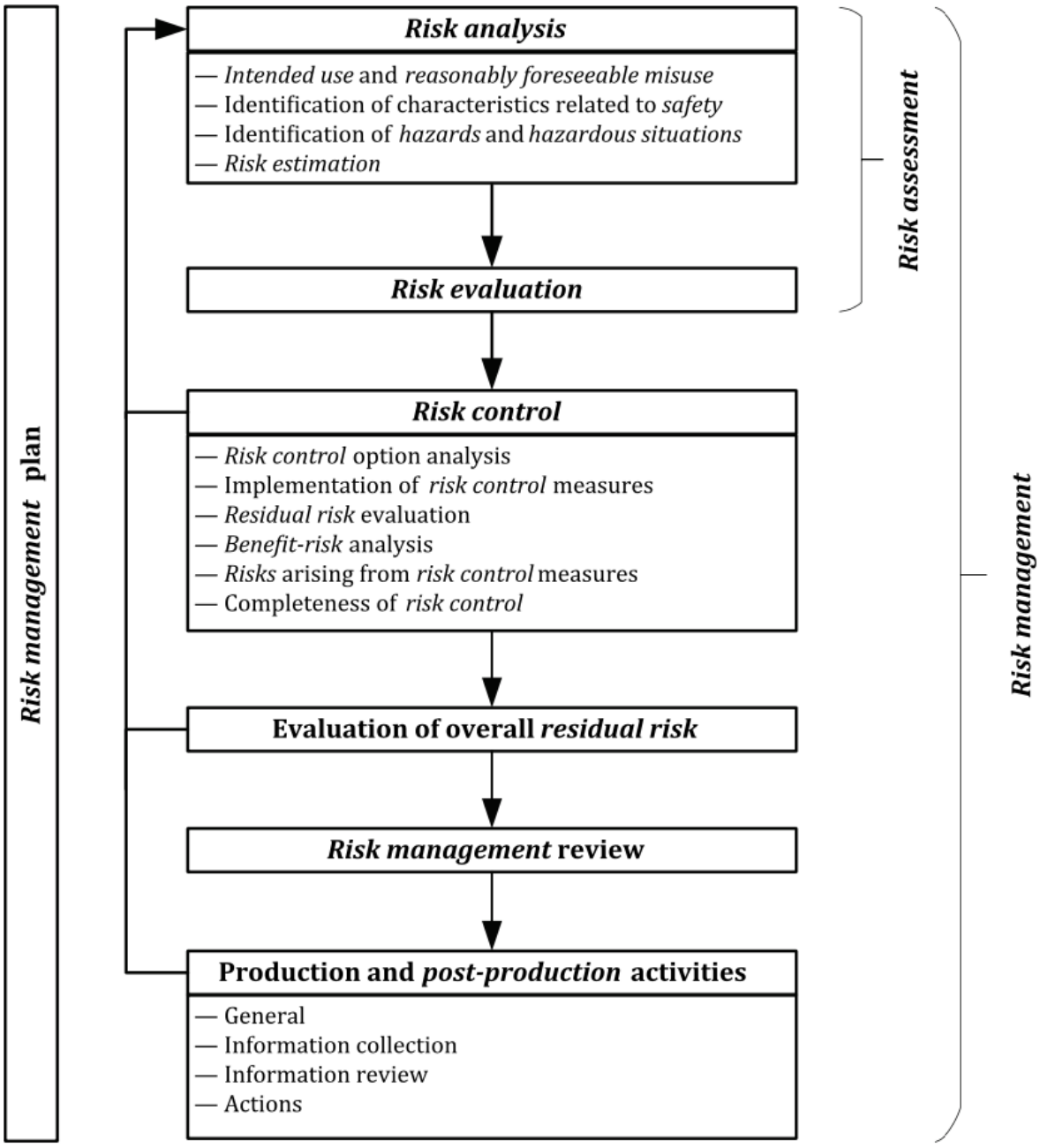


Figure 2.11: ISO 14971 Risk Management Process

2. Check if the level of risk associated with the hazardous situation is acceptable. If it is not acceptable then risk controls must be introduced for the hazardous situation

Risk Control

1. Implement measures to eliminate or mitigate hazardous situations by choosing an alternate design, selecting fault-tolerant components, adding protective measures, and providing documents and warning labels

2. Verify the correct implementation and compute the effectiveness of risk controls

3. Evaluate if risk controls reduce the over risk to an acceptable level. Additionally, ensure the risk controls themselves are not introducing new risks

The results of the activities above are recorded in the risk management file. However, risk management continues throughout the operation of the devices, and continued monitoring and tracking of safety-related and evaluation of the risk controls in the operating field is required.

There are certain temporal ordering constraints on the interoperable medical device development process. For instance, a medical device manufacturer requires, appropriate Risk Management File(RMF) disclosed by the platform provider. Similarly, the App developer requires RMF from all of the devices that are compatible with the App. Therefore it is essential to discuss the requirements for performing risk management from the perspective of stakeholders.

Ecosphere principles of Medical Application Platform(MAP)[25], provides the dependencies and temporal constraint on device development. One of the tasks for device manufacturers is that they provide interface specifications with provided and required capabilities. The process of turning the interface specification to Errors(Failure modes) is discussed in 3.3.1. Section 3.4 discusses the risk management process using the error library.

2.6.1 Challenges of Using *ISO 14971* In Distributed Risk Management

Hatcliff *et al.*[21] listed the following challenges in performing risk managements following *ISO 14971* .

- Specification of boundaries and scope of risk management
- Expanding the notion of intended use and contexts of use
- Conventional terms such as harm/hazard/hazardous situation do not adequately address hierarchical system structures and technical context of use
- Imprecision and ambiguity in describing root causes and observable effects of root causes
- Reporting of component error propagations
- Avoiding separate approaches for reasoning about safety and security root causes
- Different levels of reliability and trustworthiness
- Risk controls are likely to be distributed across multiple items and responsible organizations
- Rapid and consistent development of reliable risk controls
- Verifying individual risk control elements
- Developing appropriate notions of test coverage for interoperability variations
- Phrasing of partial analysis results and partial risk controls in such a way that these can be easily understood and consumed by integration activities
- Communication with external stakeholders and balancing the need to release risk management details for appropriate integration and use while avoiding the release of propriety information

We developed the risk analysis process and the automated risk analysis tool presented in section 7.1 by considering the above mentioned challenges.

2.7 Error Modeling

Risk analysis is the stage of safety-critical system development in which hazards are systematically identified. An analyst develops an error model of the system capturing how an error flows, transforms, and causes undesired events including the hazards that contribute the safety concerns. Analyst performs risk analysis on the error model and develops safety characteristics of the system.

In the avionics domain, specifically, ARP 4761 *guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment* contains "system safety assessment" stage to eliminate the hazard or reduce risks in the system. Similarly, in the automotive domain, ISO 26262 *Road vehicles – functional safety* document contains the Automotive Safety Integrity Level (ASIL)-oriented and safety-oriented analysis section detailing the hazard analysis and risk assessment process. For simplicity, I use the medical domain term *Risk analysis* for all safety critical systems in this dissertation.

2.7.1 Terms and Definitions

We use the following terms throughout this dissertation and their definitions vary between domains. To avoid confusion, we presented the definition that are applicable in the context of medical devices.

System Boundary

The common frontier between the system and its environment

Behavior

What a system does to implement its function. A sequence of states describes both operational and hazardous system behavior.

Structure

What enables it to generate the behavior. A composition of components/system with a set of interaction points.

Service

- In our context, we define *service* as the capabilities of a component. A component may send or receive data or perform an action.
- *Service S* is a sequence of service items throughout the component's operation. In the remainder of this document, the term service is used to represent a service item.
- A *service* is correct only when it satisfies its requirements
- Functional requirements and safety requirements are defined in terms of the properties satisfied by the service.

Service Interface

The part of a system boundary where service delivery takes place[35].

Total State

It is a set of the following states: computation, communication, stored information, interconnection, and physical condition.

External State

The part of a system's state that is perceivable at the service interface
e.g. Pulse rate reading provided by a pulse oximeter

Internal State

The part of a system's state that is not exposed at the service interface
e.g. State of the LED used in the pulse oximeter

Failure

A *failure* is a deviation in behavior from a nominal specification[36]. A component can fail due to internal faults in the component or by providing incorrect inputs. In both cases, the component can no longer function as intended which leads to malfunction and(or) loss of functionality.

Failure Mode

A *failure mode* is a state of the component in which a failure occurs[35]. Triggering a dormant fault places the component in a state that violates the functional behavior of the component. In such a state, service provided by the component deviates from its nominal behavior leading to a *failure*. The service failure modes characterize incorrect service according to four viewpoints.

1. Failure domain: Viewpoint leads us to distinguish
    - Content Failures: The content of the information delivered at the service interface deviated from implementing the system function
    - Timing Failures: The time of arrival or the duration of the the information delivered at the service interface deviated from implementing the system function
    - Halt Failures
    - Erratic Failures
2. Detectability of failures
3. Consistency of failures, and
4. Consequence of failures

Error

According to EMv2[37], the term *error* collectively addresses *failures, failure modes, and incorrect inputs*. An error type is a violation of a property set in functional or safety requirements.

Error token is a value instantiation of an error type.

Fault

The adjudged or hypothesized cause of an error. In the general fault causes errors in the internal state of the system and may or may not affect the external state. Subsequently, it may or may not cause failures. The following are some of the commonly occurring faults.

- Software Flaws
- Logic Bombs
- Hardware Errata
- Production Defects
- Physical Deterioration
- Physical Interference
- Intrusion Attempts
- Virus & Worms
- Input Mistakes

Dormant Fault

A fault that has no at present activated by the internal or external events[35].

2.7.2 Faults, Failures, and Errors

In this section, we attempt to explore the relationship between faults, errors, failures, accidents, and the causation, based on the examples and definitions from the taxonomy developed by Avizienis *et al.*[35] .

1. Fault: Can be *a)* active or *b)* dorment, an active fault is either

- Internal Fault: Fault activated by computation process or environment
- External Fault: Fault activated by application of input to a component

2. Error Propagation: Within a given component, the cause of the error propagation is by computation process(intra-component error propagation). Error propagation from component A to component B that receives service from A(inter-component propagation) occurs when an error reaches the service interface of A. At this time, service delivered by A to B becomes incorrect, and the ensuing service failure of A appears as an external fault to B and propagates the error into B via its interface.

3. Service Failure: It occurs when the propagation of an error reaches the service interface and causes the service delivered by the system to deviate from the correct service. Service failure of a component causes a permanent or transient external fault in the successor component.

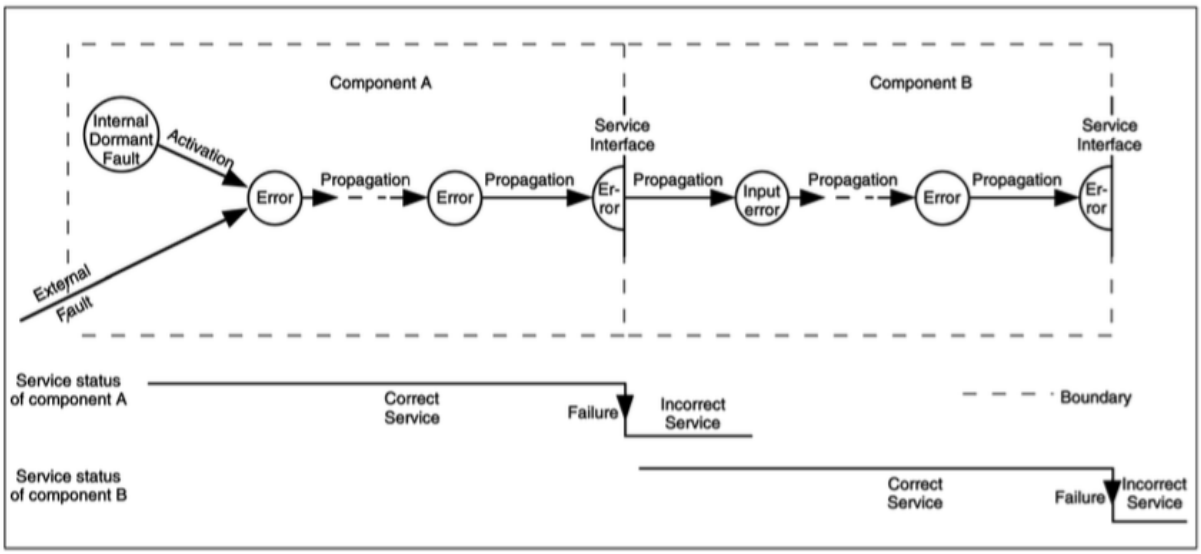


Figure 2.12: Fault, Error, and Failure relation

For example, a programming error that fails to write the correct instruction or data leaves a (dormant) fault in the written software; upon activation, the fault produces an error, and when the error affects the delivered service, a failure occurs. This example is not

restricted to accidental faults: a logic bomb created by a programmer, will remain dormant until activated. When it gets activated, it may produce an error that may lead to a storage overflow or to slowing down the program execution; as a consequence, service delivery will suffer from a denial of service.

2.7.3 EMv2

The AADL Error Modeling annex enables modeling of different types of faults, fault behavior of individual components, fault propagation across components in terms of peer-to-peer interactions and deployment relationships between software components and their execution platform, and aggregation and propagation of fault behavior across the component hierarchy.

EMv2 provides error types to capture categories of faults, failures, and propagations and organize them into error libraries. An error library is a collection of reusable collection of error type and type sets. In an interoperable development, the consortium provides an error library for a class of medical devices, and the library can be extended and adapted by each component vendor.

EMv2 also provides language features to capture component-specific fault model, in three levels of abstraction.

Error Propagation

This is the first level of abstraction, where the errors are specified at the interaction points such as ports, bus access, and bindings. They signify that the interaction point is capable of sending or receiving the specified errors through inter-component dependencies. EMv2 enables users to specify intra-component propagation and propagation of faults between the system hierarchy. The intra-component propagation can be of three kinds:

- *Error Source:* Component act as the source of error propagation may be due to an internal failure
- *Error Sink:* Component is part of the mitigation effort where the incoming error is contained

- *Error Path:* An incoming error is either propagated as is or transformed into a different error based on the component error

Component Error Behavior

A component's error behavior is a factor of its internal events, state, and incoming errors. A component's outgoing propagation depends on the current state of the component and the incoming errors. If a component is in an operational state, it produces a different combination of errors than the failure state (failure mode).

Composite Error Behavior

A component's state transitions can be specified in terms of the state of its subsystems. This specification defines the error behavior of a component based on its internal parts. If one sensor fails, the entire system need not be in a failure state. However, if enough sensors are failing, the system may switch to a safe mode of operation.

2.8 Hazard Analysis

The definition of the term *Hazard* varies from domain to domain. A hazard is an intrinsic property or condition that has the potential to cause harm or damage[38]. However, Ericson *et al.*[39] states that not all hazards are intrinsic properties, and some are the result of inadequate design consideration. Recognizing a hazard is a process of imaging ways a hazard can manifest from an assorted collection of design information. One performs hazard analysis to identify hazards, understand their effects, and pinpoint its casual factors[39]. The goal of performing hazard analysis is to demonstrate how hazards arise and how they may impact the safety of the system. With this knowledge, an engineer can investigate, eliminate or mitigate the hazards, thereby improving the system's safety.

Ericson explains hazard and mishap as two different states of the system related by a state transition. In other words, a hazard is a necessary condition for a mishap. Figure 2.13

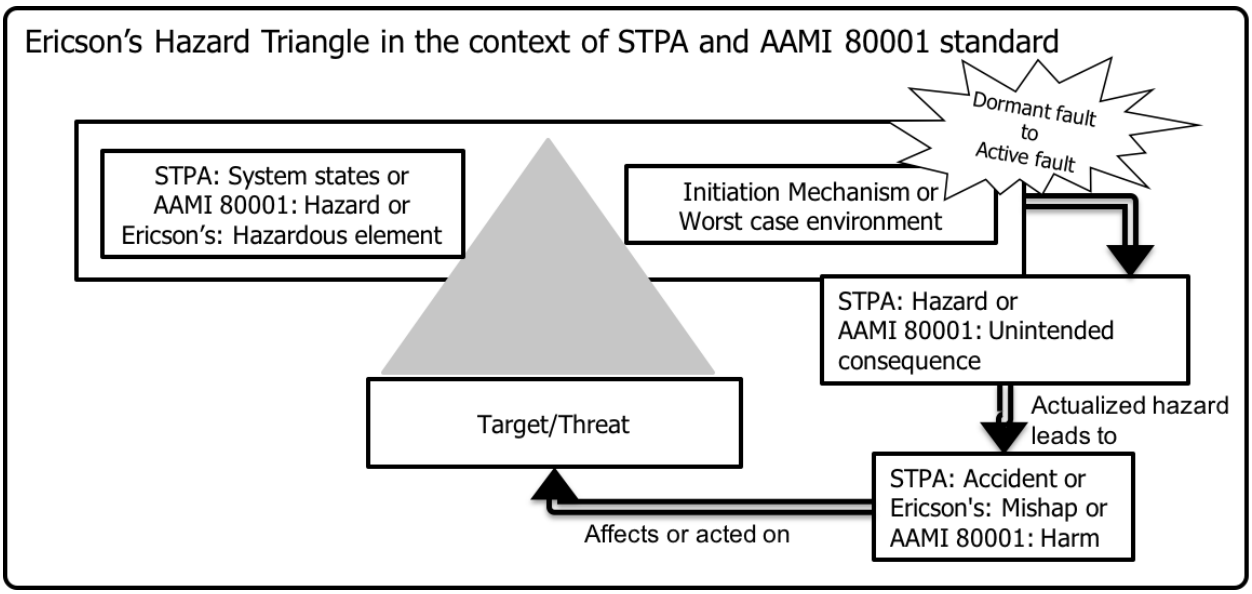


Figure 2.13: Hazard relationship in different terminology

shows the relation between the hazard and mishap and the related terminologies in other analysis techniques and risk management documents. The following section on Fault Tree Analysis (FTA) and Failure Mode Effect Analysis (FMEA) follows Ericson's terminology, where else System Theoretic Process Analysis uses the terms used by Leveson[14].

2.8.1 Fault Tree Analysis (FTA)

Fault Tree Analysis is a top-down system analysis technique used to calculate the root causes and probability. A fault tree is a graphical representation of the various possible combination of events in a system that leads to a hazardous state. Constructed of a fault tree starts from an event that causes a mishap at the root of the tree and proceeds to the children by identifying events or event combinations combined with logical operators that form the root event's necessary condition. The tree's subsequent levels are added by identifying causal events for the parent until a sufficient level of detail is captured. A key benefit of FTA is that it is easy to perform, communicate and understand. The analysis provides valuable information and explores all possible causes for a hazard at the end of the analysis.

Figure 2.14 is a part of FTA for the PCA interlock system. The root node is a system-level hazard that the patient can be overdosed by the system. The child node is the graphical

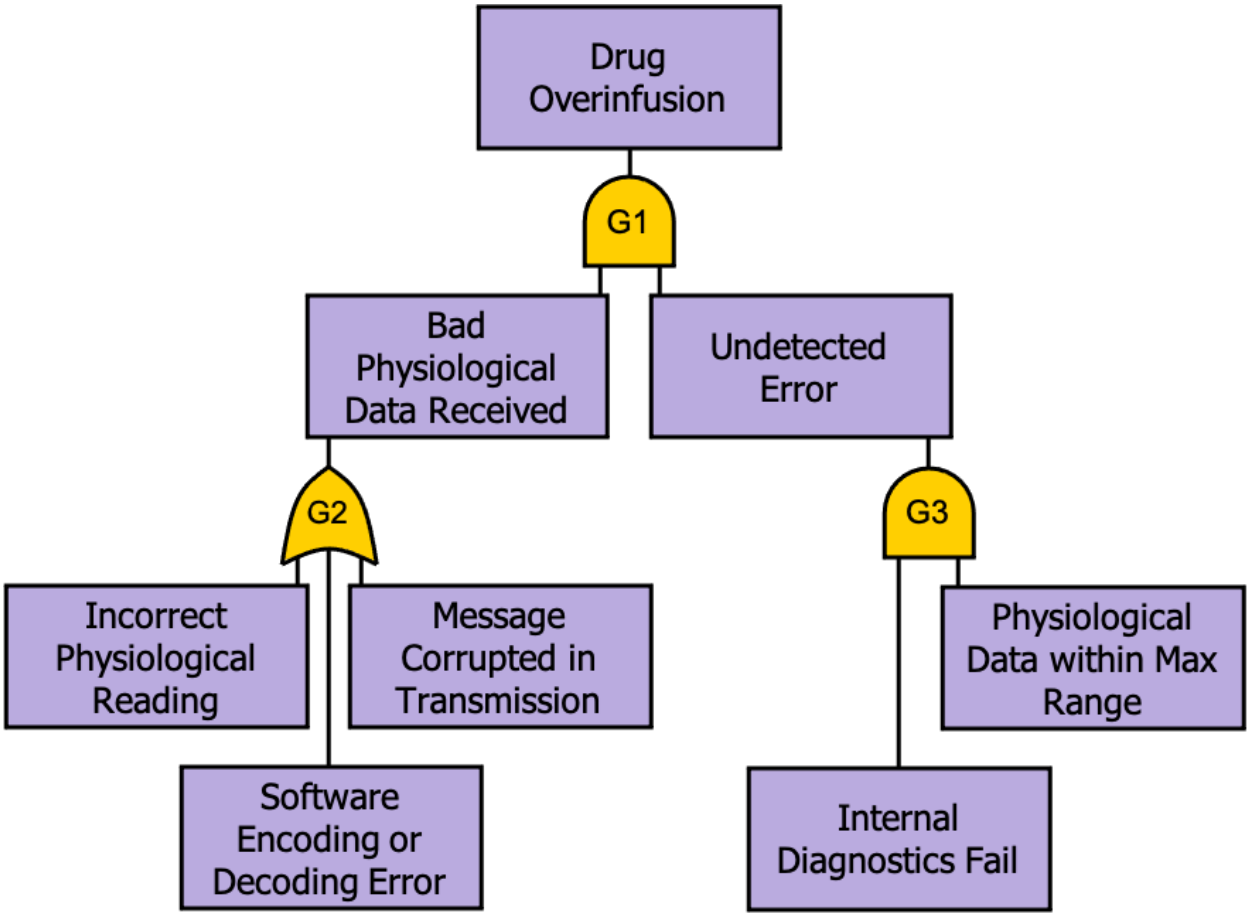


Figure 2.14: FTA analysis of PCA interlock system as demonstrated by Procter *et al.*[40]

symbol for the *And* operator, where its children are the two events 1. system received incorrect physiological data, and 2. An error that is not detected. When 1 & 2 occur at the same time, it can trigger the mishap. The node G2 represents an *Or* operator, where any of its children can cause bad physiological data. Similarly, all the possible causal events and hazards can be explored for an accident.

When an FTA is constructed for a system, it can quickly become huge with many events. An analyst can perform a *cut set* operation to compute a fault path. A fault path is a set of events that together cause the event leading to the mishap. A minimum cut set is the minimum number of events that can cause the hazard, and the causal reasoning literature defines it as a necessary condition(s) for a hazard.

Also, when there are several hazards, eliminating all of the hazards is an expensive task.

Some have a higher chance of occurring, and some are low to none. It is prudent to focus on a hazard that has a higher probability. In an FTA, a hazard probability can be in several ways. The most common approaches are the following:

- Direct analytical calculation using the *cut sets*
- Bottom-up gate-to-gate calculation
- Simulation

Computing the probability of events helps identify weak links in the system design and develop a cost-effective mitigation strategy. A qualitative approach can be taken when the quantitative probability is expensive or unavailable(failure rate of devices).

2.8.2 Failure Mode Effect Analysis (FMEA)

FMEA is a bottom-up, detailed design analysis technique used as part of several safety standards. For each component, analysis computes the effect of failure modes and, based on that, determines if the system requires any design changes. The decision is based on the unaccepted level of safety or reliability resulting from potential failure modes. Although FMEA can be applied at any system level, it is usually applied at the unit level because failure rates are available. With the probability of failure modes added, a component or subsystem failure rate can be computed. It is very effective in computing the reliability of the system. However, it is limited concerning safety purposes as FMEA fails to consider the combination of failure mode.

Larson *et al.*[41] performed FMEA on an AADL model using EMv2 specifications for an ISOLETTE incubator system. They identified the following five tasks in performing FMEA on a system.

Step 1: Identify potential hazards and the components involved in causing a failure

Step 2: For each of the components identified in **Step 1**, list the ways a component can fail and label them as failure modes of that component

Step 3: Compute the immediate effects of each failure mode observed at the successor component or the system boundary.

Step 4: For reliability calculations, compute the probability of failure of each component

Step 5: Optionally casual factors, failure detection and control techniques, residual hazards can be listed in the FMEA table.

The OSATE Integrated Development Environment (IDE) for AADL and EMv2 provides an automated mechanism to auto-generate the FMEA report with detailed EMv2 annotations. Table 2.1 is an auto-generated FMEA report for the PCA interlock model.

Component	Initial Failure Mode	1st Level Effect	Failure Mode	second Level Effect	Failure Mode	third Level Effect	Failure Mode
pulseOx	error event InternalFailure	{NoSpO2} SpO2 ->appLogic:SpO2	appLogic {NoSpO2} [Unhandled Failure Effect]				
pulseOx	{SpO2ValueHigh}	{SpO2ValueHigh} SpO2 ->appLogic:SpO2	appLogic {SpO2ValueHigh}	{TicketTooLong} CmdPumpNorm ->pcaPump:TicketInput	pcaPump {TicketTooLong}	{TooMuchAnalgesic} DrugFlow ->patient:vein	patient {TooMuchAnalgesic} [Masked]
pulseOx	{SpO2ValueLow}	{SpO2ValueLow} SpO2 ->appLogic:SpO2	appLogic {SpO2ValueLow} [Unhandled Failure Effect]				
appLogic	error event SoftwareFailure	{TicketTooLong} CmdPumpNorm ->pcaPump:TicketInput	pcaPump {TicketTooLong}	{TooMuchAnalgesic} DrugFlow ->patient:vein	patient {TooMuchAnalgesic} [Masked]		
appLogic	error event SoftwareFailure	{EarlyTicket} CmdPumpNorm ->pcaPump:TicketInput	pcaPump {EarlyTicket}	{TooMuchAnalgesic} DrugFlow ->patient:vein	patient {TooMuchAnalgesic} [Masked]		
appLogic	error event SoftwareFailure	{LateTicket} CmdPumpNorm ->pcaPump:TicketInput	pcaPump {LateTicket} [Masked]				

Table 2.1: Generated FMEA report for the PCA model

2.8.3 System Theoretic Process Analysis (STPA)

STPA is a hazard analysis technique to support Systems Theoretic Accident Model and Processes (STAMP) causality model[14]. STAMP and STPA differ from other hazard analyses by their use of systems theory and emphasis on the control loop as the artifact of analysis. STPA focuses on avoiding unsafe control actions by systematically identifying and mitigating their causes. The STPA analysis consists of the following three steps:

Step 0: Establish fundamentals for the analysis

- Define accident levels and accidents for the system
- Identify hazards (System state or set of condition + worst case environmental conditions)
- Rewrite hazards as safety constraints on the system design
- Develop high-level safety control structure

Step 1: Identify potentially unsafe control actions

Step 2: Determine how each potentially unsafe control actions could occur

The Figure 2.15 illustrates the safety control structure for the PCA Pump Interlock system described in 2.4.4. In this control structure, the unsafe control action is *pump command* from the controller. As part of the **Step 0**, there are two hazards identified.

1. Infusing drug when patient's health is deteriorating
2. Patient is uncomfortable due to under dosage

The first hazard is possible when the pump continues to infuse the drug when deteriorating the patient's health. The second hazard is caused when the pump fails to infuse the drug. In the **Step 1**, the task is to determine the potential for inadequate control action or control action that leads to a hazard. Similar to other hazard analysis techniques, STPA provides four *guidewords*. For each control action, with the help of the *guidewords*, check

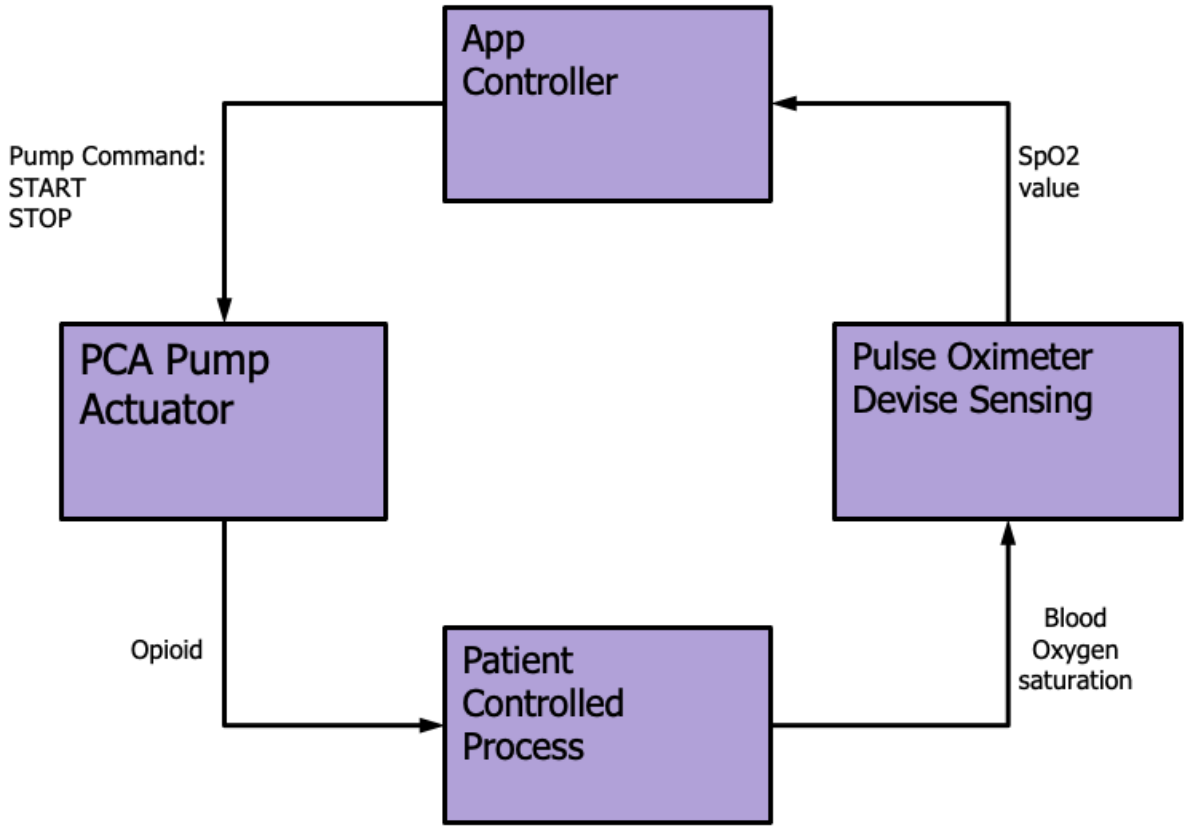


Figure 2.15: PCA Pump Interlock control structure

Control Action	Not Providing Causes Hazard	Providing Causes Hazard	Wrong Timing or Order Causes Hazard	Stopped Too Soon or Applied Too Long
Pump Command START	Patient is uncomfortable due to under dosage	Infusing drug when patient's health is deteriorating		
Pump Command STOP	Infusing drug when patient's health is deteriorating	Patient is uncomfortable due to under dosage		

Table 2.2: PCA Pump Interlock STPA Step 1

if a hazard is caused. Table 2.2 shows the result of performing **Step 1** on the PCA Pump Interlock system from 2.4.4. In the *START* command is provided when it is not supposed to be provided, there is a possibility patient activates the bolus when the patient's health is deteriorating. Similarly, if the *START* is not provided, the pump will fail to respect the bolus commands and does not infuse the drug leading to patient being in pain.

In **Step 2**, for each of the hazardous control actions identified in **Step 1**, the control structure is examined to identify if they cause or contribute to the hazardous control action. Figure 2.16 illustrates the STPA template on the control loop to identify the causal factors

from the control structure and develop a causal scenario for each hazardous control action.

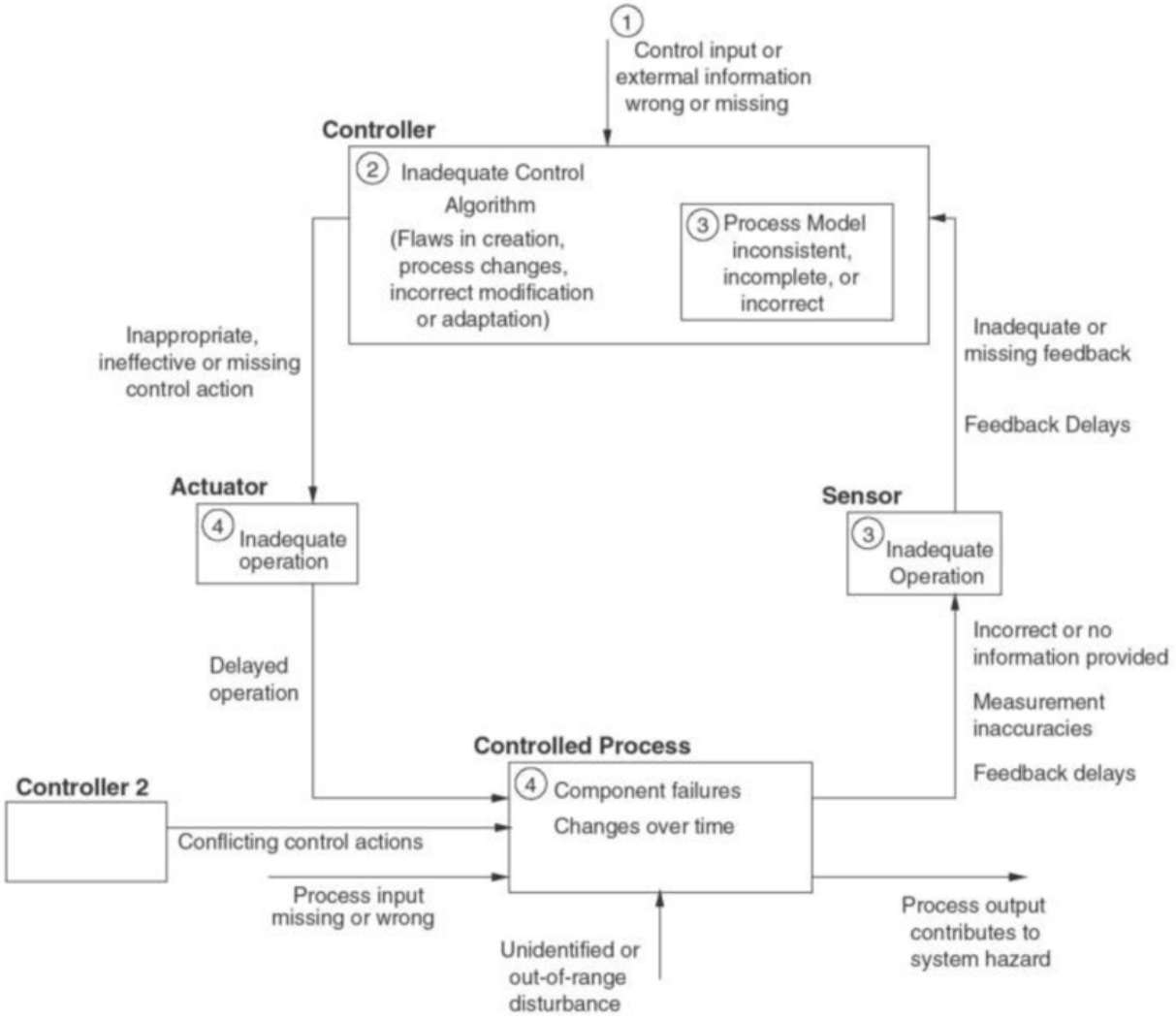


Figure 2.16: STPA Step 2 causality guidwords

With the help of the template, Figure 2.17 shows the causal scenario developed of the hazardous control action of providing the *START* command. This shows how incorrect sensing of the patient's SpO2 value could lead to the hazard of over-infusing the drug.

Overall, STPA answers the following questions:

- List of safety constraints in place to prevent an accident from occurring
- How does a component in a specific role (sensor, actuator, controller, and controlled process) contribute to harming the controlled process?

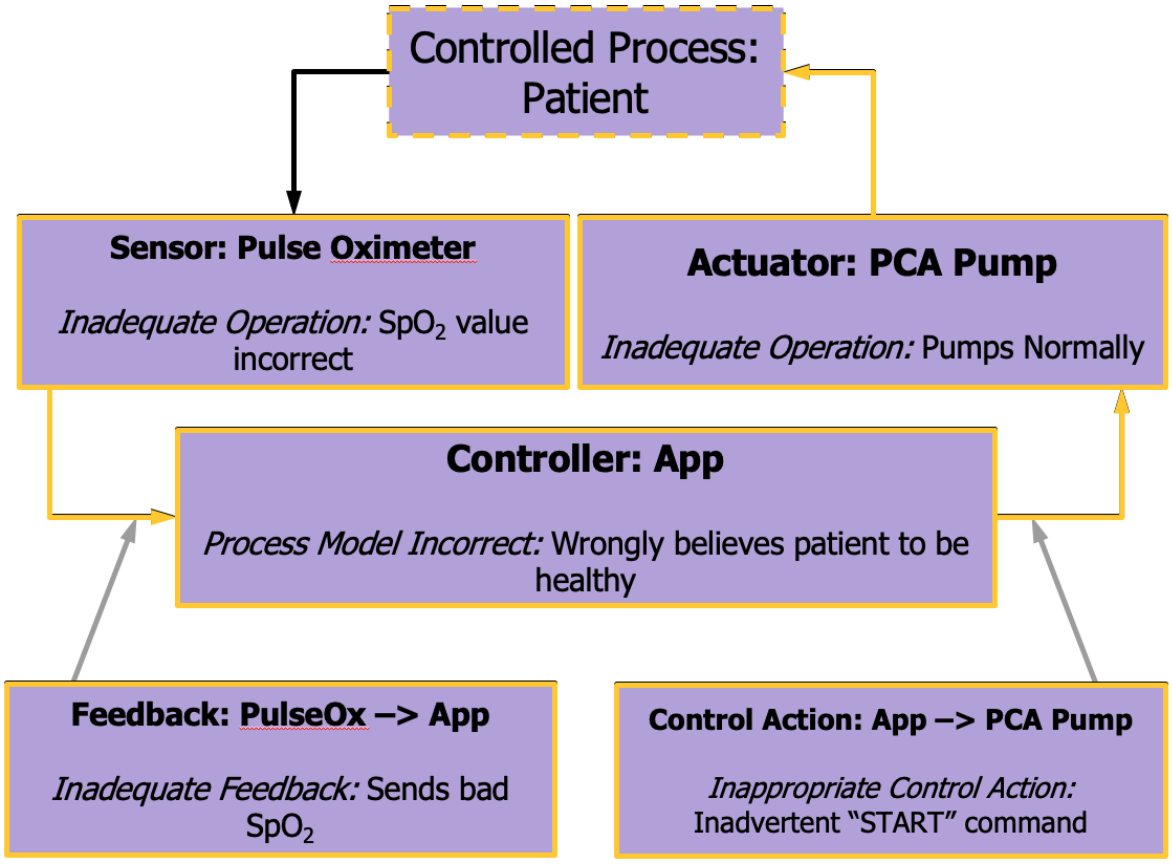


Figure 2.17: Causal scenario for inadvertently providing *START* command

- How are incoming commands handled by components and fed back to the controller?

Challenges

In my experience, the following are some of the challenges I faced in applying STPA to the PCA Pump interlock system.

- Control structure is not system model: The control structure is manually derived from the system architecture. It is often not straightforward to derive a faithful and analyzable control structure to identify hazardous control actions and causal scenarios. Maintaining the consistency between the two artifacts is error-prone.
- The steps in STPA are informal and imprecise *e.g.* types of faults and their interpretation, and causal reasoning.

- In large, STPA analysis is manual, and tooling support is limited

2.9 Interconnected Systems

One of the capabilities of interoperable medical devices is composing devices and apps into a system at point-of-care. In this system, devices and apps communicate through a communication substrate or middleware.

2.9.1 Communication Paradigms

The communication substrate follows a predefined protocol to facilitate the communication between the components. There are two communication paradigms:

1. **Shared memory:** Components communicate by accessing shared memory. This shared memory should be protected to give exclusive access to the component performing a write operation. Mutexes, semaphores, and monitors are some of the commonly used mechanisms for achieving exclusive resource access. The shared memory paradigm is fast but difficult to implement and analyze safety properties.

2. **Message passing:** The messages are sent and received between components. This method is generally slower than shared memory, but it is versatile to support complex situations.

Ranganath *et al.*[42] provide the following communication patterns to enable the interconnection of medical devices.

- Publisher-Subscriber
- Requester-Responder
- Sender-Receiver
- Initiator-Executor

- Orchestration

The quality of service required by the system can be captured by defining constraints over a small set of properties irrespective of the communication pattern used. Those identified properties are:

- *Minimum Separation:* It is the minimum required duration between providing or receiving two consecutive messages.
- *Maximum Latency:* It is the maximum duration below which the network facilitates the communication between two components
- *Minimum Remaining Lifetime:* The duration between the time of the creation of a message to the time in which the message goes stale

2.9.2 *AAMI/ANSI/IEC TIR80001*

In the past decade, the health IT arena changed significantly from monolithic medical devices to sophisticated interconnected medical devices. This shift in the development practice leads to the existing risk management process (ISO 14971) being inadequate due to the emerging new hazards related to the interaction of networked components. The response from the regulatory community is the IEC 80001, which is an extension of ISO 14971. IEC 80001 enables the risk management process for medical devices exchanging information over an IT network. The key enhancement in the risk analysis of IEC 80001 over the ISO 14971 is the redefinition of the term *Harm* from the safety domain to *Unintended consequence* to cover safety, effectiveness, and data & system security domain.

The ten step risk analysis process discussed in the *AAMI/ANSI/IEC TIR80001* [43] as follows:

Step 1: Identify Hazards

Step 2: Identify causes and resulting hazardous situations

Step 3: Determine unintended consequences and estimate potential severities

Step 4: Estimate the probability of the unintended consequence

Step 5: Evaluate risk against pre-determined risk acceptability criteria

Step 6: Identify and document proposed risk control measures and re-evaluate risk

Step 7: Implement risk control measures

Step 8: Verify risk control measures

Step 9: Evaluate any new risks arising from risk control

Step 10: Evaluate and report overall residual risk

2.10 Security

Over the last decade, there has been a series of high-profile vulnerability disclosures for safety-critical devices. Such devices are carefully designed for safety yet evidently do not adequately handle safety issues that arise from security problems. This section will discuss the Dolev-Yao network adversary model, Multiple Independent Levels of Security/safety (MILS) based system design and security policy, and AADL's security annex to model them.

2.10.1 Dolev-Yao Network Adversary Model

Dolev and Yao[44]developed several formal security models for evaluating the security for a communication protocol among two parties using public-key encryption. In the context of interoperable medical devices, we apply *Dolev and Yao* model for communications between any two pair of components in the system. In this model, we consider the presence of a malicious device or any device compromised by a saboteur to perform malicious activity. In this model, an adversary has the following capabilities:

- Can obtain any message passing through the network

- Can initiate a conversation with other devices
- Can receive messages from other devices regardless of the intended recipient

2.10.2 MILS

MILS[45] is a security system design approach that advocates decomposition of system to identify security critical components that are small and simple in both functionality and security policy. The identified components are implemented as physically distinct subsystem through the use of a separation kernel that support the following four policy[46] :

1. Information flow policy
2. Data isolation policy
3. Fault isolation policy
4. Periods processing

The security critical components are developed with strong assurance arguments to deem them as trusted components. However, the system's functionality is achieved by the untrusted components, while the trusted components enforce security by protecting the interfaces to untrusted components and by mediating the communications between the trusted and untrusted components.

The modern MILS approach supports systems of systems and concurrent systems where the assurance case for the system is derived from the individual components, thus encouraging the development of COTS MILS components.

Only the information flow policy is enforced at the middleware layer when trusted components are implemented in a separate processor or a separation kernel partition. While the other three policies are handled by the separation kernel or not a concern in a distributed system with multiple processors.

The information flow policy is enforced by labeling each component with a unique security classification. These labels are used to authorize communication between components[46], and

the ordering on the labels dictates the permissive flow of information from one domain to another.

There are two layers to security policy assurance[47]:

- **Local policy assurance:** The implementation of cryptography algorithm, filtering, and bypass are assured by formal methods and rigorous testing
- **Integration policy assurance:** Composition of local policies along with the untrusted components are assured by the separation kernel configuration and assurance of information flow policies

The security research community and Department of Defense (DoD) have long employed Bell-LaPadula and Biba information flow policy for confidentiality and integrity. Biba model is the direct inverse of Bell-LaPadula in the permitted flow of information. If L1 $\leqslant$ L2, then the Bell-LaPadula model allows the flow of information from L1 to L2 but not from L2 to L1. Similarly, in the Biba model, L2 to L1 is permitted but not in the other way.

2.10.3 AADL Security Annex

AADL security annex provides guidelines and properties to model and analyzes security properties in a system. The annex provides properties to model, data encryption, authentication, security classification labels, and extendable keys and certificate. The security annex also provides guidelines to associate the security properties with modeling elements.

```
(TopSecret, Secret, Confidential, Unclassified);
```

Figure 2.18: AADL security annex classification levels

Figure 2.18 show the definition of security classification levels. However, at the time of writing this dissertation, there was no tool to infer classification levels. Therefore to correctly analyze large models, each model element must be associated with the security property and adds considerable cost to the security modeling activity. The security classification levels of security annex does not support multi-lateral security where two or more labels are in

the same security level, and the information flow between them is restricted. The Security Modeling Framework (SMF) discussed in section addresses some of these limitations.

AADL security annex with its comprehensive set properties enables considering security characteristics from the initial design of the system and support security assurance arguments.

Chapter 3

Modeling Critical Systems

This chapter demonstrates modeling of safety and security critical systems using AADL and EMv2. First section 3.1 presents AADL information flow concepts using an unmanned aerial vehicle system. Followed by guidelines for developing an error library using EMv2 for performing risk analysis for MAP Apps. In section 3.4, we demonstrate *AAMI/ANSI/IEC TIR80001* risk analysis process integrated with AADL using a pulse oximeter (sensor) model. Section 3.5 describes the Open PCA pump model as an actuator component. Finally, section 3.6 illustrates the controller app and iterative development process of PCA interlock system.

3.1 Unmanned Aerial System

In this section, we provide a brief overview of AADL, focusing on its various forms of dependency and information flow relations. AADL concepts are illustrated using a simple Unmanned Aerial System (UAS). The simple UAS is adapted from an example used by the Collins Aerospace team on DARPA Cyber-Assured Systems Engineering (CASE) project.

Figure 3.1 presents a high-level view of an example system fragment (upper left) along with excerpts of AADL modeling artifacts. The system concept is based on a Unmanned Air Vehicle (UAV) for conducting surveillance. The UAV receives mission information (e.g., a map with a set of targets) and sends status information from/to a ground station. The

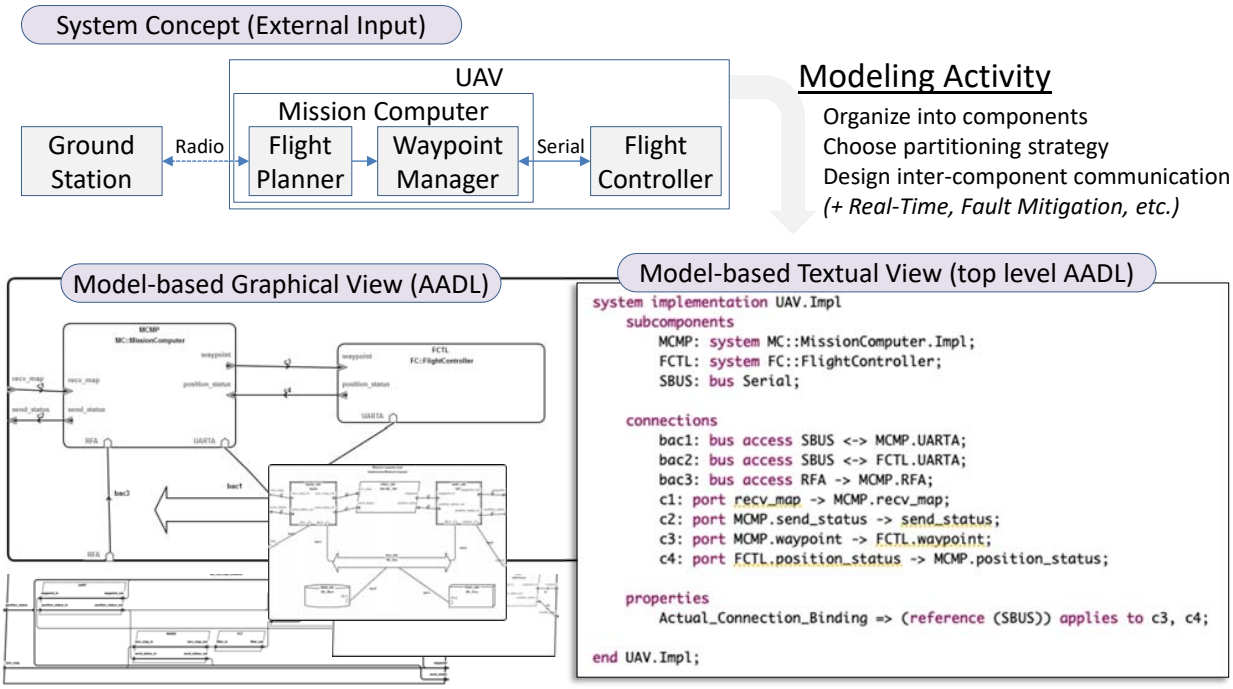


Figure 3.1: A Simple UAS Example with AADL Modeling Artifacts

UAV system includes a mission computer and a flight controller components. The mission computer calculates waypoints from ground station inputs. The flight controller acts upon these waypoints by setting aircraft controls to fly in a manner that will advance the surveillance task. The mission computer functionality is inspired by US Air Force Research Lab's Unmanned Systems Autonomy Services (OpenUxAS)[48].

Inter-component dependencies: The most prominent inter-component relationships are *connections*, which capture data and control flows between software components such as threads and processes (e.g., the flow of waypoint data between the mission computer and flight controller). Connections associate *ports* on sending and receiving components. *Ports* are the interaction points of a component to model directional transfer of data and control. AADL includes different port categories to specify communication patterns between components (e.g., asynchronous message passing, synchronous shared memory).

- Data port: transfer of data without queuing mechanism
- Event port: sends and receives signals using a queue

```
system implementation UAS.Impl
 subcomponents
  GND: device GS::GroundStation;
  UAV: system UAV::UAV.Impl;
  RFB: bus RF;
 connections
  c1: port GND.send_map -> UAV.recv_map;
  c2: port UAV.send_status -> GND.recv_status;
  bac1: bus access RFB <-> GND.RFA;
  bac2: bus access RFB <-> UAV.RFA;
 properties
  Actual_Connection_Binding => (reference (RFB)) applies to c1, c2;
end UAS.Impl;
```

Figure 3.2: Simple UAV system top level model – illustrating inter-component dependences

- Event data port: sends and receives both data and events using a queue

In AADL, relationships between middleware components can be captured by specifying connections via *bus accesses* (intuitively, a bus access is a feature on a software component indicating that it utilizes a communication substrate). For example, the mission computer and flight controller declare access to a *bus* that models a serial bus communication medium. Finally, software elements such as threads/processes and connections can be allocated to middleware and hardware resources such as processors and buses using *bindings*. These dependencies can have multiple layers. For example, a process can first be bound to a *virtual processor* used to model a partition in a hypervisor and then the hypervisor partitions can be bound to a *processor*. Similarly, a communication *connection* can be bound (transitively) through *virtual buses* representing layers of abstraction and associated protocols in a protocol stack.

3.1.1 UAS: The top level system with ground station and UAV

Figure 3.2 provides excerpts of the simple UAS AADL model that illustrate some dependencies described above. It captures the implementation of the system consisting of three sub-components of type *GroundStation*, *UAV*, and *RF (radio frequency communication)* with

connections representing information flows between them. These connections include both: (a) port connections representing application level communication at lines 7-8, and (b) bus accesses representing component infrastructure utilization of the underlying communication RFB substrate at lines 9-10. Line 12 is a *binding* specifying that the port communication between the ground station and the UAV components is realized using the RFB bus at a lower level of abstraction. Line 4 instantiates the implementation of *UAV* system illustrated in right-bottom of Figure 3.1.

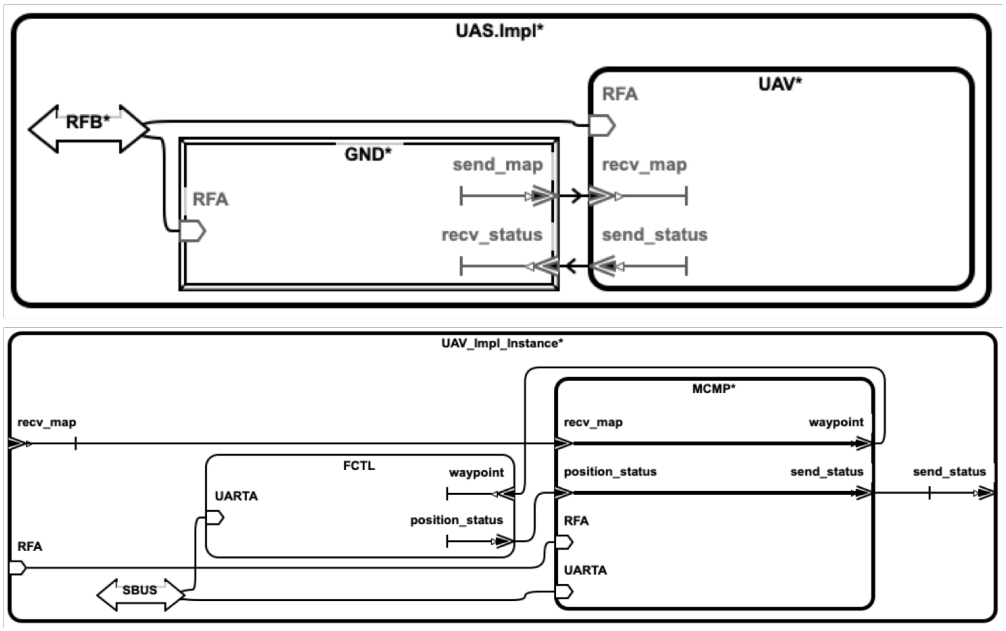


Figure 3.3: Instance diagram of top level system and the UAV subsystem

The model information in Figure 3.2 is part of what AADL terms as a *declarative model* because it declares the architectural structure organized into a hierarchy using various AADL containment structures. In addition, the model may have multiple component implementation declarations for a given component type. Given a selection of particular component implementations, AADL tools will construct an *instance model* which instantiates the declarative model to a particular implementation configuration/instance while removing some of conceptual containment components to more directly associate connections between ports of components corresponding to actual hardware and software units. Figure 3.3 illustrates a portion of an instance model diagram for the system. Note that some of the dependence

relations such as bindings, e.g., the realization of the connection *c1* and *c2* through the bus *RFB*, are present in the instance model, but OSATE does not display them in the diagram view.

Intra-component dependencies: AADL also provides multiple notions of intra-component dependencies. The most basic of these are *flow specifications*, which model data and control flow relationships between a component's input and output ports. Figure 3.6 presents additional model details for the mission computer component. The *flow* annotation at line 10 indicates that computing a waypoint will involve taking map information as input from the component's `recv_map` port and producing waypoints that will flow out of the `waypoint` port. Similarly, computing status information will take input from the `position_status` port and send information out of the `send_status` port (line 11). AADL does not define a precise semantics for flows, and it does not make an explicit distinction between data and control flows. Flow annotations may be given different interpretations by different analysis tools. For example, a latency analysis tool may consider a flow to model a single or a collection of execution paths through the component source code, with an associated worst case execution time for the path. A security analysis tool may interpret the flow as a specification of information flow (e.g., a combination of data and control flow). In this work we interpret the flow annotation as an abstract entity that forms a dependency between input and output ports. This understanding is refined further in the risk analysis where the flows propagates errors between ports.

3.1.2 Security Requirments

For this system, a portion of the system requirements will state security properties related to data manipulated by the system. Here are two such requirements, followed by rationale.

Req-1: All communications from the Ground station to the UAV must be authenticated

Req-2: The Commands from the Ground station shall be checked for well-formedness

For the sake of simplicity this model excludes access control policies intentionally and we assume all authenticated commands from the Ground station shall be authorized to control

the UAV. The details of the notion of "well-formedness" are irrelevant for our discussion here. A command is "well-formedness" when it conforms to a valid command data format and the command does not cause unsafe maneuvers that leads to damage to the UAV or direct the UAV into any "keep out" zones defined in the map. The rationale for these requirements is as follows. In situations where an adversary poses as a legitimate Ground Station, the adversary may send malicious messages to the UAV designed to exploit some control-flow/memory-safety/algorithmic-complexity flaw in the code that parses and processes the UAVs flight plan and map data that causes that function to crash or wedge, leading to a (possibly brief) period where UAV will not accept new commands. The adversary repeats this message as often as needed to cause prolonged DoS. This leads to the UAV being unable to receive new flight plans and failing to complete the surveillance mission, since the ground station cannot direct UAV to follow convoy turns or surveil new areas.

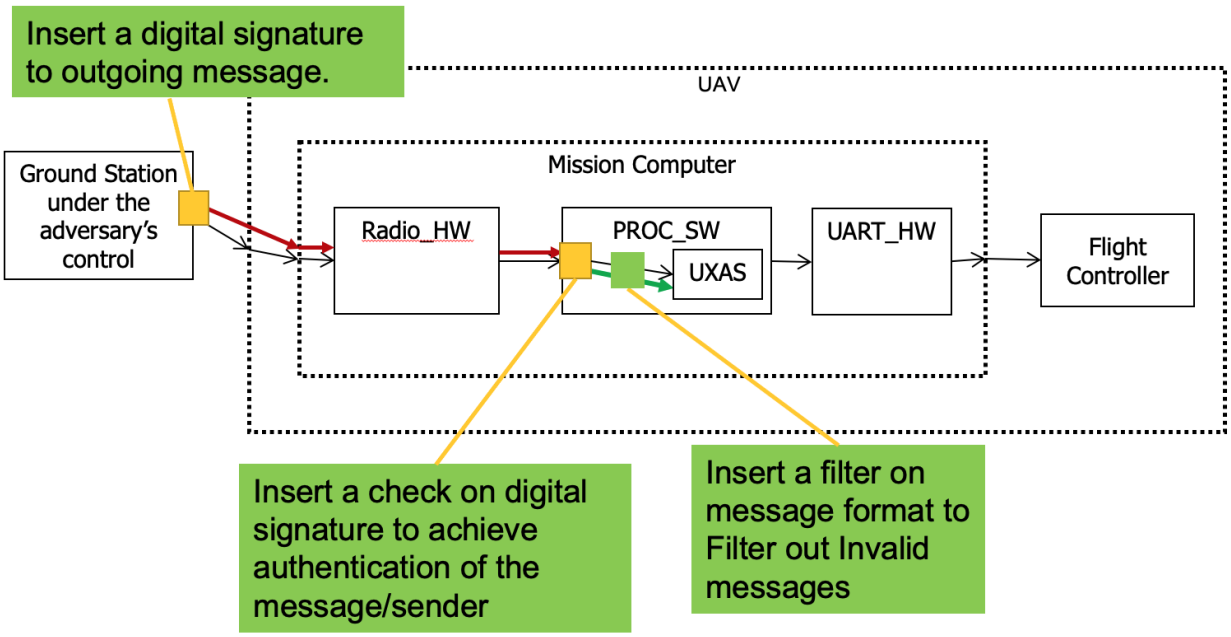


Figure 3.4: Mitigation concept to satisfy Req-1 and Req-2

The system architecture may be modified to ensure that the requirements are satisfied. For instance, Req-1 indicates that an authentication check should be performed in the Radio Driver software. The messages failing the authentication checks are discarded. Even if the adversary acquires the authentication keys, Req-2 indicates that messages from the Ground

```
1  process MC_SW
2    features
3      recv_map: in event data port Command.Impl;
4      send_status: out event data port Coordinate.Impl;
5      waypoint: out event data port MissionWindow.Impl;
6      position_status: in event data port Coordinate.Impl;
7    flows
8      compute_waypoint_flight_plan: flow path recv_map -> waypoint;
9      compute_waypoint_pos_status :flow path position_status -> waypoint;
10     compute_status : flow path position_status -> send_status;
11 end MC_SW;
12
13 process implementation MC_SW.Impl
14   subcomponents
15     RADIO: thread RadioDriver;
16     FLT: thread Filter;
17     FPLN: thread FlightPlanner;
18     WPM: thread WaypointManager;
19     UART: thread UARTDriver;
20   connections
21     c1: port recv_map -> RADIO.recv_map_in;
22     c2: port RADIO.send_status_out -> send_status;
23     c3: port RADIO.recv_map_out -> FLT.filter_in;
24     c4: port FLT.filter_out -> FPLN.recv_map;
25     c5: port FPLN.flight_plan -> WPM.flight_plan;
26     c6: port WPM.waypoint -> UART.waypoint_in;
27     c7: port UART.position_status_out -> WPM.position_status;
28     c8: port UART.position_status_out -> FPLN.position_status;
29     c9: port UART.position_status_out -> RADIO.send_status_in;
30     c10: port UART.waypoint_out -> waypoint;
31     c11: port position_status -> UART.position_status_in;
32 end MC_SW.Impl;
```

Figure 3.5: Software Sub-system

Station pass through a filtering component inserted between the radio driver and flight planner. The filter component performs the "well-formed" check to stop malicious messages reaching flight planner. Figure 3.4 illustrates the implementation changes to satisfy the requirements.

Figure 3.5 describes the type and the implementation of the modified software sub-system of the UAV. Line 16 indicates the inclusion of the filter(FLT) component and the connection

in lines 21-24 describes the messages received by the radio device passes through the radio driver and the filter component before reaching the flight planner. When a system build is created from the model, the underlying separation kernel plays a key assurance role by guaranteeing that there are no other paths for ground station data to flow through the flight planner beyond explicitly-specified path that goes through filter component.

AADL includes other notions of flows that augment the basic flows above. For example, it provides an Error Modeling (EMv2) annex to support multiple forms of model-based hazard analysis. In EMv2, tokens representing errors/faults and error flow annotations are added to model propagation of errors through a component. One can also use these annotations to model various types of security issues. To reason about the above mentioned requirements, one can use AADL error tokens to simulate/approximate a *cartesian abstract interpretation* that captures different combinations of well-formedness and authentication properties. We define the following four tokens EMv2 error tokens: *wellformed_authenticated, wellformed_unauthenticated, not_wellformed_authenticated,* and *not_wellformed_unauthenticated*[1].

The system developer may use AADL EMv2 annotations to model the intent of the added security control. In Figure 3.6, lines 15-16 models that the mission computer component may receive any combination of error through its port *recv_map*. However, it may propagate only the *wellformed_authenticated* message to the flight controller – reflecting the fact that somewhere in the mission computer architecture security functions "filter out" malformed messages and unauthenticated messages. Line 18 shows the propagation of the token *wellformed_authenticated* received in the input port *recv_map* to the *waypoint* port (i.e., "good" information is allowed to flow through and form the basis of waypoints to the flight controller). On the other hand, the mission computer acts as a *sink* for (i.e., filters out) any token that indicates a "bad" message (i.e., tokens *not_wellformed_unauthenticated*, *not_wellformed_authenticated*, and *wellformed_unauthenticated*).

[1]Future enhancement of AADL EMv2 might allow those notions to be captured as pairs {*wellformed, authenticated*} and the tool calculates Cartesian products to form underlying domains.

```
system MissionComputer
 features
  recv_map: in event data port DataType.Impl;
  position_status: in event data port DataType.Impl;
  waypoint: out event data port DataType.Impl;
  send_status: out event data port DataType.Impl;
  UARTA: requires bus access UAV::Serial;
  RFA: requires bus access UAS::RF;
 flows
  compute_waypoint: flow path recv_map -> waypoint;
  compute_status: flow path position_status -> send_status;
 annex EMV2 {**
  use types UAS_Errors;
  error propagations
   recv_map : in propagation {wellformed_authenticated,
      wellformed_unauthenticated, not_wellformed_authenticated,
      not_wellformed_unauthenticated};
   waypoint : out propagation {wellformed_authenticated};
   flows
    wellformed_authenticated : error path
       recv_map{wellformed_authenticated} ->
       waypoint{wellformed_authenticated};
    unauthenticated_map : error sink
       recv_map{not_wellformed_unauthenticated,
       wellformed_unauthenticated};
    not_wellformed_map : error sink
       recv_map{not_wellformed_unauthenticated,
       not_wellformed_authenticated};
  end propagations; **};
end MissionComputer;
```

Figure 3.6: AADL Flow and Error Propagations Annotations in Mission Computer

3.2 Modeling Error Library

As demonstrated in the previous section 3.1.2 developing an error library is useful in capturing high level properties on the model. However, the intended purpose of the error library is to support hazard analysis and estimate risk level of the system. The error library plays a vital role in distributed development where a consortium or platform developer provides a *Error Library*. This error library provides a common error type hierarchy that can be

extended or adapted with context-specific error types by device manufacturers over the device interfaces. Additionally, a system integrator can use the same library for performing system-wide risk management. Overall, error library serves as a communication vocabulary among stakeholder in identifying the propagation of errors in the system.

3.2.1 Error Library

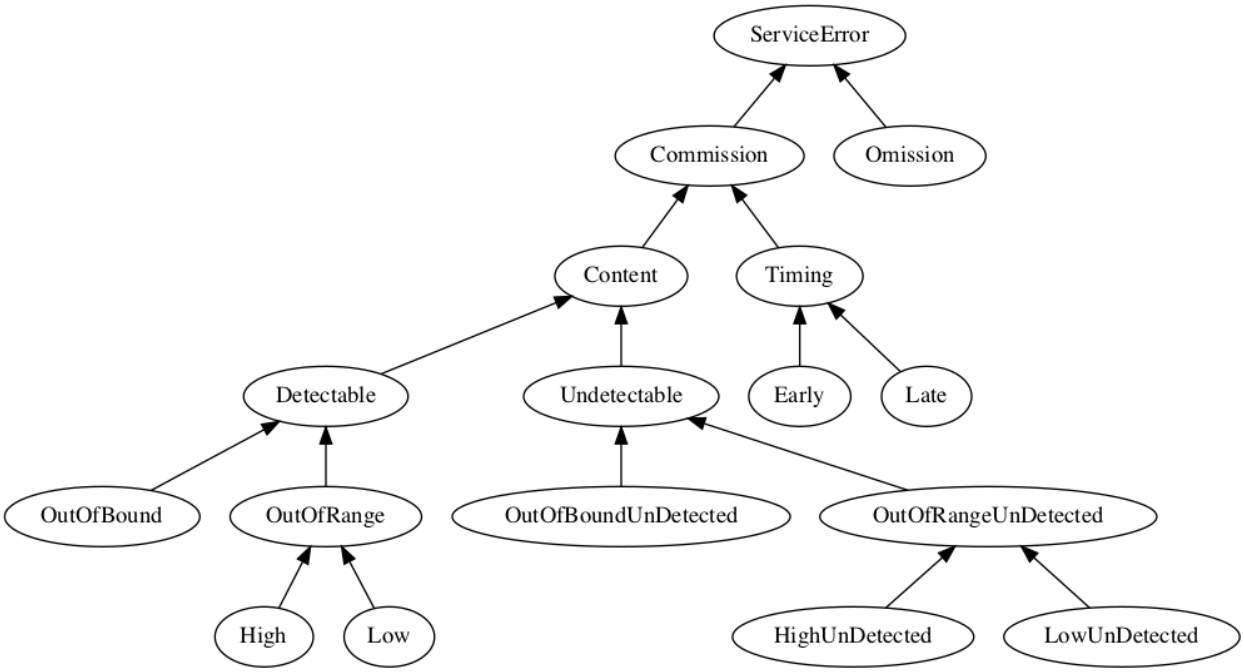


Figure 3.7: Error Library

To illustrate this concept, we provide a simplified hierarchical error library structure provided by the MAP and our interpretation of each error type. Defining the error types hierarchically serves the following two purposes:

- Reduces the amount of effort in modeling the system. A collection of errors can be replaced by an element representing the super set of errors
- Mitigating each error is tedious, it is easier to tackle a class of error types together

Figure 3.7 represents the hierarchical structure of the *simplified error library* which illustrates the error library provided by the EMv2 in unified tree structure. Our interpretations

of the error types are as follows:

- *Service Error:* Root node in our hierarchy, used to represent erroneous service but unsure of exact error in the service. Conservatively, the analysis assumes all possible leaf level error types

- *Omission:* This error type represents a situation where there is a lack of service when expecting a service. It can also be interpreted as service item that is never delivered

- *Commission:* The delivered service is either incorrect or unexpected. The child types of this error type are specific to a property domain that this service violates

- *Content:* The content domain of the service is incorrect

- *Detectable:* Content of a service violates a specification or a property *e.g.* outside an expected range

- *UnDetectable:* There are situations, where a property may not be violated, and yet the content is incorrect. For instance, the value produced by a sensor may be within the specified limit. However, the sensed value may be an inaccurate measurement of the sensed phenomenon. For example, a patient's actual pulse rate is 70 bpm and the value provided by a pulse oximeter is 85 bmp with error $\pm$ 3

  We distinguish *Detectable* and *UnDetectable* content error, to inform an App developer to develop the software with appropriate constraints and to suggest them to consider redundancy.

- *OutOfBound:* This error type represents the content that does not satisfy a property over an enumerated type. For example, content not belonging to the expected set of commands. We should also use this error type to capture any unsatisfied property that is not numeric in nature. For example, App with email field requiring the input to satisfy email address format

- *OutOfRange:* Content that is outside of the specified range. *High*, represents the content of a service above the specified range, where else *Low* represents the content that is below the specified range

- *Timing:* The timing domain of the service is incorrect with subtypes *Early* when the service is delivered before is it expected and *Late* when the service is stale

We designed the error library in figure 3.7 instead of EMv2 error library to have a reduced number of error types that sufficiently captures the error behavior. Although, we would like to build a lattice structure, the EMv2 language does not allow such a structure. A lattice is useful to capture a combination of errors such as *LateAndHigh*, where the parents of this error would be *OutOfRange* and *Timing*. EMv2 provides the * (star) operator to dynamically form such combinations of error. According to Bondavalli *et al.*[49], we can assign a "simplest" or "possible" domain for errors with more than one domain. For example, the detection mechanism for the message with errors (Late * OutOfRange) can be detected and mitigated by just the timing error. If such a mitigation or detection mechanism is not possible, extend the common parent error type with new type representing both the domains.

3.2.2 Guidelines for developing device specific error library

The following are the general guidelines for developing error library for a device:

- For each output port in the device, identify the corresponding error type by asking the following questions:
  - What is the type of the port? Data, event, or both?
  - In the case of a data port, what is the data domain? Consider the possibility of *Content* error in both detectable and undetectable branches of error types
  - In the case of an event port, investigate the type of event, such as synchronous or asynchronous event. In a synchronous event, what is the rate of transmission? Consider the possibility of *Early* or *Late* error types

  - In the case of event data port, consider assigning a "simplest" or "possible" domain in which the error can be detected and mitigated. If not, extend the *ServiceError* with new error type to represent the domain combination
  - Will the device be part of a network? consider the possibility of a delayed message, no message, incorrect/corrupted message, or inadvertently leaking the message (implicit leak)
  - What will be the security measure in the network? Is there a possibility of receiving a message from an untrusted source? Consider corrupted or delayed message

- Consider renaming the error types for better indication of the erroneous situation
- Refine the abstract error token, such that each refined token leads a different hazardous situation or the lack of one.
- Limit the creation and refinement of error types to attain an efficient analysis
- Develop different error libraries to capture erroneous situations in physical features. For example, TubeLeakage, DrugContamination, etc.

3.2.3 Effect of violation of communication properties mapped to error library

The error library defined in section 3.2 captures the violation of communication QoS properties of *Minimum Separation*, *Maximum Latency* and *Minimum Remaining Lifetime* defined at 2.9.1.

If a component provides or receives messages less than *minimum separation*, the effect can be captured by *Early* error token. Similarly, when the latency of the network or the communication component exceeds the *Maximum Latency*, the service would be delayed therefore, the error associated is *Late*. Additionally, a message goes stale when the communication component fails to deliver the message within the expected lifetime of the message. The effect of this failure directly corresponds with *Late* error token. In case of entire network

failure, we can use error type *Service Omission* where the message is not delivered. Table 3.1 captures this mapping from violation of communication property to *simplified Error Library*

Table 3.1: Communication Errors

Communication Property	**Error Type**
Minimum Separation	Early or Service Commission
Maximum Latency	Late or Service Omission
Minimum Remaining Lifetime	Late or Service Omission

3.2.4 Effect of violation of security properties mapped to error Library

The Dolev-Yao adversary model essentially replaced the network connecting system components[50]. This adversary is restricted from knowing information secret to the components, such as shared passwords or keys, but otherwise has unrestricted access to the network. When examining this attacker from a message-centric point of view, we derive the following list (many items are mutually exclusive):

1. The message is *wrong* in a non-obvious way, e.g. the contained data value is within acceptable range, the message is *not* replayed – is fresh – and the message is correctly authenticated. This situation is not not explicitly part of the Dolev-Yao model, and could arise in the infinitesimally unlikely situation wherein the adversary has successfully brute-forced a secret key of a legitimate component within the system.

2. The message is corrupted in an obvious way, e.g. data out of range or incorrectly authenticated. This maps to the Dolev-Yao "corrupt" capability (alternatively, it may be the result of unsuccessfully trying to forge a message).

3. The message is a duplicate according to its sequence number, mapping to Dolev-Yao "replay".

4. The message is late according to its timestamp, mapped to the "delay" capability in Dolev-Yao.

Dolev-Yao property	Error types
Read	NoError
Delay	Late
Modify	Content
Fabricate & Send	Content / Early

Table 3.2: Violation of security property captured as basic error types[50]

5. The message is missing according to the receiver's expectation, mapped to Dolev-Yao "drop". It is also equivalent to a message being delayed past its usability threshold.

6. The message is early according to its timestamp or the receiver's expectation. This is not an explicit Dolev-Yao capability, but results from either 1 or an unlikely combination of clock skew of the source component combined with 5 and 3.

These capabilities allow an adversary to perform the following action. Read, modify, delay or block, and replay or produce a message from previously recorded communication. We can capture the effect of these actions in terms of the error types defined in section 3.2.1. Procter *et al.*[51] provided a direct mapping from violation of security property to EMv2 error types.

Table 3.2, provides the mapping between the violation of security properties to error types in *Error Library*. As part of risk analysis, the device manufacturer would consider how improper interactions over the network interface could cause safety problems. Using AADL error modeling framework such causes would be represented as error types flowing in to the AADL ports corresponding to the network interface.

3.3 Application

In this section we will walk through the process of developing an error library from the perspective of a device manufacturer with the possibility of the device being used in an inter-operable system.

As a device manufacturer, the context of use of the device in a system is unknown,

therefore the error library has to be conservative to cover all possible use-cases.

3.3.1 Pulse Oximeter (PulseOX) - Sensor

As mentioned earlier, one of the tasks for the device manufacturer is to provide *ISO 14971* risk management file. Thus, we assume that a risk management file provides a set of errors for each external interface of the device.

A sensor is a device that senses a physical phenomenon from the environment. It's input interfaces are the entry point into the system. In the PCA shutoff system, there are three sensors,a pulse oximeter(PulseOX), a capnography, and a electrocardiogram(EKG).

In this section, we discuss the development of error type related to PulseOx. The PulseOx device emits infra-red light typically though patient's finger and measures the amount of light absorption. The The amount of light absorption corresponds to the amount of oxygen in the blood. Pulse oximeter performs a complex calculation to convert the sensed light to Oxygen saturation and pulse rate. For the sake of simplicity, we can assume that a pulse oximeter senses two physical phenomenon and provides their real time value as digital signals.

Out of the two output ports let us first consider the SpO2 port with the following specification, using the guidelines identify the properties of the port such as

SpO_2 Accuracy: Adult - Neonate	
Saturation (% SpO_2 ± 1 SD)	
70% to 100% ± 2 digits; ± 3 digits (Motion)	
60% to 80% ± 3 digits	
Low perfusion	70% to 100% ± 2 digits
Pulse Rate	20 to 250 bpm ± 3 digits
Low perfusion	20 to 250 bpm ± 3 digits
Alarms	Adjustable Alarm Limits SpO_2 high, SpO_2 low, Pulse Rate high, Pulse Rate low
Sat Sec Range	10, 25, 50, 100

Figure 3.8: Pulse Oximeter Specification

- *Port*: SpO2

- *Kind*: Data
- *Data Type*: Numeric
- *Specification*: 60 - 100 range with ± 3 digits

Furthermore, based on the error library defined in section 3.2.1, we consider the possibility of the following errors for port SpO2:

- *SpO2OutOfRange*: This informs that there is a possibility that the device may produce data that are outside of its specifications. If successor devices fails to handle this error appropriately, it may cause faults in successor devices.
  Note: There are variations of Pulse Oximeter devices with additional features. For example, alarming, ability to configure a tighter range, etc. However, in our case, we consider the minimal set of features for demonstration purposes.
- *SpO2High*: This error type informs that there is a possibility of inaccurate sensing may be due to incorrect calibration or other internal faults. The data provided by the PulseOx is above the correct measurement of the phenomenon even after considering the specified error range and still within the device specification. I accurate sensing may lead to poor decisions by the successor device or the clinicians.
- *SpO2Low*: This is similar to *SpO2High* however, the data produced by the device is lower than the correct measurement of the phenomenon.
- *SpO2Early*: Similar to *ContentErrors*, data provided at incorrect timing may get rejected or obstruct the communication system on the platform. If connected directly to display, incorrect timing may misguide the clinicians.
- *SpO2Late*: This is similar to *SpO2Early* yet, without the context of the device use-case, we cannot presume to combine the error types.
- *NoSpO2*: The device may fail or dislodge, in such situations the device fails to provide any data. One can argue that this error is a redundancy of *SpO2Late*. However, the

reason behind this error type is to distinguish between the situations where the device is connected to a display device, and the display device may choose to display stale data with a warning message (captured by *SpO2Late*), or the screen may become blank which is captured using *NoSpO2*.

```
-- PulseOx manifestations
 SpO2OutOfRange renames type MAP_Errors::OutOfRange;
 SpO2ValueHigh renames type MAP_Errors::HighUnDetected;
 SpO2ValueLow renames type MAP_Errors::LowUnDetected;
 SpO2Early renames  type MAP_Errors::Early;
 SpO2Late renames  type MAP_Errors::Late;
 NoSpO2 renames  type MAP_Errors::Omission;
```

Figure 3.9: EBL error types adapted to the PulseOX

In this device, we didn't find any appropriate scenario in which providing SpO2 value may cause a hazardous situation. Thus we choose to ignore *ServiceCommission*.

Similarly to SpO2 port identify appropriate error types associated with rest of the ports. However, not all hazardous ports are identified in the functional requirement documentation. A device may have unintended interactions with its environment and other components. Identify these interaction points and associate them with *ServiceCommission* error type.

3.4 *AAMI/ANSI/IEC TIR80001*

In this section, I will illustrate the ten step risk analysis process introduced in the section 2.9.2 and its integration with AADL models. we used AADL's *property set* extensibility mechanism to add schemas for new properties capturing the IEC 80001 notions of unintended consequence (harm), hazard, hazardous situation, and initiating cause. We configured AADL's property association mechanism to allow the analyst to associate declarations of hazard, hazardous situation, and initiating cause to various points in the architecture and to specific error tokens. The analyst uses the existing AADL EMv2 error type mechanism to declare fault/error classifications appropriate to the product. Further, the analyst uses the EMv2 error propagation rules to indicate how error, faults, and their effects propagate

through the system, according to their knowledge of the system's behavior and structure.

In IEC 80001, We follow the prescribed ten-step process for performing risk analysis.

1. Identify Hazards and Hazardous Situation: A Hazard is defined as the potential source of *Harm or Unintended consequence*[43]. Hazards are the essential resource impetus to an unintended consequence. Hazards are often inherent to the properties of the system, and some may develop during the life of the system. Example hazards are electrical energy, network connectivity, sharp edge,[43] Annex A provides a list of hazards related to the medical IT-network. An external or internal event actuates a hazard, which leads to the *Unintended consequence*. Hazards are captured using the hazard property on each component.

```
Hazard: type record (
    ID: aadlstring;
    Description: aadlstring;
);
```

Figure 3.10: Hazard Property Set

Hazardous Situation is defined as the circumstance in which the system exposes the *Hazard(s)*. In ordered to identify the complete list of *Hazardous Situation*, it is recommended to perform both top-down and bottom-up analysis of the system. A *Hazardous Situation* is a sequence of events up-to an incorrect service provided by the system which leads to an undesired outcome.

Apart from the ID and the Description of the *Hazardous Situation*, we also capture the associated *Unintended Consequence* and the contributing factors in the property set as shown in Figure 3.11

2. Identify Causes and resulting Hazardous Situations: From the identified Hazards, consider the actuating event that exposes the *Hazard(s)*.[35] provide a comprehensive list of causes to consider. The actuating event for a *Hazard*can be internal or external to the component under consideration. In our property set, we also capture the likelihood by which a *Cause* may lead to a *Hazardous Situation*.

```
Hazardous_Situation: type record (
  ID: aadlstring;
  Description: aadlstring;
  P1_Likelihood: AAMI_IEC80001::LikelihoodScales;
  Paths_to_Unintended_Consequences: list of record (
    Unintended_Consequence: AAMI_IEC80001::Unintended_Consequence;
    Contributing_Factors: list of AAMI_IEC80001::Contributing_Factor;
    Likelihood_of_Transition: AAMI_IEC80001::LikelihoodScales;
    Overall_Likelihood: AAMI_IEC80001::LikelihoodScales;
    Risk: AAMI_IEC80001::RiskLevels;
    );
);
```

Figure 3.11: Hazardous Situation Property Set

```
Cause: type record (ID: aadlstring;
  Description: aadlstring;
  Likelihood: AAMI_IEC80001::LikelihoodScales;
  Location: AAMI_IEC80001::Location;
);
```

Figure 3.12: Cause Property Set

In this paper, we utilize the AADL's Fault Impact Analysis(FIA) to perform Failure Mode Effect Analysis(FMEA) on the system. FMEA is a bottom-up analysis with result containing a sequence of causal events from the root causes to a *Hazardous situation*.

3. Determine *Unintended Consequence* and estimate the potential severities: *Unintended Consequence* is defined as the unwanted outcome of an event that results in one or more degraded key properties(safety, effectiveness and network and system security). Not all *Unintended Consequence* are equally damaging. To prioritize, we assign a level of severity to the identified *Unintended Consequence*. The added severity level is useful when considering the trade-offs between the safety and functionality of the system. In[43], there are five levels of severity from the most severe to the least *Catastrophic*, *High*, *Medium*, *Low*, and *Negligible*.

```
Unintended_Consequence: type record (
  ID: aadlstring;
  Description: aadlstring;
  Severity: AAMI_IEC80001::SeverityScales;
  Key_Properties: list of AAMI_IEC80001::key_properties_type;
);
```

Figure 3.13: Unintended Consequence Property Set

Table 3.3: Definition of $\bigoplus$ operator used for combining likelihoods of different branches in the sequence of events leading to *Hazardous Situation*

Likelihood	**Improbable**	**Remote**	**Occasional**	**Probable**	**Frequent**
Improbable	Improbable	Remote	Occasional	Probable	Frequent
Remote	Remote	Remote	Occasional	Probable	Frequent
Occasional	Occasional	Occasional	Occasional	Probable	Frequent
Probable	Probable	Probable	Probable	Probable	Frequent
Frequent	Frequent	Frequent	Frequent	Frequent	Frequent

4. Estimate the likelihood of *Unintended Consequence*: We consider a qualitative analysis in which, we classify likelihoods into the following five categories namely *Improbable*, *Remote*, *Occasional*, *probable*, and *Frequent*. To estimate the total likelihood, (a) we compute the likelihood of causing a hazardous situation and combine it with (b) the likelihood of the hazardous situation leading to an Unintended consequence. A hazardous situation may be caused by multiple *Causes* either together or separately. We use $\bigoplus$ operator defined in table 3.3 when two causal paths are merged. Similarly, we use $\bigotimes$ operator defined in table 3.4 when two causes are in sequence in the same path.

Table 3.4: Definition of $\bigotimes$ operator used for combining likelihoods in sequence in the sequence of events

Likelihood	**Improbable**	**Remote**	**Occasional**	**Probable**	**Frequent**
Improbable	Improbable	Improbable	Improbable	Improbable	Improbable
Remote	Improbable	Remote	Remote	Remote	Remote
Occasional	Improbable	Remote	Occasional	Occasional	Occasional
Probable	Improbable	Remote	Occasional	Probable	Probable
Frequent	Improbable	Remote	Occasional	Probable	Frequent

5. Evaluate Risk: Risk is a function of severity and the likelihood. For each *Hazardous situation* and *Unintended consequence* pair, we compute the overall likelihood using the tables 3.3 & 3.4 and fill in the property defined by the property set in Figure 3.11 line 9. From the *Unintended consequence* property defined in step 3, we get the appropriate severity level. Using the table 3.5, we can compute the *Risk* and record it in the *Hazardous situation* property.

Table 3.5: Risk Level Matrix[43]

Severity	**Likelihood**				
	Improbable	**Remote**	**Occasional**	**Probable**	**Frequent**
Catastrophic	Moderate	High	High	High	High
High	Moderate	Moderate	Moderate	High	High
Medium	Low	Moderate	Moderate	Moderate	High
Low	Low	Low	Moderate	Moderate	High
Negligible	Low	Low	Low	Moderate	Moderate

6. Identify and document proposed *Risk Control* measures and re-evaluate *Risk*: We compare the *Risk* computed in the previous step against the acceptable risk level defined in the *Responsible organization's* risk policy. If the computed *Risk* is higher, then the *Risk control* measures are taken.[43] suggests the following three *Risk control* measure:

   - Control by design: In this method, a *Risk* is eliminated or reduced by considering an alternative design. This is recommended method of *Risk control*.
   - Protective measure: In case of no suitable alternative design, we can implement protective measures using additional components to the system.
   - Assurance case argument: Identifying and providing information on the *Risk* is also considered as a *Risk control* measure as this informs the user to avoid it. However, consider this method as a less effective *Risk control* strategy.

   Once a *Risk control* measure is chosen, it can be recorded in the model using the

property set defined in figure 3.14. Apart from the identifier and the description of the *Risk control*, a likelihood is documented either to capture the reduction in likelihood or to capture how likely the *Risk control* is effective in eliminating the *Risk*. In the earlier definition of the likelihood, the documented likelihood replaces the value in the causal chain. Consequently, in the later one, we compute an inverse of the documented *Risk control* likelihood to capture the likelihood in which the *Risk control* may fail to eliminate the *Risk*. Later we combine the inverse of *Risk control* likelihood with the likelihood in the causal path using $\bigotimes$ operator.

```
Risk_Control: type record (
    ID: aadlstring;
    Description: aadlstring;
    Updated_Likelihood: AAMI_IEC80001::LikelihoodScales;
                      );
```

Figure 3.14: Risk control Property Set

7. Implement *Risk Control* measures: In Model-Based Engineering(MBE) practice the *Risk control* is modeled and analyzed to evaluate the effectiveness. We use the FIA to compute the causal paths between the causes and *Hazardous situation* with the *Risk control* in the path.

8. Verify *Risk control* measures: Verification of the *Risk control* is performed in the implementation. Simulating and verification of the model is out of the scope of the work described in this paper.

9. Evaluate any new *Risk* arising from *Risk control*: Some *Risk control* measures either introduce new functionality or alter the existing ones. This process to introduction of new *Hazardous situation* into the system. Therefore, we reevaluate the system from step 2 to step 9 whenever the *Risk controls* are introduced. It is important to capture when a *Cause* and a *Hazardous situation* are introduced into the report as the risk analysis process is iterative and in a complex system several iterations are possible.

Additionally, a good versioning system is useful in answering why a certain design decision is made.

10. Evaluate and report overall residual risk: Residual risk is the risk computed after the application of all the *Risk controls*. For a qualitative approach discussed in this work, an acceptable criteria is decided and documented as part of the RO policies. In a qualitative approach, acceptable criteria is the maximum number of *Hazardous situations* that remain at a certain level of *Risk* after the application of the *Risk control* measures. The empty table 3.6 provides a format for generating the report form the properties defined in the model.

Table 3.6: IEC 80001 Risk analysis report format

Hazard	Hazardous Situation	Causes/Contributing Factor	Unintended Consequence	Initial Risk			Mitigation/Risk Control	Reference to Responsible Org.	Residual Risk	Overall Risk
				Severity	Likelihood	Risk				

3.4.1 Performing *AAMI/ANSI/IEC TIR80001* on PulseOX

In this section, we describe a case study that represents the risk analysis of a pulse oximeter device from the perspective of the device manufacturer. The primary purpose of the pulse oximeter is to measure the patient's blood Oxygen saturation level(SpO2). The SpO2 value is one of the indicators of the patient's health status. In the PCA interlock scenario, SpO2 levels are used to asses the patient's health status and based it the app decides whether to infuse opioid. However, in the context of the interoperable system, the device manufacturer does not know the context of the device's use. Therefore, while performing risk analysis, an analyst considers a wide range of use cases.

Although this system senses both *SpO2* value and *Pulse* rate, for the sake of simplicity, we consider only the *SpO2* value. A realistic pulse oximeter is much more sophisticated than the model described. However, in this model, we capture the core functionality of a pulse oximeter that is essential for the PCA Interlock System. Figure 3.15 shows the com-

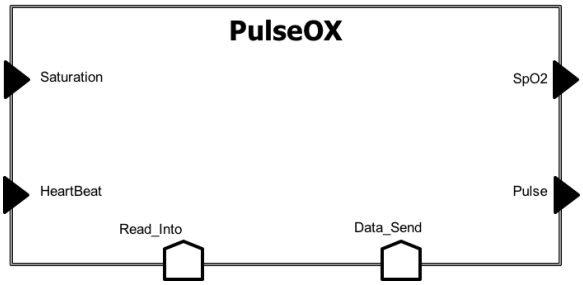


Figure 3.15: Pulse Oximeter Subsystem

ponents of the PulseOX device. In this device, a LED emits infrared light of a particular wavelength through the patient's finger, and on the other side, a photoelectric diode detects the amount of light that got absorbed. The amount that gets absorbed directly correlates with the amount of Oxygen in the blood. A controller computes the SpO2 value and the Pulse rate based on the output of the detector.

One of the responsibilities of the device manufacturer is to perform the risk analysis and share the report with other stakeholders along with the attestation provided by the third-party certification authority.

Risk Analysis

We will follow the ten step process described in section 3.4 to perform the risk analysis of the *PulseOx* system.

Step 1a: Identify *Hazards*: In this system, there are multiple hazards such as, the electricity flowing into the Pulse oximeter, the finger clip that holds the sensor to the patient, the infrared emitter, etc. Apart from these, we choose the measurement of the SpO2 as a hazard to demonstrate the process. Although SpO2 is not any form of energy source, we consider the sensitivity of the data to be hazardous. Figure 3.16 illustrates the definition of the hazard and figure 3.17 shows the application of the same in a component.

Step 1b: Hazardous Situation: For the *Hazard: Haz01*, identified in the previous step, we iden-

```
Haz01: constant AAMI_IEC80001::Hazard => [
  ID => "Haz01";
  Description => "SpO2 Measurement";
];
```

Figure 3.16: Haz01 definition

```
process PulseOx_controller
  features
    power: feature;
    detected_signal: in data port;
    SpO2: out data port;
    Pulse: out data port;
  properties
    AAMI_IEC80001::Hazards => (Hazards::Haz01) applies to SpO2;
end PulseOx_controller;
```

Figure 3.17: Haz01 application on the controller component

tify the following *Hazardous situations*:

- HS01: Incorrect SpO2 value beyond the specified error range
- HS02: No SpO2 value provided
- HS03: Sending SpO2 value over a public network in plaintext

Among these, the HS01 and HS02 cause unintended consequence in the safety and effectiveness domain, and the HS03 causes unintended consequence in both safety and security domain. Figure 3.18 shows the definition of HS01. Remember, there are a few more fields in the *Hazardous Situation* property set. We can fill those fields in the subsequent steps.

```
  HS01: constant AAMI_IEC80001::Hazardous_Situation =>  [
  ID => " HS01";
  Description => "Incorrect SpO2 value beyond the specified error range";
];
```

Figure 3.18: Definition of Hazard HS01

EMV2 enable us to define error types to capture anomalous service and associate it with a port over which the service is provided. To capture the incorrect SpO2 value, we define an error type *IncorrectSpO2* and associate it with the *SpO2* port in the PulseOx system. Furthermore, we can associate the (port, error type) with the HS01 as shown in figure 3.19.

```
annex EMV2 {**
        use types PulseOx_Errors;
        error propagations
           SpO2: out propagation {IncorrectSpO2};
        end propagations;
        properties
          AAMI_IEC80001::Hazardous_Situations =>
             (hazardous_situation::HS01) applies to SpO2.IncorrectSpO2;
**};
```

Figure 3.19: Application of Hazardous situation HS01 on EMV2's propagation

Step 2: Identify causes and resulting Hazardous situations: For each component, we consider all of the classes of faults described in[35]. We record the relevant causes in the component as a property as shown in figure 3.20

```
  Detecter_C01: constant AAMI_IEC80001::Cause => [
  ID => "Detecter_C01";
  Description => "electromagnetic interference";
  Likelihood => Probable;
  Location => External_cause;
];
```

Figure 3.20: Definition of a cause

As part of this step, we need to identify the causal relationship between a *Cause* and the *Hazardous Situation*. This step can be tedious and error-prone as a manual process for a large system. To overcome these challenges we use the Fault Impact Analysis(FIA) to automate this process. The result of the FIA analysis is the sequence of transformation from the error source associated with the *Causes 'Detecter _C03' and 'Controller _C01'* until the error sink associated with the *Hazardous situation 'HS01'*. Table 3.7 shows

```
    device implementation  Infrared_emitter.impl
  annex EMV2 {**
    use types PulseOx_Errors;
    error propagations
      power : in propagation {power_outage};
      infrared : out propagation {NoIR, IncorrectIR};
      flows
        no_power : error path power{power_outage} -> infrared{NoIR};
        faulty_emitter: error source infrared{IncorrectIR, NoIR}; --
           wrong frequency
    end propagations;
    properties
      AAMI_IEC80001::Causes => (causes::Emitter_C01) applies to
         faulty_emitter;
      AAMI_IEC80001::Causes => (causes::Emitter_C02) applies to
         power.power_outage;
  **};
end Infrared_emitter.impl;
```

Figure 3.21: Application of cause C01 and the association of the flow

the causal path between various internal causes that lead to the *Hazardous situations* associated with the error type *IncorrectSpO2* associated with the external port SpO2.

Similarly, table 3.8 shows the causal path between the error types associated with external causes and the *Hazardous situation 'HS01'*.

Table 3.7: Causal path between internal causes and *Hazardous Situation*

Fault Impact of System Internal Error Sources				
Component	**Initial Failure Mode**	**1st Level Effect**	**Failure Mode**	**second Level Effect**
detecter	{NoVitals}	{NoVitals} vitals ->controller:detected_signal	controller {NoVitals}	{NoSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]
detecter	{IncorrectVitals}	{IncorrectVitals} vitals ->controller:detected_signal	controller {IncorrectVitals}	{IncorrectSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]
controller	{IncorrectSpO2}	{IncorrectSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]		
controller	{SpO2Early}	{SpO2Early} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]		
controller	{SpO2Late}	{SpO2Late} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]		
controller	{NoSpO2}	{NoSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]		

Table 3.8: Causal path between external causes and *Hazardous Situation*

Fault Impact of System External Error Sources						
Root System	**External Error Source**	**1st Level Effect**	**Failure Mode**	**second Level Effect**	**Failure Mode**	**third Level Effect**
PulseOx_impl_Instance	infrared_in {NoIR}	{NoIR} infrared_in ->detecter:infrared	detecter {NoIR}	{NoVitals} vitals ->controller:detected_signal	controller {NoVitals}	{NoSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]
PulseOx_impl_Instance	infrared_in {IncorrectIR}	{IncorrectIR} infrared_in ->detecter:infrared	detecter {IncorrectIR}	{IncorrectVitals} vitals ->controller:detected_signal	controller {IncorrectVitals}	{IncorrectSpO2} SpO2 ->PulseOx_impl_Instance:SpO2 [External Effect]

Step 3: Determine the Unintended Consequence and estimate the potential severities: We can identify the *Unintended Consequence* as described in section 3.4 and capture it

as a property in the model as shown in figure 3.22. In case of the electromagnetic interference, the sensor may not provide the correct SpO2 measurement. Therefore, if the patient's critical care is relies upon the SpO2 measurement provided by the PulseOX, then the discrepancy in the SpO2 value may lead to misdiagnosis or incorrect treatment. In both the cases, the patient is harmed and may lead to loss of life depending upon the treatment and the level of autonomy in the controller of the following system. Thus, a severity level of Catastrophic is assigned.

```
UC01 : constant AAMI_IEC80001::Unintended_Consequence => [
  ID => "UC01";
  Description => "Misdiagnosis or Incorrect treatment";
  Severity => Catastrophic;
  Key_Properties => (Safety);
];
```

Figure 3.22: Definition of Unintended Consequence UC01

Step 4: Estimate the probability of Unintended Consequence: In this step, we associate the *Hazardous situation* with the *Unintended Consequence* along with the other worst-case environmental conditions that are necessary for harming the patient. If the PulseOx is not in use or the usage is not in critical care, then there is no possibility of harming the patient. We capture these as part of each *Hazardous situation* along with the likelihood of the contributing factors as shown in figure 3.23.

Once the complete causal chain from *Cause* to *Unintended Consequence* is established, we can use the tables 3.3 and 3.4 to compute the likelihood of a *Hazardous situation* to *Unintended consequence*.

Step 5: Evaluate Risk: Risk = F(Severity, Likelihood). Using the table 3.5, we can compute risk form the likelihood level of probable and severity level of Catastrophic to a risk level of High. High risk is unacceptable for medical devices. Therefore risk control measures are to be undertaken. However, the root cause of this *Unintended Consequence* is due to an external cause of electromagnetic interference. Therefore, we cannot eliminate

```
HS01: constant AAMI_IEC80001::Hazardous_Situation =>  [
    ID => " HS01";
          Description => "Incorrect SpO2 value beyond the specified
             error range";
          Paths_to_unintended_Consequences => (
[Unintended_Consequence => UnintendedConsequences::UC01;
Contributing_Factors => (ContributingFactors::CF01);
Likelihood_of_Transition => Probable;
Overall_Likelihood => Probable;
Risk => High;]
         );
];
```

Figure 3.23: Updated Hazardous situation

the root cause. However, we can mitigate the likelihood of the resulting *Hazardous situation*.

Table 3.9: Risk analysis report at the end of Step 5

Hazard	**Hazardous Situation**	**Causes/Contributing Factor**	**Unintended Consequence**	**Initial Risk**			
				Severity	Likelihood (Internal cause)	Likelihood (External cause)	Risk
Haz01: SpO2 Measurement	HS01: Incorrect SpO2 value beyond the specified error range	C01: electromagnetic interference	UC01: Misdiagnosis or Incorrect treatment	Catastrophic		Probable	High

Step 6: Identify and Document proposed *Risk control* measures: The *Risk* evaluated for the *Hazardous situation HS01* is High, which is unacceptable based on the *Responsible Organization's* policy. Therefore, we attempt to mitigate the *Risk* using the risk control measures discussed in section 3.4 step 6. As the cause is external to the system, 'control by design' risk mitigation strategy such as redundant detector is not suitable because even the redundant sensor will be affected by the same interference. Therefore, we choose protective measures such as better shielding and the design of the fingerclip to minimize the interference. Additionally, the controller is modified to detect and stop outputting any value out of the specified range. *e.g.* controller refrains to provide a SpO2 value of 110 as it is out of the operational range. Furthermore, the possibility of electromagnetic interference and the likelihood of a *Hazardous situation* is documented, and the user manual is updated with mitigation procedures for this *Hazardous situation*.

Table 3.10: Risk analysis report at the end of step 6

Hazard	Hazardous Situation	Causes/Contributing Factor	Unintended Consequence	Initial Risk				Risk Control		Reference to Responsible Organization's document	Residual Risk		
				Severity	Likelihood (Internal cause)	Likelihood (External cause)	Risk	Mitigation Measure	Likelihood		Severity	Updated Likelihood	Risk
Haz01: SpO2 Measurement	HS01: Incorrect SpO2 value beyond the specified error range	C01: Electromagnetic interference	UC01: Misdiagnosis or Incorrect treatment	Catastrophic		Probable	High	RC01: Better shielding of the infrared light and better finger clip design	Remote	Updated user manual containing procedures to reduce EM interference	Catastrophic	Remote	High

Table 3.10 illustrates the updated risk analysis report. Although we have not reduced the risk level, we have implemented some protective measures. In case the PulseOx is used as a part of another system, the system integrator is responsible for mitigating this risk further.

Step 8: As we are concerned with the design of the system, implementation and verification of the risk control are out of scope for this paper.

Step 9: Evaluate any new *Risks* from *Risk control*: In the PulseOx system, we do not see any new risks arising from the above-described risk control at this point.

Step 10: Evaluate and Report overall *Residual risk*: Table 3.10 describes the overall risk of the system only regarding the cause 'Detector::C01'. However, the overall risk is computed by performing the steps 1-10 for all the identified *Hazards* and *Hazardous situation*. The overall *Residual Risk* is computed by the number of *Hazardous Situation* that remains at Medium level risk or above after applying all the *Risk controls*. The *Responsible Organization(RO)* policy provides the acceptability criteria based on the overall risk.

The key feature of this risk analysis process is the ability to prioritize the crucial *Hazardous situations* over not severe ones. Additionally, this helps in the safety vs functionality trade-off.

Redacted risk analysis report

A report developed by performing the steps described in the 3.4.1 is provided to a third party certification authority. A third party certification authority, verifies the risk analysis process and the report against the manufactured device. If the risk analysis of the device is satisfactory, then a certificate is issued. This certificate is shared among the other stakeholders.

Table 3.11: PulseOX redacted risk analysis report

Hazard	Hazardous Situation	Causes/Contributing Factor	Unintended Consequence	Residual Risk			
				Severity	Likelihood (Internal cause)	Likelihood (External cause)	Risk
Haz01: SpO2 Measurement	HS01: Incorrect SpO2 value beyond the specified error range	C01: Electromagnetic interference C02: Nailpolish C03: Other medications in the blood C04: Skin tone C05: Movement / dislodge of the probe C06: Incorrect calibration C07: Internal failures within the specified life time	UC01: Misdiagnosis or Incorrect treatment	Catastrophic	Remote	Remote Remote Remote Probable Probable Remote	High
	HS02: Pulse Oximeter fails to provide SpO2 reading	C05: Movement / dislodge of the probe C07: Internal failures within the specified life time	UC02: Unable to provide treatment	Catastrophic	Remote	Probable	High

Table 3.11 shows the redacted version of the risk analysis report that is shared with other stakeholders. A noticeable difference from the report format described in 3.4 step 10 is the separation of the likelihood based on the root cause's point of origin. In this way, a device manufacturer can abstract all the internal failures and their likelihood in to one value. Thus, the device manufacturer protects her intellectual property at the same time providing the required information to the system integrator. This report contains only the final status of the device without including all the intermediary risk controls. The list of external causes and their corresponding likelihood informs the system integrator to take appropriate measures to mitigate these causes with respect to the context of use.

3.5 Open PCA Pump - Actuator

This section provides a brief summary of the Open PCA Pump model, with a focus on attributes related to the risk analysis.

Figure 3.25 shows the Open PCA Pump in its context as a system (rounded-corner rectangle). It's primary purpose is to infuse drug into a patient, modeled as an abstract component (dashed-line rectangle). The patient can press a button requesting a bolus of pain medication, that will only be delivered when several safety conditions are met. A clinician (nurse) sees information displayed on the control panel including current infusion rate, and alarm or warning indications, hears warning or alarm sounds, and controls the pump through a touch screen. A maintenance jack allows physical connection with a maintenance cable (double-ended arrow) to a dongle on a laptop (perspective box) running tech software (parallelogram) which can load a drug library or extract event or fault logs. As an inter-

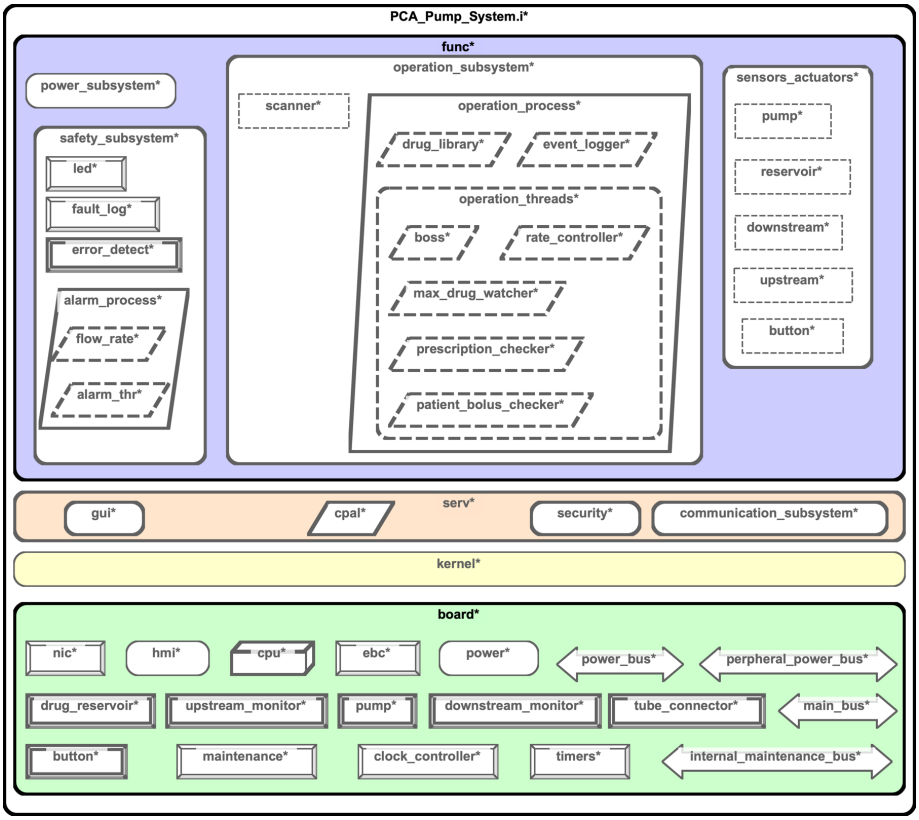


Figure 3.24: Open PCA Pump Containment Hierarchy

operable device, it connects through an ICE network, which may also serve other medical devices, to an ICE supervisor processor which executes control software (apps) to coordinate operation of medical devices. The `pca_pump` system is expanded in Figure 3.24.

Figure 3.24 shows the main pump function, together with its maintenance interface reach though a physical jack, and its ICE interface for networking. The `pump` system is elaborated in Figure 3.25.

The Open PCA Pump extends and specializes the ISOSCELES medical device reference architecture[52]. The ISOSCELES reference architecture is essentially an AADL model that separates functional architecture (including software) from the physical architecture (components, wires and assemblies), and includes generic subsystems for operation, safety, user interface, network interface, power, and sensors/actuators. The Open PCA Pump AADL model extends the ISOSCELES architecture with sensors and actuators for drug infusion, and detailed software behavior.

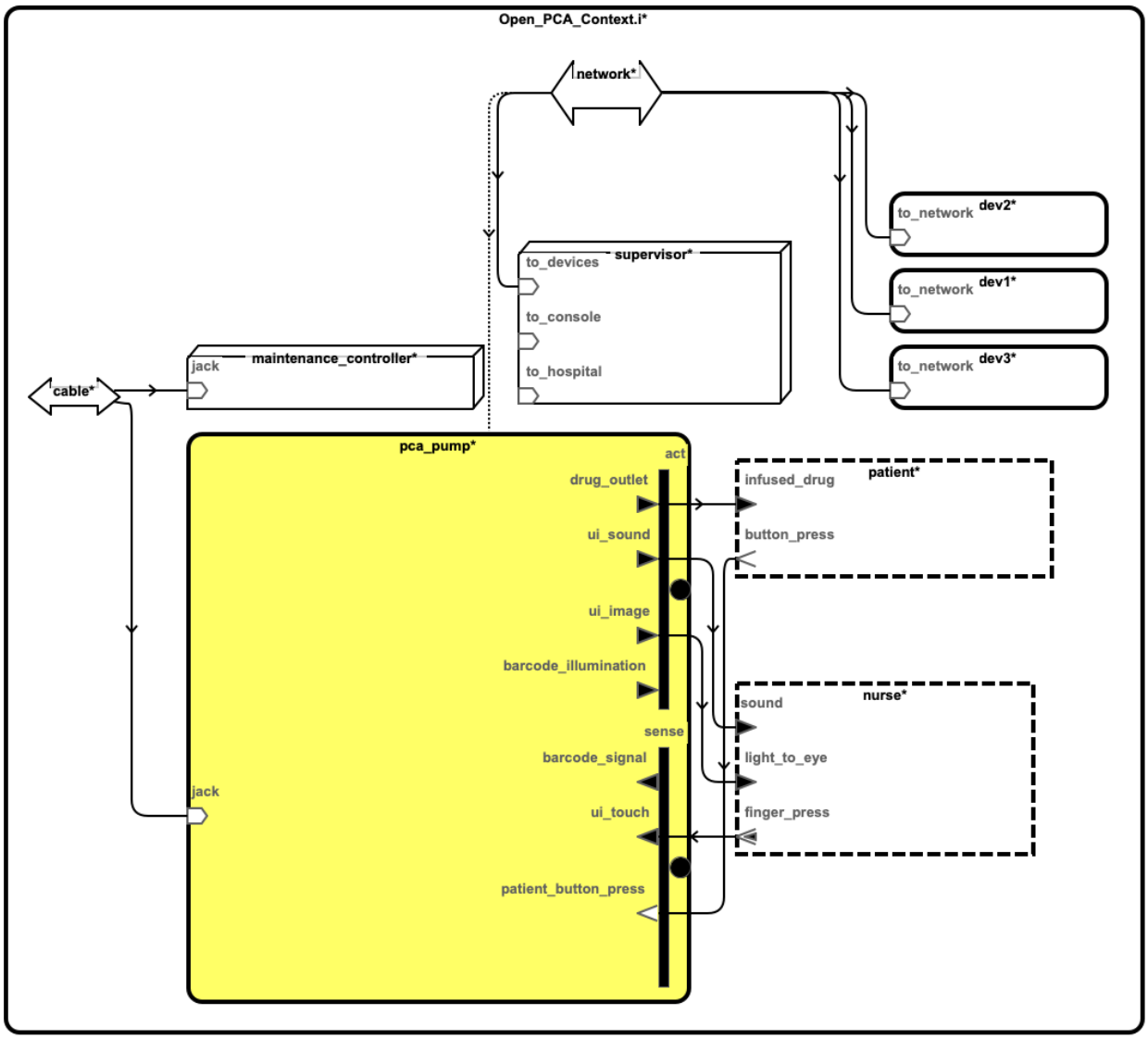


Figure 3.25: Context for Open PCA Pump

Figure 3.24 shows the Open PCA Pump containment hierarchy which retains the ISOSCELES architectural layering of a functional architecture using ISOSCELES runtime services, isolated by a separation kernel, executed by physical hardware.

The full architecture includes separate AADL projects for the ISOSCELES medical device platform and its Open PCA Pump refinement, having thirty-nine packages together. AADL distinguishes component *types* which define externally-visible interfaces, from component *implementations* which define internal behavior and decomposition into subcomponents. The Open PCA Pump AADL model defines in total 121 component types and implementations.

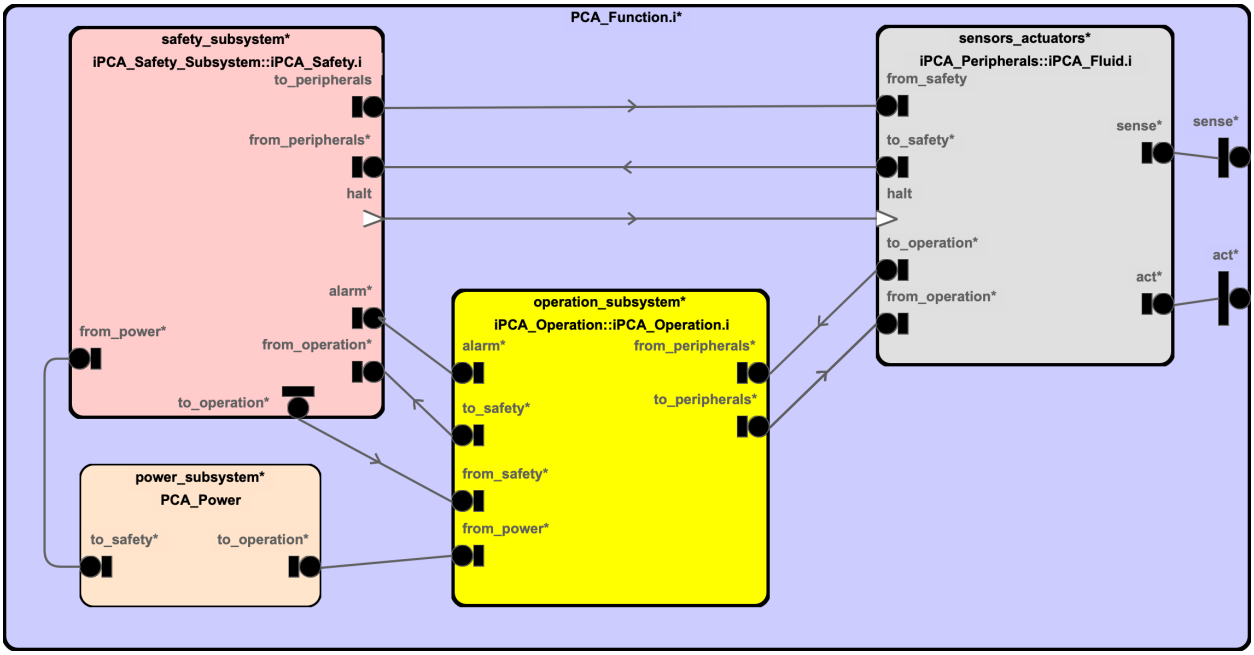


Figure 3.26: PCA Pump Functional Architecture

Figure 3.26 shows the top-level of the Open PCA Pump functional architecture, and its four subsystems: operation, safety, sensors/actuators (fluid), and power. The ports and connections between components are modeled using AADL feature groups – with each connection (line) aggregating many event and data flows (these are automatically broken out in visualizations of Section 7.3).

The architecture shown in Figure 3.25 has some physical components (power bus, internal bus, processor, and memory), but it is mostly a functional architecture having four

subsystems: safety, operation, power, and fluid.

3.5.1 Safety Subsystem

The safety subsystem presented in Figure 3.27 is an independent monitor that looks for unsafe conditions and malfunctions in other subsystems. The design of the safety subsystem is based on principles for a medical device safety architecture presented in[53]. The two main function of safety subsystem is to detect errors and raise alarms when there is an error.

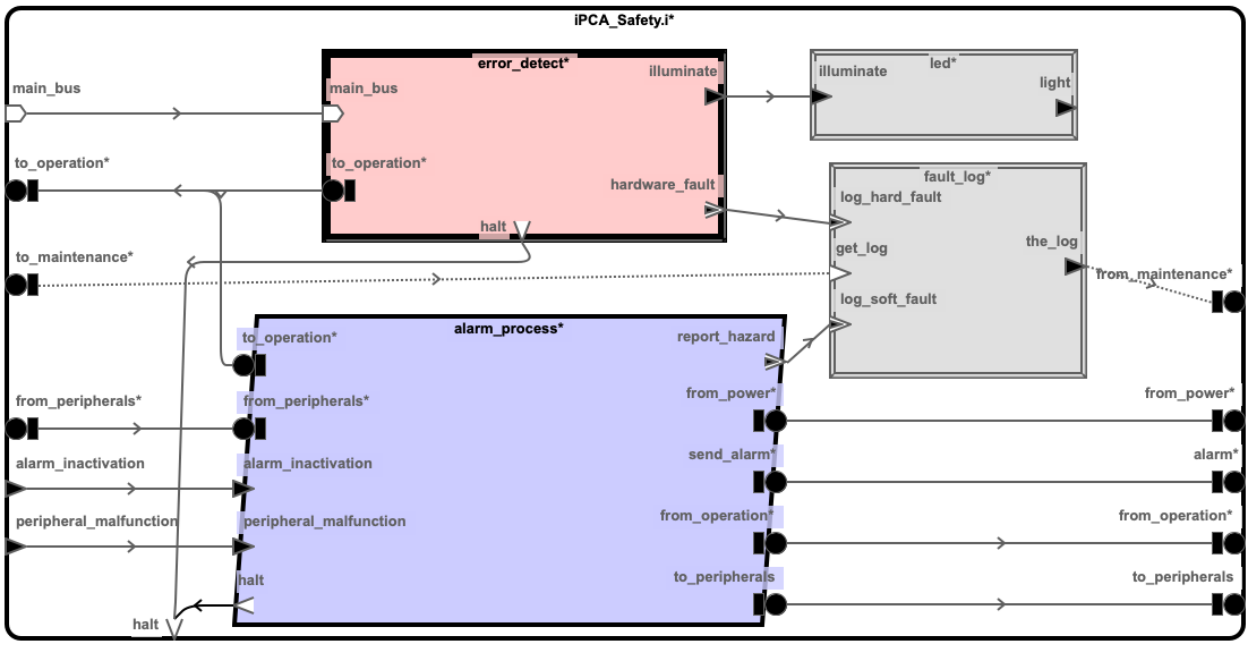


Figure 3.27: Safety Subsystem

In AADL, a process is a protected address space. All threads must be contained in a process. The Alarm Process, Figure 3.28 holds threads that perform safety functions that check whether the actual flow rate is within tolerance, and determine which warning or error should be displayed and sounded. The Alarm Thread also determines appropriate actions in response to warnings or errors. In some conditions, the pump flow should be stopped completely; in others, a keep-vein-open (KVO) rate is appropriate.

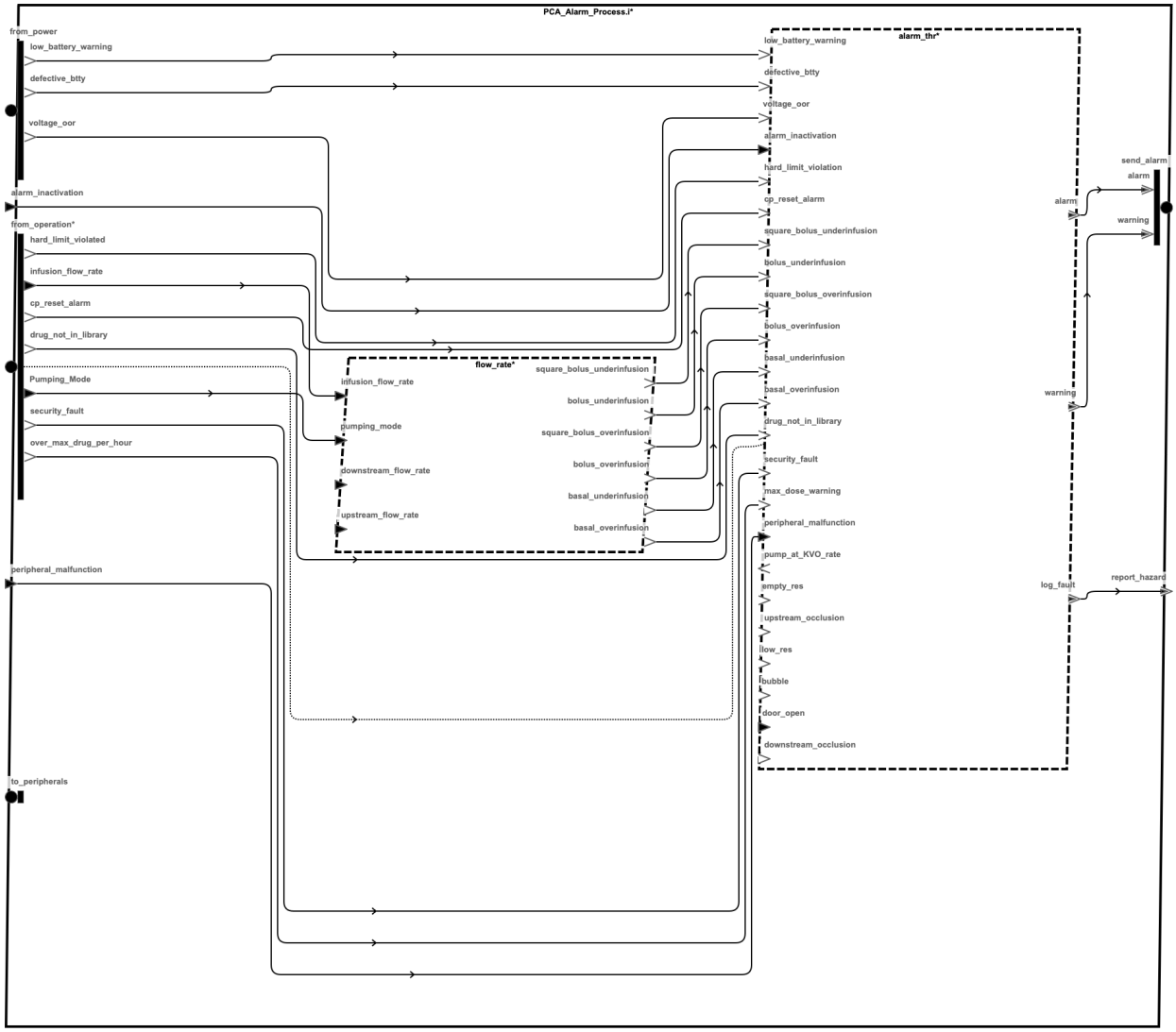


Figure 3.28: Safety Process

3.5.2 Fluid Subsystem

The fluid subsystem includes the components that hold, move, or measure the liquid drug.

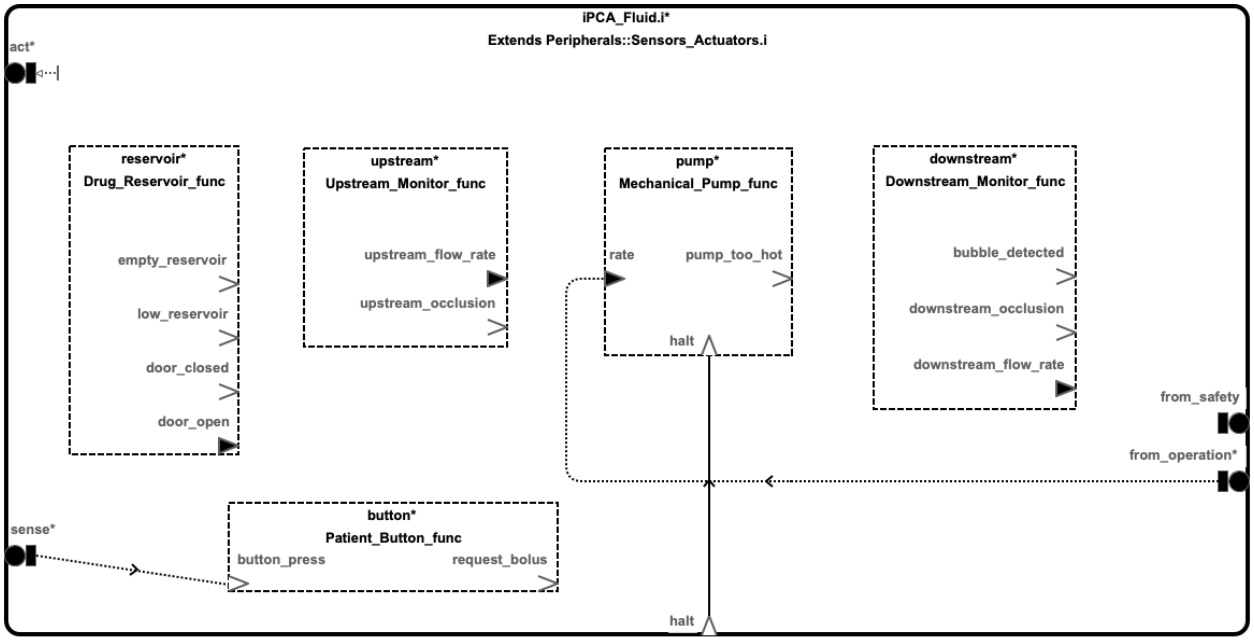


Figure 3.29: Fluid Subsystem

3.5.3 Operation Subsystem

The operation subsystem (Figure 3.30) makes most of the decisions regarding pumping, but can be overridden by the safety subsystem. It is depicted here without ports or connections because they would be too small to read. It contains two devices: a patient button requesting an additional bolus of pain medication, and a scanner for reading the patient's wristband, clinician badge, and drug vial label. The scanner could be optical, reading bar or QR codes, or RFID.

A security subsystem (not shown) provides functionality to authenticate the patient, clinician, and drug to ensure that the right drug is infused into the right patient, and authorizes the earlier authenticated clinician to ensure that the pump is attended by a trained clinician.

The operation process holds most of the software. The drug library holds a database of allowed pain medication with soft limits (exceeded with clinician approval) and hard limits

which prevent operation if exceeded. All actions are recorded by an event logger for later review.

AADL allows thread to be combined into a thread group, when convenient. The `operation_threads` group combines threads that work together:

`boss` sequences use and exception cases

`ice_thread` communicates with the `bus_adaptor`

`rate_controller` determines pumping rate

`max_drug_watcher` limits the volume of drug infused in any hour

`prescription_checker` compares prescription from drug vial label with the drug library for soft and hard limit enforcement

`patient_bolus_checker` enforces prescription's minimum time between boluses

3.5.4 Power Subsystem

The power subsystem conditions DC voltage from either mains source or battery backup, and provides information and warnings about the power supply. Though comparatively simple, the power subsystem requires careful engineering for dependable power supply and ordering during power-on.

3.6 App - Controller

Developing a safety critical system is an iterative process. Figure 3.32 illustrates the interaction between the model and its risk analysis report. After a change to the model, the risk analysis report indicates the level of risk in the model. Based on the risk level, the system integrator performs appropriate mitigation steps. Once again, the model is subjected to the risk analysis, and this process continues until the residual risk is reduced to acceptable levels.

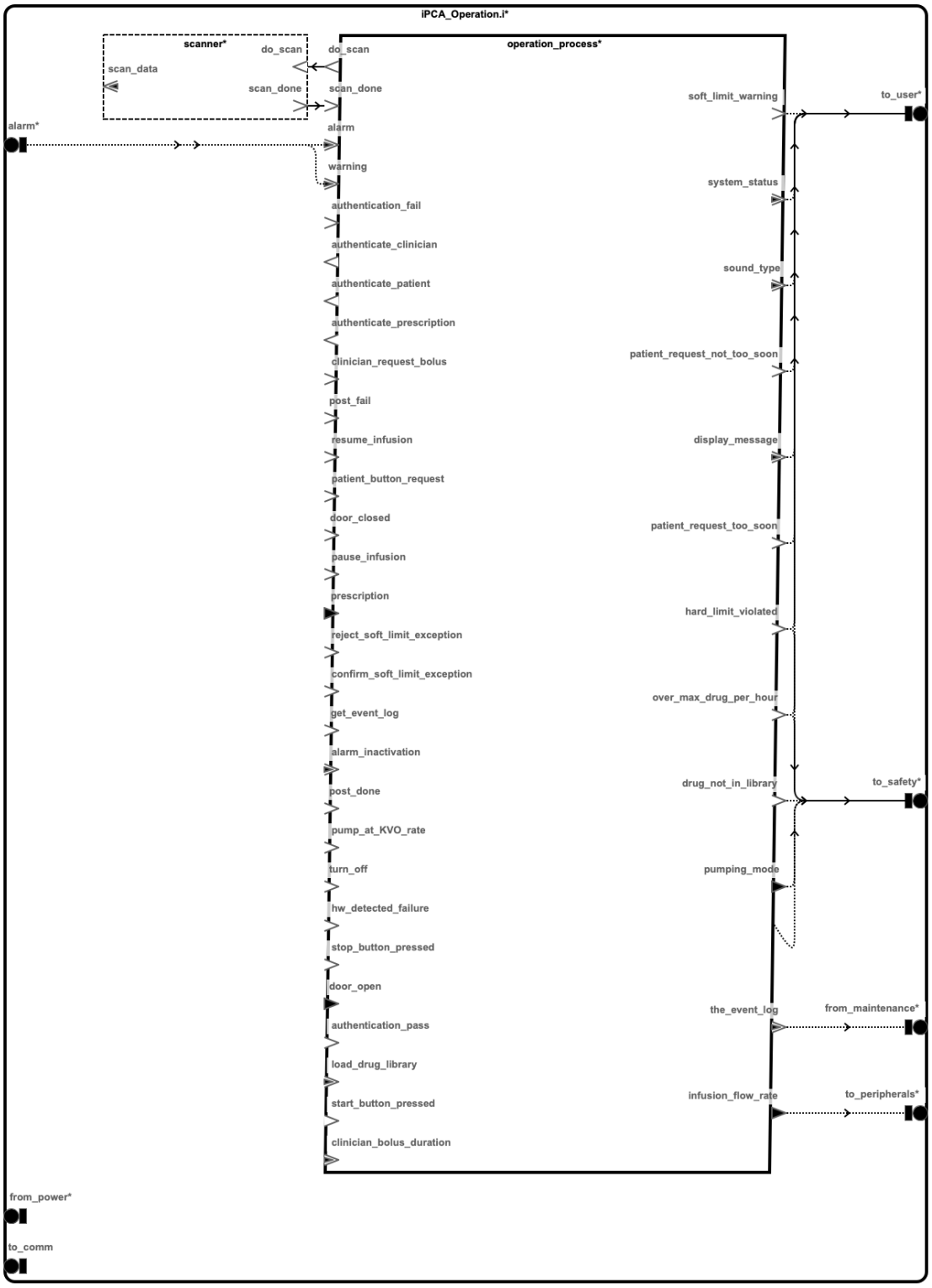


Figure 3.30: Operation Subsystem

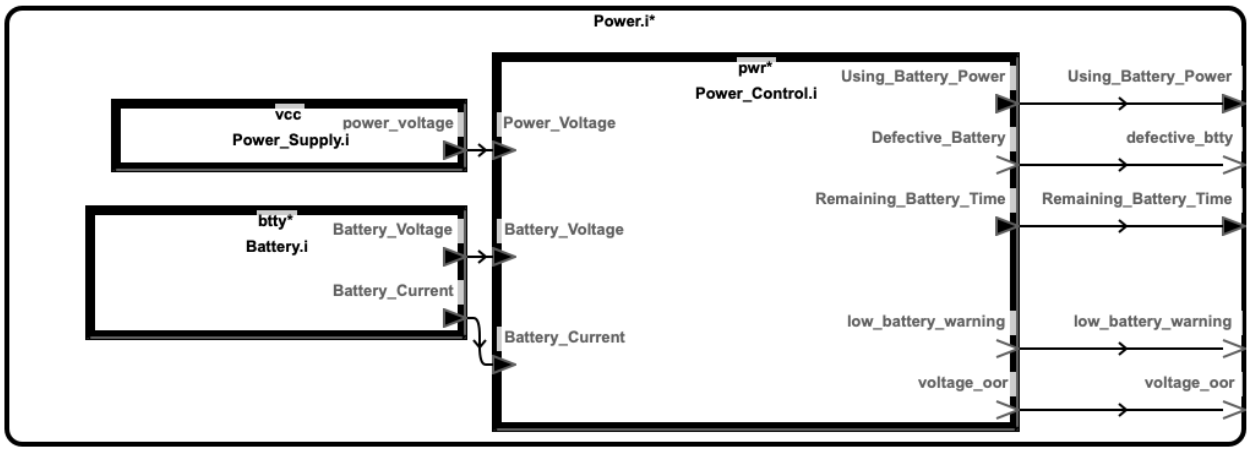


Figure 3.31: Power Subsystem

Often the system integrator is also the one who develops the App for the system. Figure 3.33 provides a pseudo code for the app capturing the PCA interlock scenario discussed in section 2.4.4.

In this version, risk analysis should address the possibility of communication failures between the App and PCA Pump. Such errors could cause a shut-off command from the App to be lost, allowing the Pump to continue in"run normally" mode and thus cause an over-infusion. Security analysis can be applied to consider situations where the command is blocked or altered. It may be the case that the same error tokens are used for safety, but there may be both a safety and security interpretation for each token.

3.6.1 Version II

Safety issues caused by the above scenario lead the designer to replace the shutoff command between the App and pump with a risk control measure based on the notion of a "ticket to run normally for MM:SS". Based on the sensed values, that App periodically issues a tick to the Pump enabling it to pump normally for a certain period of time (MM:SS).

The risk analysis should consider the same communication failure between the App and PCA Pump as before. However, the analysis should enable the analyst to determine that even if the communication fails (a ticket is not delivered to the PCA Pump), the Pump will stop

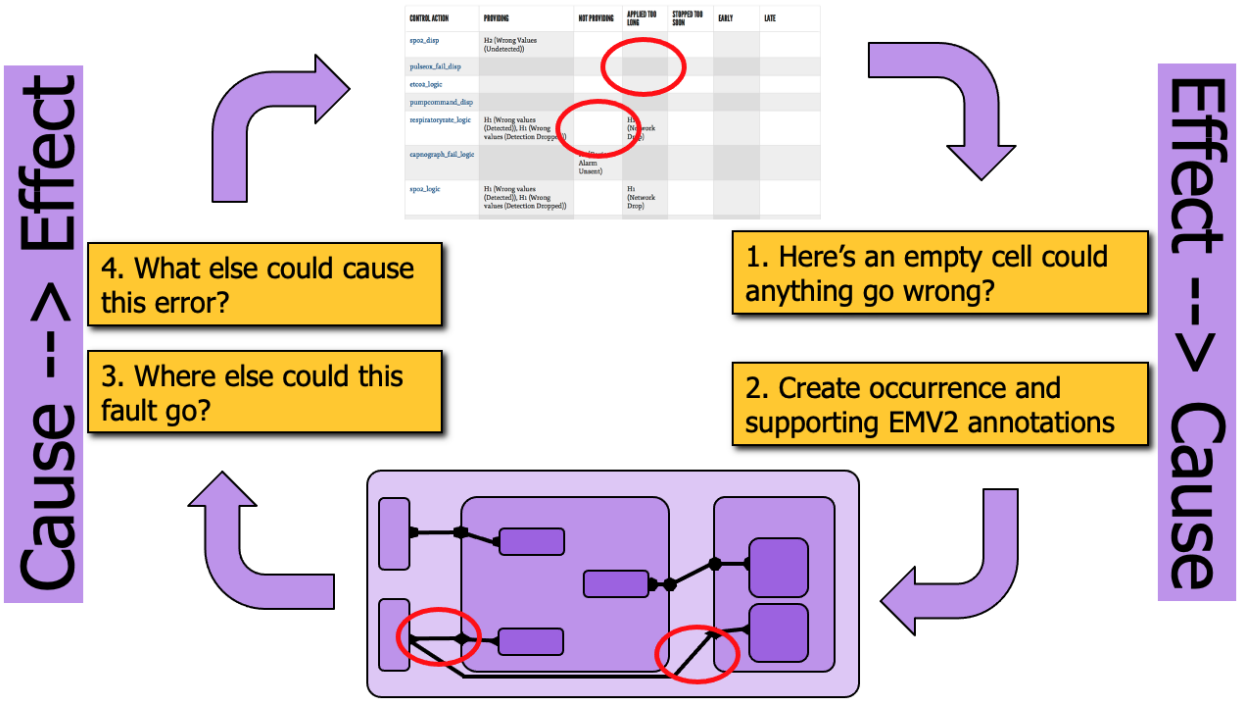


Figure 3.32: Interaction between Report and Model

infusion (its ticket will expire) before deteriorating the patient's health. The App is modified with ticket-based control logic and develops a test case that simulates the communication failure between the App and the Pump to check the effectiveness of the mitigation.

The analyst should consider communication failure between the sensor (e.g., pulse ox) and App. Such a failure should cause the analyst to consider the impact on the App's ability to generate a valid ticket. Doing this requires an analyst's understanding of the functionality of the App's ticket generating algorithm and requires some indication if the App is aware of the failure of communication.

3.6.2 Version III

To mitigate the risk in the previous version, the system is added with redundant sensing components (e.g., 3 pulse oximeters or multiple forms of respiratory health sensing). Figure 3.34 shows the PCA interlock system with the addition of capnography and respiratory rate monitor. The App developer should update the logic to consider the additional sensors.

```
while TRUE do
    read spO2, rr, etCO2;
    read pulseOx_tech_alarm, capnography_tech_alarm;
    if (etCO2 > 60 mmHg ) or (rr < 10) or (spO2 < 94) or pulseOx_tech_alarm
     or capnography_tech_alarm then
        Send STOP_INFUSION signal;
    else
        Send START_INFUSION signal;
    end
    delay(100)
end
```

Figure 3.33: Interlock Algorithm

Figure 3.35 captures the modified error behavior for the App. The App produces incorrect ticket values only when all three sensors are producing incorrect values. If only two or fewer fails, the App can refrain from giving a ticket and thus avoid hazardous situations.

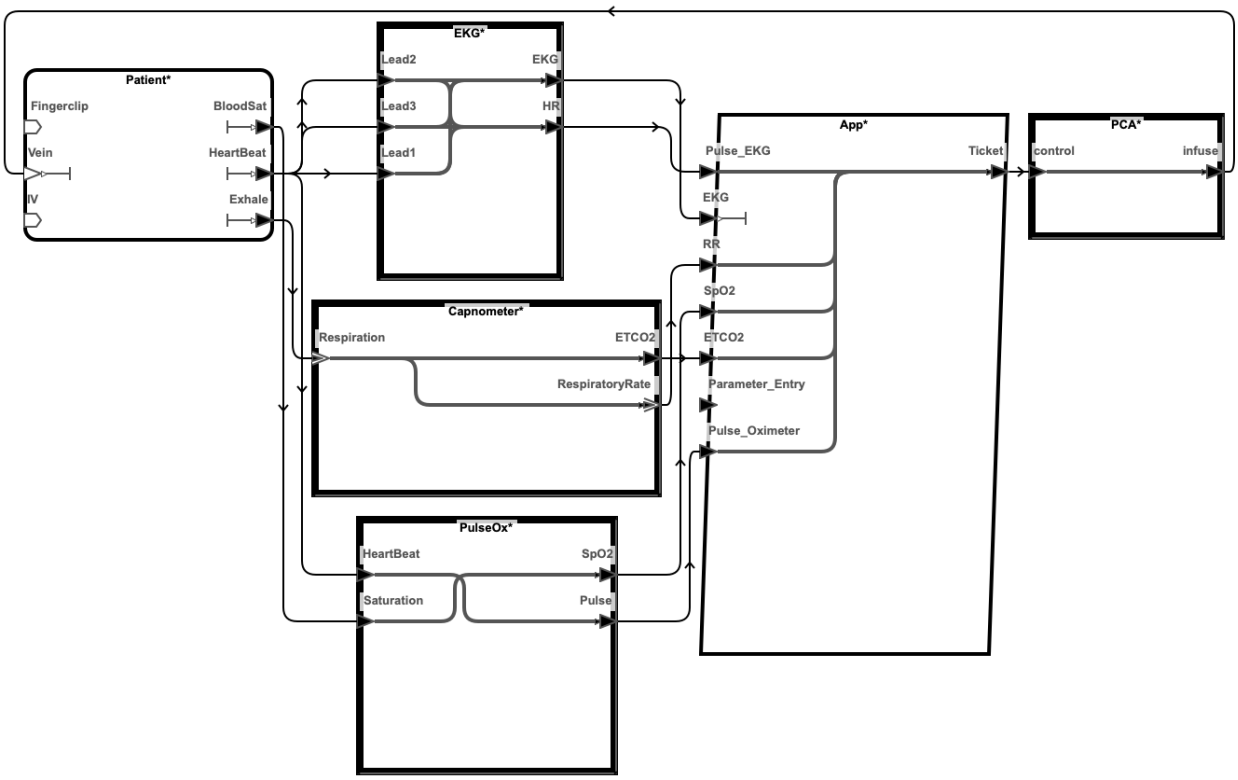


Figure 3.34: PCA interlock with redundant sensors

```
-- the overdose hazard situation
    InadvertantPump: Normal -[SpO2{SpO2ValueHigh} and
        ETCO2{ETCO2ValueLow} and
        RR{RespirationRateHigh}
    ]-> CmdPumpNorm{TicketTooLong};

-- redundency mechanism to mitigate fault
    StopPump: Normal -[2 orless(SpO2{SpO2ValueHigh},
        ETCO2{ETCO2ValueLow},
        RR{RespirationRateHigh}
    )]-> CmdPumpNorm{NoError};
```

Figure 3.35: Augmented error behavior

Chapter 4

Theories and Tools

This chapter provides the theoretical background for all the analyses discussed in this dissertation. Section 4.2 illustrates the fault propagation and transformation calculus. Finally, section 4.3 reviews other tools that can perform safety analysis.

4.1 Lattice Theory

Lattice is an algebraic structure consisting of partial order sets.

Definition 1: Partial order($\mathcal{L}$, $\sqsubseteq$) or poset, consists of nonempty set $\mathcal{L}$ and a binary relation $\sqsubseteq$ on $\mathcal{L}$ which is,

1. **Reflexivity:** $\forall\ x \in \mathcal{L},\ x \sqsubseteq x$
2. **Anti-symmetry:** $\forall\ x,\ y \in \mathcal{L},\ (x \sqsubseteq y) \wedge (y \sqsubseteq x) \Rightarrow x = y$
3. **Transitivity:** $\forall\ x,\ y,\ z \in \mathcal{L},\ (x \sqsubseteq y) \wedge (y \sqsubseteq z) \Rightarrow x \sqsubseteq z$

Figure 4.1 is a Hass diagram representing the power set of {x, y, z}. Where the binary relation is the subset operation ($\sqsubseteq\ =\ \subseteq$). For example, {x} $\subseteq$ {x, y} $\subseteq$ {x, y, z}. When I mention $x \sqsubseteq y$, that means y is a safe approximation of x, or x is at least as precise as y.

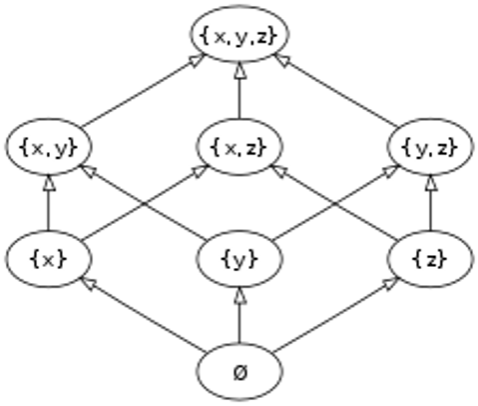


Figure 4.1: Hass diagram for power set of {x, y, z}

Definition 2: Greatest element or maximum in a poset is an element where all other elements are lesser or equal by the binary relation $\sqsubseteq$. Formally defined as:

$$\text{Let } (\mathcal{L}, \sqsubseteq) \text{ be a poset, } \exists\ x \in \mathcal{L},\ \forall\ y \in \mathcal{L},\ y \sqsubseteq x.$$

From the figure 4.1, set {x, y, z} is the greatest element.

Definition 3: Least element or infimum in a poset, is an element where all other elements are greater or equal.

$$\text{Let } (\mathcal{L}, \sqsubseteq) \text{ be a poset, } \exists\ x \in \mathcal{L},\ \forall\ y \in \mathcal{L},\ x \sqsubseteq y.$$

Definition 4: Upper bound For a poset $(\mathcal{L}, \sqsubseteq)$ and a subset $\mathcal{A} \sqsubseteq \mathcal{L}$ say p is an upper bound of $\mathcal{A}$ iff

$$\forall \text{q} \in \mathcal{A},\ \text{q} \sqsubseteq \text{p}$$

Say p is a least upper bound (LUB) of $\mathcal{A}$ iff

1. p is an upper bound of $\mathcal{A}$, and
2. for all upper bound q of $\mathcal{A}$, p $\sqsubseteq$ q.

In figure 4.1, if $\mathcal{A} = \{\{x\}, \{y\}\}$, then the upper bound based on the definition is $\{\{x, y\}, \{x, y, z\}\}$. The least upper bound (LUB) is $\{x, y\}$

Definition 5: Lower bound For a poset $(\mathcal{L}, \sqsubseteq)$ and a subset $\mathcal{A} \sqsubseteq \mathcal{L}$ say p is an lower bound of $\mathcal{A}$ iff

$$\forall\ q \in \mathcal{A},\ p \sqsubseteq q$$

Say p is a greatest upper bound (GLB) of X iff

1. p is a lower bound of $\mathcal{A}$, and
2. for all lower bounds q of $\mathcal{A}$, we have $q \sqsubseteq p$.

Greatest lower bound of $\mathcal{A}$ is written as $\sqcap\mathcal{A}$.

If $\mathcal{A} = \{\{x,\ y\}\}$, then the lower bound $= \{\{x,\ y\},\ \{x\},\ \{y\},\ \varnothing\}$. The greatest lower bound (GLB) is $\{x,\ y\}$.

Least upper bound of $\mathcal{A}$ is written as $\bigsqcup\mathcal{A}$. Similarly greatest lower bound is written as $\bigsqcap\mathcal{A}$. Also for a pair of elements, infix notation $x \sqcup\ y$can be used instead of $\bigsqcup\{x,y\}$. This operator is also called as join of x and y. Likewise, $x \sqcap\ y$is called as meet of x and y. Most conservative program analysis are performed by computing LUB.

Definition 6: Ascending chain: An w-chain is an increasing chain of the poset elements of the form $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n$

Definition 7: Complete partial order is a partial order $(\mathcal{L},\ \sqsubseteq)$ if it has LUBs of all w-chain $d_0 \sqsubseteq d_1 \sqsubseteq \cdots \sqsubseteq d_n \sqsubseteq \cdots$, *i.e.* any increasing chain of elements in $\mathcal{L}$ has a LUB $\bigsqcup\{d_n | n \in w\}$ in $\mathcal{L}$.

Definition 8: Monotonic function: A function of the form $f : D \to E$ between two complete partial order is monotonic iff

$$\forall d, d' \in D, d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$$

Definition 9: Continuous function: A function is continuous iff it is monotonic and for all chains in complete partial order set $\mathcal{L}$ we have

$$\bigsqcup_{n \in w} f(d_n) = f(\bigsqcup_{n \in w} d_n)$$

Definition 10: Fixed point: For a function of the form $f : \mathrm{D} \to \mathrm{D}$ on a complete partial order D, a fixed point is an element $d \in \mathrm{D}$ such that $f(d) = d$.

Definition 11: Prefixed point of a continuous function f is an element d of D such that $f(d) \sqsubseteq d$.

Theorem: Fixed-point

Let $f : \mathrm{D} \to \mathrm{D}$ be a continuous function on a complete partial order set with bottom $\perp$.

$$\text{Define } \mathit{fix}(f) = \bigsqcup_{n \in w} f^n(\perp)$$

Then $\mathit{fix}(f)$ is a fixed point of f and the least prefixed point of f *i.e.*

1. $f(\mathit{fix}(f)) = \mathit{fix}(f)$ and
2. if $f(d) \sqsubseteq d$ then $\mathit{fix}(f) \sqsubseteq d$.

Definition 9: Complete lattice is a partial order set$(\mathcal{L}, \sqsubseteq)$ in which every subset of $\mathcal{L}$ has a GLB and LUB. Trivially every complete lattice is a lattice. Every complete lattice $(\mathcal{L}, \sqsubseteq)$ has a unique greatest element called Top $\top = \bigsqcup \mathcal{L}$ and unique least element called bottom $\perp = \bigsqcap \mathcal{L}$.

Theorem: Knaster-Tarski Theorem for minimum fixed points[54;55] Let $(\mathcal{L}, \sqsubseteq)$ be a complete lattice, let $f : \mathcal{L} \to \mathcal{L}$ be a monotonic function. Define

$$\mathrm{m} = \bigsqcap \{\ x \in \mathcal{L} \mid f(x) \sqsubseteq x\ \}.$$

Then m is a fixed point of f and the least prefixed point of f.

In other words this theorem proves that the greatest lower bound of all prefixed points of $\mathcal{L}$ is a fixed point and it is the least prefixed point of f.

Theorem: Knaster-Tarski Theorem for maximum fixed points[54;55] Let $(\mathcal{L}, \sqsubseteq)$ be a complete lattice, let $f : \mathcal{L} \to \mathcal{L}$ be a monotonic function. Define

$$\mathrm{M} = \bigsqcup \{\ x \in \mathcal{L} \mid x \sqsubseteq f(x)\ \}.$$

Then M is a fixed point of f and the greatest postfixed$(x \mid\ x \sqsubseteq f(x))$ point of f.

The proof is similar to the theorem of minimum fixed points by the monotonicity property on (L, $\sqsubseteq$).

Complete lattice and Knaster-Tarski theorem plays an important role in classical dataflow analysis. Typically, the lattice represent an abstraction information about the control flow graph (CFG) node. For reaching definition analysis the lattice is the *powerset* of set of all assignments. Computing the least fixed point from the *iota* initial set provides the set of reachable definition for every use of variables. The Knaster-Tarski theorem tells that the solution is optimal, and the fixed point theorem tell that the algorithm will terminate as long as the lattice is complete and the transfer function is monotonic.

4.1.1 Error Domains

EMv2 error types can be structured as a lattice of errors, with the top node being *error* denoting a set of all possible errors in the system. Each subset of *error* can be a category of error such as *service error* or *timing error*. At the bottom of this lattice are the concrete values such as natural number 70 that cause the system to deviate from its normal behavior.

4.1.2 Security Domains

Denning *et al.*[56] demonstrated the application of lattice to model security domains and provided a mechanism to guarantee secure information flow. An information flow model is F is defined as

$$F = (N, P, SC, \oplus, \rightarrow)$$

Where,

1. N : is a set of program variables in case of AADL, it can be set of ports
2. P : is a set of process or components that are responsible for all information flow
3. SC : is a set of security class/domain modeled as a lattice
4. $\oplus$: is the combining operator on SC, typically LUB on SC

5. $\rightarrow$: is a relation defined as SC X SC. This relation captures the permitted flow of information between the security domains.

With this mechanism, an analyst can detect and prevent the leak of confidential information in a system. However, access control policies are not addressed by this technique.

4.2 Failure Propagation and Transformation Calculus (FPTC)

FPTC is a methodology to capture local failure behavior as propagation and transformation of failures in each component. Based on the local failure behavior, global failures are automatically computed by the FPTC analysis. The EMv2 error behavior is based on the FPTC semantics. Therefore there are a lot of common aspects between the EMv2 and FPTC. The failure behaviors are captured in terms of the *guidewords* or error tokens such as *early, late, stale, etc.* FPTC also includes a special representation '*' for no failure. Similar to EMv2, there are three kinds of transformation expressions: *Source* - Where a component produces failure when there are no failures provided to it. *Sink* - Where a component consumes or handles a failure. Lastly, *propagate* - Where the incoming failure is passed through the outports and dispatched to the subsequent component.

FPTC uses the term node to represent components and connection and assumes that edges connecting the nodes are infallible. If there are multiple inputs and outputs to a node, the transformation expression is captured using a tuple of tokens. For example, a node with two inputs and two outputs can have a transformation expression of *(late, late)* $\rightarrow$ *(value, late)*. In that expression the tuple *(late, late)* is the LHS and tuple *(value, late)* is the RHS.

The FPTC analysis is performed by computing the least upper bound using the following transfer function.

$$\mathrm{in}(\mathrm{c}_i) = \mathrm{out}(\mathrm{p}_j), \text{ where } \mathrm{p}_j = \mathrm{predecessor}(\mathrm{c}_i)$$

$$\mathrm{out}(\mathrm{c}) = \bigcup_{tup \in perms(in(c_0),\ldots,in(c_k))} \mathrm{rhs}(\mathrm{select}_{tup}(\mathrm{transforms}(\mathrm{c})))$$

Starting from the source transformation expressions, the subsequent sets are computed by copying the outs of the predecessor component. If there is more than one predecessor component, the result is the union of the predecessor out sets. The out set is computed by selecting the transformation expression based on the in set and computing the union of the RHS.

The lattice is finite in the analysis because there is only a finite number of ports and a finite number of tokens. Therefore, *powerset(Ports X Tokens)* is also finite. The transfer function computes the union of the facts at each step. Therefore it is monotonic. The LUB computed using this transfer function will produce an optimal solution.

The FPTC analysis provides the following facts:

- Possible set of tokens for all ports
- Set of transformation expression that are applied during the analysis
- List of tokens and in ports for which there is no transformation expression provided. In other words, a list of unhandled errors

4.3 Model Checking

Model-checking is a well-established technique for the verification of critical systems. Given a model and a property, a model check automatically checks whether the model satisfies the property. Most modern model checkers are capable of producing counterexamples when a property is not satisfied. The counterexample consists of the initial state of the system and a trace *i.e.* subsequent states from the initial state until the state violates the property. Usually, the properties are expressed as temporal logic. Temporal logic is a logic for expressing changing truth of propositions with respect to time[57].

The following are the model checking tools developed to perform casual analysis for a critical system.

4.3.1 Agree

The Assume Guarantee REasoning Environment (AGREE) is a compositional model checker for AADL models. In this tool, the properties are specified as an annex of AADL in an assume-guarantee style of contracts. These contracts and the model are translated to LUSTRE dataflow language[58] and verified using JKind model checker[59].

In the assume-guarantee[60] contracts, the assume part corresponds to the environmental constraints on the component and the component's invariants. The guarantee part corresponds to the component's requirement.

AGREE can perform formal verification on large systems by verifying individual subcomponents and composing the results to verify the parent component. Furthermore, this approach support AADL's architecture-based requirements refinement using the system hierarchy. In distributed development, if system properties are captured as component contracts, then virtual integration can be performed by checking the transitive conformance of the properties.

```
annex agree {**
  assume "The Flight Planner shall receive an authenticated command from the
                                      Ground Station" : recv_map.HMAC = True;
  assume "The Flight Planner shall receive a well-formed command from the
                                      Ground Station" : SW.good_gs_command(recv_map);
  guarantee "The Flight Planner shall generate a valid mission"
                                      : SW.good_mission(flight_plan);
**};
```

This example AGREE snippet illustrates the system property of authenticated and well-formed commands generating valid missions. If this property is not satisfied, AGREE provides a counterexample showing the failure case.

4.3.2 Resolute

Resolute[61] is a language and tool for developing assurance cases and for enforcing architectural constraints on AADL architecture models. Using the Resolute language, users can develop first-order predicates to capture a model property and develop rules to query architectural models to generate portions of an assurance case. Using Resolute's predicate language, there is some overlap with some of Awas reachability analysis. For example, Resolute can specify a modified version of the Query concept 4 from Section 6.3: *all the commands received by the flight controller comes from filter?*

```
-- Checks if the specified component is a filter
  is_filter(c : component) : bool =
      has_property(c, COMP_TYPE) and
      property(c, COMP_TYPE) = "FILTER"


-- there is no pathway that avoids the filter
  not_bypassed(c : component) <=
      ** "Filter cannot be bypassed" **
  -- get incoming connections of type Command
      let cmd_conns : {connection} = {conn for (conn : connections(c)) |
                                        destination_component(conn) = x and
                                        has_type(conn) and
                                        type(conn) = SW::Command};
  - all of these connections are from a filter
  forall(conn : cmd_conns).is_filter(source_component(conn))
```

This example illustrates that the notion of reachability is specified explicitly by iterating over connections. Because of its expressive scripting language, Resolute can specify general properties compared to Awas, with the burden of more verbose specification (i.e., essentially by having users to "script" the properties). Behind the scene, Resolute uses JKind

model checker to compute the query results. By default, Resolute does not utilize the intra-component dependencies and error propagation and it does not provide significant support for flow visualizations. Awas is specially designed and excels in calculating more complex reachability at scale with easily understandable visualizations as analysis feedback. Awas and Resolute are complementary tools that are used in conjunction on the DARPA CASE project.

4.3.3 AltaRica

AltaRica is a failure effect modeling-based safety analysis language that uses dataflow semantics. AltaRica can perform safety assessment, fault tree generation, and functional verification using the NuSMV model checker. The AltaRica model is composed of hierarchical nodes, each representing a component. Sibling components communicate through flows and synchronizations. The synchronizations are declared in an equipment node containing component nodes.

An AltaRica model is specified in terms of an Interface Transition System(ITS). ITS for a component node is composed of a set of states, initial conditions, state transitions, a set of events, and flow variables, and the composition of subnode ITS gives the ITS for the equipment node. This version of AltaRica has substantial tooling support from the Cecilia OCAS safety assessment platform developed by Dassault Aviation[62].

AltaRica 3.0 language is updated to support the Guarded Transition System (GTS) for easier modeling of safety properties. OSATE provides an experimental tool that supports the translation of a small subset of AADL EMv2 models into AltaRica models to perform safety assessment[63].

4.3.4 xSAP

xSAP is yet another purpose-built platform for model-based safety analysis. Similar to AADL, it provides a customizable library of fault modes and a collection of safety analyses, including FTA, FMEA, and Common Cause Analysis (CCA). These analyses are accom-

plished using advanced techniques such as property-directed reachability (IC3[64]), SAT and SMT based model checking techniques, and Binary Decision Diagram(BDD) based fault tree generator. xSAP is built on top of NUXMV symbolic model checker which is an extension of NuSMV. The NUXMV supports the verification of finite- and infinite-state systems.

xSAP has been used widely in both industry and academia, particularly as a backend for COMPASS tool developed by European Space Agency (ESA).

Chapter 5

Information Flow Framework

This chapter details the translation of interesting AADL features to Awas in section 5.1. The rest of the chapter provides the inner workings of Awas dependence analysis engine in various levels with detailed algorithm and correctness arguments in section 5.3.

5.1 AADL to Awas Graph

Awas build an internal graph representation by extracting the dependence information from the AADL instance model. Awas utilizes this graph representation in computing reachability and visualizing the model and results.

Extraction of Dependence Information and Construction of Internal Graph: Awas builds a collection of inter-connected graphs to capture the dependence relations of an AADL model. Starting from the top-level system, Awas traverses down the component hierarchy and translates each AADL non-leaf component (*i.e.* component with connected sub-components) into a graph data structure. The graph nodes consist of components, connections, and ports that are part of the parent component. The following sections describe the design decisions for graph construction for select AADL constructs.

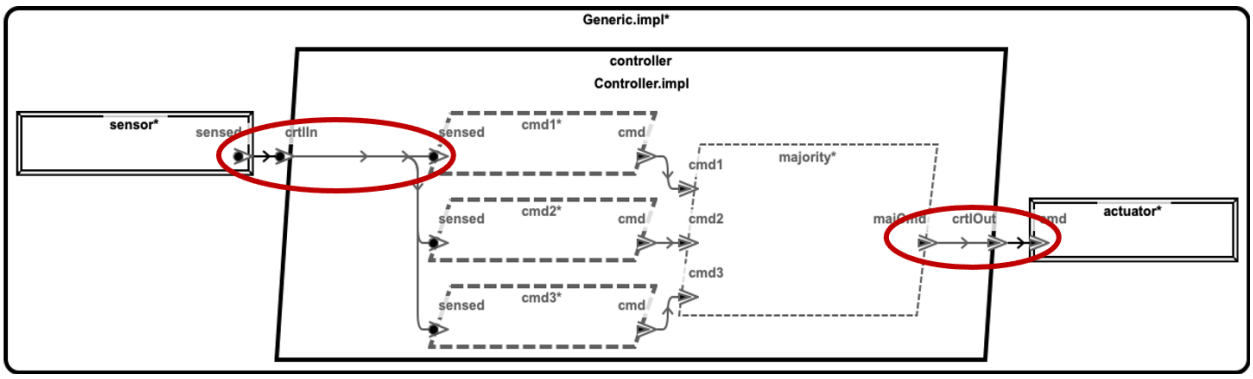


Figure 5.1: Generic triple modular redundancy system

5.1.1 Connection Instance

Figure 5.1 shows a simple triple modular redundant controller, where three different threads process a sensed value and provide their majority vote to an actuator. The top-level system consists of a sensor, a controller, and an actuator. The controller component is composed of three compute units and a majority finder.

The standardized AADL instance model creation process implemented by OSATE flattens the model hierarchy and "tunnels through" the ports, connections, and feature group structures that AADL provides to structure a system. While the resulting instance model provides a simplified structure that captures the essence of the system's threading and communication semantics (which in turn simplifies analyses and code generation), the dropped model features make it difficult in some cases to establish traceability to features in the original declarative model and to provide the user with a visualization that captures the originally designed hierarchy. Therefore, even though the Awas graph representation is based on the AADL instance model, it adds information into the graph structure to enable traceability to portions of the declarative model that are dropped in the OSATE-implemented instance model construction.

In the standardized AADL instance model construction implemented by OSATE, the ports shared between the hierarchical components and the connections through these ports are replaced with direct connections. For example, the ports `ctrlIn` and `ctrlOut` and connections involving these ports are removed and replaced with connections that directly

connect `Sensor.sensed` to `cmd.sensed` and `Majority.majCmd` to `actuator.cmd`.

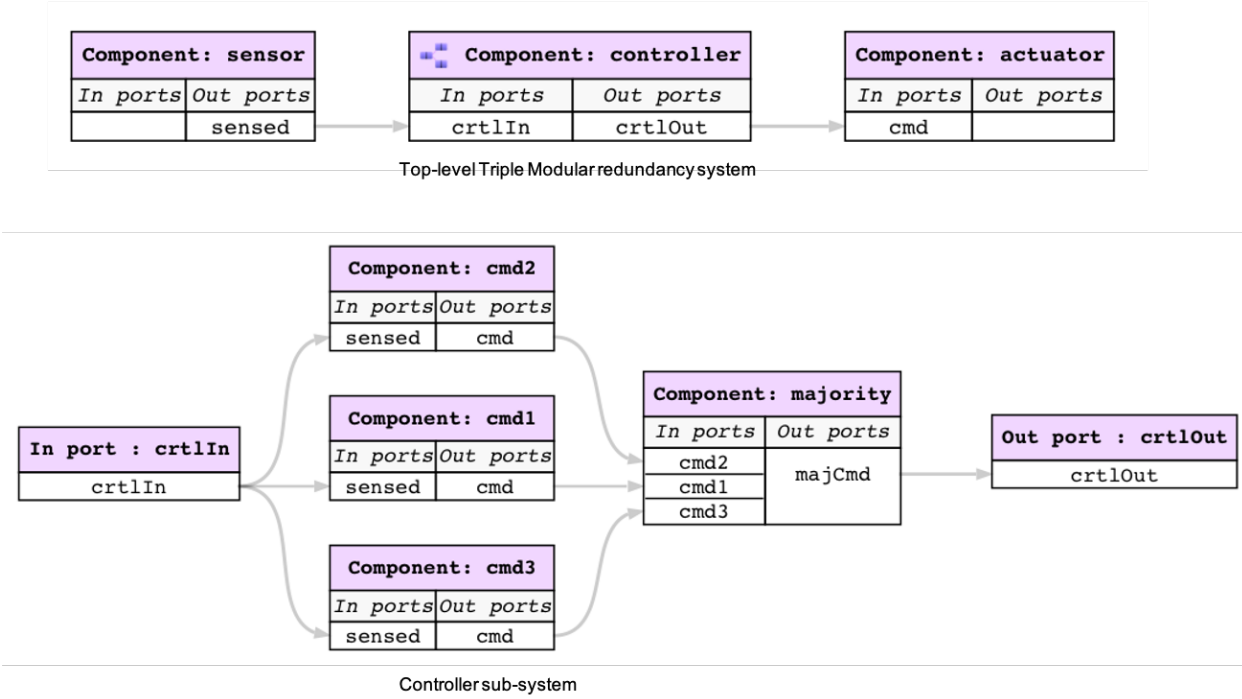


Figure 5.2: Awas graphs of triple modular redundancy system

To support clearer visualization of the port/connection structure across component boundaries, we add the ports and connections OSATE removed during the instance model construction back in the Awas graph by consulting the declarative model. A component with sub-system communicate with the sub-system graph through shared ports. These ports are represented as nodes in the sub-system graph. Figure 5.2 shows the ports `ctrlIn` and `ctrlOut` as part of the component controller in the top-level system and independent nodes in the controller's sub-system.

5.1.2 Feature Groups

In AADL, feature groups support the grouping of commonly connected ports and features to reduce the number of connections. A single connection between feature groups can capture the effect of multiple connections between individual features. Figure 5.3 provides an example of a feature group definition at lines 1-5 and a bi-directional connection between the feature groups at line 15. The feature group port in `subsystem_in` is an inverse of the feature

```
feature group fg
  features
    test_feature1 : in data port;
    test_feature2 : out data port;
end fg;

system toplevel
end toplevel;

system implementation toplevel.i
  subcomponents
    sub1 : process subsystem;
    sub2 : process subsystem_in;
  connections
    conn:  feature group sub1.fg_port <->  sub2.fg_port_reverse;
end toplevel.i;

process subsystem
  features
    fg_port :  feature group fg;
end subsystem;

process subsystem_in
  features
    fg_port_reverse :  feature group inverse of fg;
end subsystem_in;
```

Figure 5.3: Feature groups and bi-directional connection

group definition, meaning the directionality of the ports in the feature group are flipped. In AADL, feature groups can be nested within another feature group to form compact and flexible endpoints. AADL follows a dot-separated naming convention to access individual features from a feature group.

Awas eliminates the feature groups by flattening them. This entails adding the features in the feature groups directly to the components. Connections involving feature groups are removed and replaced by direct connections between the added features. In this process, Awas also eliminates bi-directional connections by replacing them with two unidirectional connections. For example, for `conn` in Figure 5.3, Awas generates two connections `sub1.fg_port.test_feature2 ->`

`sub2.fg_port_reverse.test_feature2`, and

`sub2.fg_port_reverse.test_feature1 ->`

`sub1.fg_port.test_feature1`.

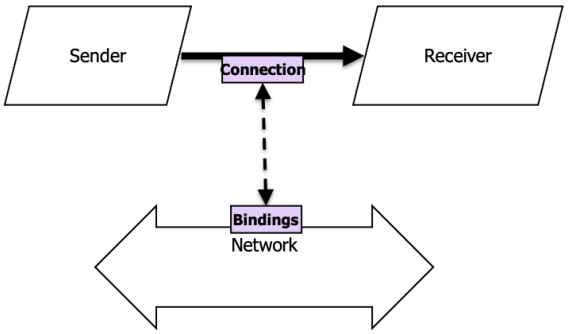


Figure 5.4: Network bus realizing a connection between a Sender and a Receiver

5.1.3 Bindings

Bindings are AADL constructs that associate hardware components with their software counterparts. For example, a processor on which one or more process(es) execute on or a bus that facilitates (or realizes) connections between components. Figure 5.4 shows the binding relation between a connection and a `Network` bus. In EMv2, there are special keywords to capture the flow of errors through the bindings. These constructs pose several challenges in building a reachability analysis framework.

- Bindings are the only component that allows information flow outside of the containment hierarchy (component sub-component relation).
- Bindings create cycles between the dependent and dependee.
- Multiple connections may be bound to the same network, and the reachability analysis may get into the bus through one connection and exit via another connection even if the network has a channel separation.

Awas reachability analysis maintains a mapping of the bindings relations. Due to the challenges mentioned above, by default, these relations are ignored in the dependence analysis.

However, users are presented with an option to choose whether to respect the binding relationship in reachability computation. By opting to respect the bindings relation, the analysis has to be performed on the whole system.

5.2 Awas Graph Definitions

Awas internal graph is formalized via the following definitions:

Definition 1 Awas graph is a triple, $\mathcal{A} = (\sum_{\mathcal{G}}, \rightarrow_{\mathcal{B}}, \sum_{Error})$, where

- $\sum_{\mathcal{G}}$ is a set of graphs, each representing a (sub-)system
- $\rightarrow_{\mathcal{B}} \subseteq \sum_{\mathcal{P}} \times \sum_{\mathcal{P}}$ is the binding relation between two ports capturing the dependencies between a hardware and software component.
- $\sum_{Error}$ is the set of all error tokens from the error library defined for the AADL model

Definition 2 A graph $\mathcal{G} \in \sum_{\mathcal{G}}$consists of a set of nodes(vertices) $\sum_{\mathcal{N}}$ and a set of edges $\sum_{\mathcal{E}}$.

Definition 3 A node is a quintuple, $\mathcal{N}= (\sum_{\mathcal{P}}, \rightarrow_{\mathcal{F}}, kind, \mathcal{G}_{sub}, \mathcal{N}_{p})$, where

- $\sum_{\mathcal{P}}$ is a set of ports in a component, connection, or port node. Each port is associated with a IN or OUT token representing the port's role in either bringing the information into the component or out. In components where error propagation is defined, each port is also related to a set of errors $\sum_{Error}$. In the case of a port node, the $\sum_{\mathcal{P}}$ contains one element and points to the parent graph's corresponding port.
- $\rightarrow_{\mathcal{F}} \subseteq \sum_{\mathcal{P}} \times \sum_{\mathcal{P}}$ is the intra-component dependency relation between the in ports and out ports. In error flows, $\rightarrow_{\mathcal{F}} \subseteq (\sum_{\mathcal{P}} \times \sum_{Error}) \times (\sum_{\mathcal{P}} \times \sum_{Error})$. Awas transforms EMv2 error flows with multiple error tokens into multiple flows with each form (port, Error) $\rightarrow$ (port,error) relation.
- *kind*, captures the node type. A node can represent a component, connection, or a port from the parent component.

- $\mathcal{G}_{sub}$ is a graph representing the sub-system for node $\mathcal{N}$.
- $\mathcal{N}_p$ is a node from the parent graph. If $\mathcal{N}$ is part of a sub-system graph, $\mathcal{N}_p$ represents the node that abstracts the graph containing $\mathcal{N}$.

Definition 4 An edge $\mathcal{E} \in \sum_{\mathcal{E}}$, consists of two binary relations of the form $\rightarrow_{\mathcal{N}} \subseteq \sum_{\mathcal{N}} \times \sum_{\mathcal{N}}$ and $\rightarrow_{\mathcal{P}} \subseteq \sum_{\mathcal{P}} \times \sum_{\mathcal{P}}$. The analysis picks appropriate relations depending on the granularity of the analysis.

5.3 Dependence Analysis

The following primary objectives shaped the implementation of the Awas dependence analysis framework.

- Ability to analyze large scale system models and provide responsive feedback
- Ability to visualize and query both abstract early designs as well as rich models built later in the development process that include AADL annotations (e.g., error behavior and fault propagation)
- Provide a portable visualization of the system model that is independent of the modeling environment
- Ability to provide high fidelity results when the model contains flows and error behavior

In Awas, the dependency analysis problem is treated as a reachability analysis on the internal graph. The reachability algorithms are independent of the underlying graph representation. In large models where in-memory graph is not sufficient, we store the graphs on disks without impacting the scalability of the analysis.

A user performs the dependency analysis by issuing a query to Awas. A query can be either from the user interface or a Domain Specific Language (DSL) statement. A query consists of reachability from or to an element at its core. The Awas framework selects from among several analysis algorithms based on the query. Awas communicates the result in the

same granularity as the query criterion. The user can pose reachability queries regarding components, connections, ports, or errors propagating through a port. When a user queries about a component's dependency, Awas invokes a node-level reachability analysis and produces results in terms of components and connections. However, if the query criterion is in terms of ports, Awas performs port level reachability analysis. In the rest of this section, we use the term element in place of component, port or port error combination.

5.3.1 Node-level Analysis

The atomic step in computing the reachability of an element is to compute the next reachable element. In a forward reachability analysis, the subsequent elements are computed by calculating the successor elements, and similarly, a backward analysis computes predecessor elements.

Algorithm 1: successor_node

input : node : $\mathcal{N}$, useBindings : Boolean
Result: (N, E)
N $\leftarrow \varnothing$ `// initialize successor node`
E $\leftarrow \varnothing$ `// initialize successor edge`
if *node.kind = PORT* ***and*** *node.port is OUT direction* **then**
  $E \leftarrow$ node.parent outgoing edge with port as source
  $N \leftarrow$ E's target node
else `// component, connection or IN port node`
  $E \leftarrow$ node's outgoing edges
  $N \leftarrow$ E's target nodes
  if *useBindings* **then**
    $bN \leftarrow$ target nodes from binding relation($\rightarrow_{\mathcal{B}}$)
    $N \leftarrow N \cup bN$
  end
end

Algorithm 1 provides the pseudo-code for computing the successor nodes for a given node. If the given node represents an outgoing port, the corresponding port's successor in the parent graph is returned. In other cases, the analysis computes the successor based on the edge relation. If the user considers the bindings relation, the successor nodes include the nodes reachable by the binding relation.

Algorithm 2: predecessor_node

input : node : $\mathcal{N}$
Result: (G, N, E)
N $\leftarrow \varnothing$ `// initialize successor node`
E $\leftarrow \varnothing$ `// initialize successor edge`
if *node.kind = PORT* ***and*** *node.port is IN direction* **then**
  $E \leftarrow$ node.parent incoming edge with port as target
  $N \leftarrow$ E's source node
else
  `// component, connection or in port node`
  $E \leftarrow$ node's incoming edges
  $N \leftarrow$ E's source nodes
  if *useBindings* **then**
    $bN \leftarrow$ source nodes from binding relation($\rightarrow_{\mathcal{B}}$)
    $N \leftarrow N \cup bN$
  end
end

Algorithm 2 provides the pseudo-code for computing the predecessor nodes. It is similar to computing the successor nodes other than the inverse transitivity.

Algorithm 3 provides the pseudo-code for a worklist-based reachability analysis algorithm at the node level. The analysis has two steps. In the first step, the analysis calculates the reachable nodes within the graph and its parent graphs. In the second step, for each node in the result of step 1 that has a sub-graph, the analysis recursively descends and collects all the nodes. We memoize the result of step 2 to avoid recomputing the descendants of a node.

The algorithm 3 computes the least upper bound on the power set of nodes on each graph with $\sqcup = \cup$ as the join operator. Because of the finite set of nodes in the graph and the lattice formed by the power set, the lattice has a finite ascending chain length.

The transfer function of the node-level reachability is :

$$R_N(C) = \begin{cases} \bigcup_{c \in C} successor\_node(c) & \text{if isForward = true} \\ \bigcup_{c \in C} predecessor\_node(c) & else \end{cases} \tag{5.1}$$

Because the transfer function is monotonic, there exists a fixed point. Based on the

Algorithm 3: reach_node

input : criterions : $\sum_{\mathcal{N}}$, isForward : Boolean
Result: (N, E)
E ← ∅ `// initialize result edge`
N ← ∅ `// initialize result node`
W ← *criterions* `// initialize worklist`
while *exists* *currentNode* ∈ *W* **do**
  N ← N ∪ currentNode
  W ← W − currentNode
  if *isForward* **then**
    (nextNodes, nextEdges) ← successor_node(currentNode)
  else
    (nextNodes, nextEdges) ← predecessor_node(currentNode)
  end
  E ← E ∪ nextEdges
  for *nNode* ∈ *nextNodes* **do**
    if *nNode* ∉ *N* **then**
      W ← W ∪ nNode
    end
  end
end
W2 ← N
`// adding sub graph nodes`
while *exists* *node in W2* **do**
  W2 ← W2 - node
  if *node has subgraph* **then**
    W2 ← W2 ∪ node.subgraph's nodes
    N ← N ∪ node.subgraph's nodes
    E ← E ∪ node.subgraph's Edges
  end
end

Knaster-Tarski theorem defined in section 4.1, the computed fixed point is the least fixed point, and the solution is optimal. The time complexity of the node-level analysis is bounded by $O(\sum_{\mathcal{N}})$.

The correctness of node-level analysis can be defined as each node in the result set is reachable from the criterion node by following the successor node or predecessor node relation. This can be inductively proven using the induction hypothesis *Hyp(k)*: Each node added to the result set at iteration k' such that $k' \leqslant k$ is reachable. In base case when $k = 0$, the result contains the criterion meaning a node is reachable from itself. In the inductive step, we assume that for k iteration *Hyp(k)* holds. To prove *Hyp(k+1)*, the new nodes computed at $k + 1$ iteration from the intermediate results computed at k iteration belong to the relationship defined by the successor node or predecessor node. Because the next and previous node relations are user specified in terms of connections, the induction steps follows immediately.

5.3.2 Port-level Analysis

Like the node level analysis, the port-level reachability analysis requires the successor and predecessor ports for a given port. Algorithm 4 provides a pseudo-code for computing the successor ports for a given port. Depending on the kind of node and the direction, it computes the successor using the flow and edge relations. If the intra-component dependency for a port is not defined, the analysis conservatively assumes that there may be a flow to all output ports. Like the predecessor_node, the predecessor_port differs from the successor_port by traversing using the inverse of the edge and flow relation.

The pseudo-code provided in Algorithm 5 computes port-level reachability within a graph. The *climbUp* flag dictates if the analysis flows into the parent graph and its ports. Likewise, the *climbDown* flag decides the exploration of ports in the child graph. This ability to restrict reaching into sub-graphs enables compositional analysis.

To compositionally perform the analysis, we compute the intermediate results, in this case, the information flow in the sub-graph captured by a flow. For a flow with both source

Algorithm 4: successor_port

```
input  : port : 𝒫,
         climbUp, climbDown, useBindings : Boolean
Result: (P, F, E)
P ← ∅ // initialize successor ports
F ← ∅ // initialize successor flows
E ← ∅ // initialize successor edges
node ← node containing port
if port' direction is IN then
  if port.flows ≠ ∅ then // flows with port as source and target ≠ ∅
    P ← port.flows // target of the flows
    F ← node.→_F where, flow's source = port
  else // conservative choice
    if node.subgraph = ∅ then
      P ← OUT ports of the node
    end
  end
  if climbDown and node.subgraph ≠ ∅ then
    port_sub ← find subgraph's ports = port
    E ← E ∪ outgoing edges of node where port_sub is source
    P ← P ∪ E's targets
  end
else // port's direction is OUT
  if node.kind = PORT and climbUp then
    parentPort ← port.parent
    parentNode ← parentPort.node
    E ← E ∪ outgoing edges of parentNode where parentPort is source
    P ← P ∪ E's targets
  else
    E ← E ∪ outgoing edges of node where port is source
    P ← P ∪ E's targets
  end
end
if useBindings and port.bindings ≠ ∅ then
  P ← P ∪ port.bindings targets
end
```

and target defined, the flow reachability computes the forward reachability from the flow's source port intersected with backward reachability from the flow's target port. The result includes the ports, flows, and edges from source to target in the sub-graph. For each flow, the flow reachability can be computed once and used in future queries.

The complete port-level reachability is computed based on the user's choice of including

Algorithm 5: reach_ports_partial

```
input  : criterions : ∑_P,
         isForward, climbUp, useBindings : Boolean
Result: (P, F, E)
P ← ∅ // initialize result ports
F ← ∅ // initialize result flows
E ← ∅ // initialize result edges
W ← criterions // initialize worklist
while exists currentPort ∈ W do
  P ← P ∪ currentPort
  W ← W − currentPort
  if isForward then
    (nextPorts, nextFlows, nextEdges) ←
        successor_port(currentNode,
                       climbUp,
                       climbDown = false,
                       useBindings)
  else
    (nextPorts, nextFlows, nextEdges) ←
        predecessor_node(currentNode,
                         climbUp,
                         climbDown = false,
                         useBindings)
  end
  F ← F ∪ nextEdges
  E ← E ∪ nextEdges
  for nPort ∈ nextPorts do
    if nPort ∉ P then
      W ← W ∪ nPort
    end
  end
end
```

the bindings relation. Traversing through a binding relation leads to a graph outside of the containment hierarchy(parent-child system). Therefore, each query traverses all the reachable ports. However, ignoring the binding relation enables an efficient two-step algorithm: Step (1) compute the reachable ports within the sub-graph and its ancestors; Step (2) for each flow in the results of Step 1, compute the flow reachability to compute the reachable ports in the decedent graphs. In the absence of intra-component flow dependencies, the analysis computes a conservative result.

Although the port-level analysis had an additional level of details due to flows, it is very

Algorithm 6: flow_reach

```
input  : flow : ∑_F
Result: (P, F, E)
P ← ∅ // initialize successor ports
F ← ∅ // initialize successor flows
E ← ∅ // initialize successor edges
node ← node containing flow
if node.subgraph ≠ ∅ and
flow.sourcePort ≠ ∅ and
flow.targetPort ≠ ∅ then
    sourcePort ← find in node.subgraph's where
                 port = flow.sourcePort
    targetPort ← find in node.subgraph's where
                 port = flow.targetPort
    (sPort, sFlow, sEdge) ←
        reach_ports_partial(sourcePort,
                            isForward = true,
                            climbUp = false,
                            climbDown = false,
                            useBindings = false)
    (tPort, tFlow, tEdge) ←
        reach_ports_partial(targetPort,
                            isForward = false,
                            climbUp = false,
                            climbDown = false,
                            useBindings = false)
    P ← sPort ∩ tPort
    F ← sFlow ∩ tFlow
    E ← sEdge ∩ tEdge
end
```

similar to the node-level analysis at its core. A simplified version of the transfer function is:

$$R_P(C) = \begin{cases} \bigcup_{c \in C} successor\_port(c) & \text{if isForward = true} \\ \bigcup_{c \in C} predecessor\_port(c) & else \end{cases} \tag{5.2}$$

In the flow_reach algorithm, we restrict the analysis to one graph at a time. With the finite number of ports in each graph, the fixed point over the transfer function terminates and is wellfounded. The time complexity is similar to the node-level analysis and bounded by $O(\sum_{\mathcal{P}})$. In large graphs, Awas performs node level analysis first, and then the port level analysis within the results of the node level analysis.

Algorithm 7: reach_ports

```
input  : criterions : ∑_P,
         isForward, useBindings : Boolean
Result: (P, F, E)
P ← ∅ // initialize result ports
F ← ∅ // initialize result flows
E ← ∅ // initialize result edges
fW ← criterions // initialize worklist
if useBindings then
    (P, F, E) ← reach_ports_partial(criterions,
                    isForward,
                    climbUp = true,
                    climbDown = true,
                    useBindings)
else
    (P, F, E) ← reach_ports_partial(criterions,
                    isForward,
                    climbUp = true,
                    climbDown = false,
                    useBindings = false)
    fW ← F
    while exits flow ∈ fW do
        fW ← fW - flow
        (Pf, Ff, Ef) ← flow_reach(flow)
        fW ← fW ∪ Ff
        P ← P ∪ Pf
        F ← F ∪ Ff
        E ← E ∪ Ef
    end
end
```

5.3.3 Error Propagation Analysis

The error reachability analysis is complementary to the port reachability analysis. The reachable ports computed in the error reachability is a subset of the port reachability analysis. Therefore, Awas computes the error reachability using the results of the port reachability.

Algorithm 8 describes the computation of the successor port-error pairs for a given port and an error. If the component lacks error flow information, the analysis resorts to port reachability. Composing the error reachability from the successor error is similar to the port reachability.

In error reachability the *powerset of* $(\sum_{\mathcal{P}} \times \sum_{\text{Error}})$ forms the lattice elements. In large

models, this set can have several thousand elements, and yet it is finite. Therefore the fixed point using the algorithm 8 will terminate and yields the optimal solution. The correctness of the port level and error level analysis is similar to the node level analysis. Where the successor/predecessor port/error is defined by the user in-terms of connections and flows.

Awas performs reachability analysis in varying granularity. The component-level analysis is a traditional graph reachability analysis that includes all the sub-components nodes in the reachable paths. The next level is the port level analysis using the intra-component flows. An intra-component flow captures the effect of reachability through its sub-components. In an analysis without the binding relation, Awas computes reachability compositionally and caches the intermediate results to speed up future analysis. In the absence of intra-component flows, Awas assumes that all outgoing ports are reachable from any incoming ports.

When the user is interested in enumerating individual paths, the analysis first computes the unified result for all paths using the above-described algorithms. From the results, Awas extracts individual paths. In the absence of intra-component flows, the path enumeration analysis computes exponential paths that are not useful to the end-user. Awas also includes simple paths and paths with cycles. If multiple cycles are associated with a simple path, Awas produces a path with all the cycles associated with it. We made this choice to avoid overwhelming users with analysis results that are not useful. Awas provides a filtering mechanism on the paths to extract a specific path of interest. Section 6.3 demonstrate the filtering capability in detail.

5.3.4 State Reachability

Apart from the error propagation and error flows, EMv2 has two other error behavior constructs.

- Component error behavior:
  - Set of error states a component may go to from the operational state

Algorithm 8: successor_port_error

```
input  : port : 𝒫, error: Error,
         climbUp, climbDown, useBindings : Boolean
Result: ((P, Error), F, E)
(P, Error) ← ∅ // initialize successor ports
F ← ∅ // initialize successor flows
E ← ∅ // initialize successor edges
node ← node containing port
if port' direction is IN then
    if (port, error).flows ≠ ∅ then
        (P, Error) ← (port, error).flows // target of the flows
        F ← node.flows where flow source = (port, error)
    else // conservative choice
        if node.subgraph = ∅ then
            for (Out_P, Out_E) in OUT propagations do
                (P, Error) ← (P, Error) ∪ (Out_P, Out_E)
            end
        end
    end
    if climbDown and node.subgraph ≠ ∅ then
        port_sub ← find subgraph's ports = port
        E ← E ∪ outgoing edges with port_sub as source
        (P, Error) ← (P, Error) ∪ (E's target, Error)
    end
else if port' direction is OUT and
useBindings and port.bindings ≠ ∅ then
    (P, Error) ← (P, Error) ∪ port.binding's (targets, error)
else // port's direction is OUT
    if node.kind = PORT and climbUp then
        parentPort ← port.parent
        parentNode ← parentPort.node
        E ← E ∪ outgoing edges of parentNode where parentPort is source
        (P, Error) ← (P, Error) ∪ (E's target, error)
    else
        E ← E ∪ outgoing edges of node where port is source
        for edge ∈ E do
            if edge.target propagation ≠ ∅ then
                if error ∈ edge.target propagation then
                    (P, Error) ← (P, Error) ∪ (edge.target, error)
                else
                    for tError ∈ edge.target propagation do
                        (P, Error) ← (P, Error) ∪ (edge.target, tError)
                    end
                end
            else
                warning: insufficient information
                (p, f, e) ← successor_port(port)
                (P, Error) ← (P, Error) ∪ (p, ∅)
                F ← F ∪ f
                E ← E ∪ e
            end
        end
    end
end
```

- Set of possible error events that may get triggered over the lifetime of a component
- Set of state transitions
- Set of error propagations

- Composite error behavior: defines the parent component error state based on subcomponents error state

The state reachability analysis is another version of FPTC analysis utilizing the component error behavior. This analysis answers whether the system can reach an unsafe state. The state here represents a particular combination of $(\sum_{\mathcal{P}} \times \sum_{Error})$ or component error states. If the unsafe state is reachable, the analysis provides the set of transitions, port errors, and error states for each component that leads to an unsafe state. In essence, this analysis is similar to the cone-of-influence computation in model checking.
A state transition is of the form:

```
Id : source error state -[error condition]-> target error state
```

```
error_condition ::=
  condition_trigger | ( error_condition )
  | error_condition and error_condition
  | error_condition or error_condition
  | numeric_literal ormore ( condition_trigger ( , condition_trigger )+ )
  | numeric_literal orless ( condition_trigger ( , condition_trigger )+ )
```

Figure 5.5: Error condition

Figure 5.5 provides the syntax for the error condition where the condition_trigger can be an error event or an element from $\sum_{\mathcal{P}} \times powerset(\sum_{\text{Error}})$.
An error propagation is of the form:

```
Id : source error state -[error condition]-> propagation target
```

where the propagation target $\subseteq \sum_{\mathcal{P}} \times powerset(\sum_{\text{Error}})$. Figure 5.6 is a sample error behavior for the PCA interlock app component. Where the first two transitions are constrained based on the incoming error on port *SpO2* and the third one is constrained on the error event *SoftwareFailure*.

```
component error behavior
    transitions
        EarlySensor: Normal -[SpO2 {SpO2Early}]-> Normal;
        EarlySensorDetect: Normal -[SpO2 {SpO2Early}]-> DetectedError;
        SoftwareRealtedIssues: Normal -[SoftwareFailure]-> UnDetectedError;
    propagations
        InadvertantPump: Normal -[SpO2 {SpO2ValueHigh}]-> CmdPumpNorm {TicketTooLong};
        MissingInfo: Normal -[SpO2 {NoSpO2, SpO2Late}]-> CmdPumpNorm {NoError};
        EarlyInfo: DetectedError -[]-> CmdPumpNorm {LateTicket};
        SoftwareIssues: UnDetectedError -[]-> CmdPumpNorm {TicketTooLong, EarlyTicket, LateTicket};
```

Figure 5.6: Error behavior of PCA interlock app

Definitions:

- Store σ: is a mapping of the form port $\rightarrow \sum_{Error}$ and component $\rightarrow$ set of error states
- ι: initial state of the system, where each port is mapped to empty error set and each component mapped to a set containing operational state
- Union of store

$$\sigma' \cup \sigma'' = \begin{cases} p \rightarrow \sigma'(p) \cup \sigma''(p) & \text{if p} \in \text{domain}(\sigma') \text{ and p} \in \text{domain}(\sigma'') \\ p \rightarrow \sigma'(p) & \text{if p} \in \text{domain}(\sigma') \text{ and p} \notin \text{domain}(\sigma'') \\ p \rightarrow \sigma''(p) & \text{if p} \notin \text{domain}(\sigma') \text{ and p} \in \text{domain}(\sigma'') \end{cases} \tag{5.3}$$

- union is closed under σ

The analysis uses two evaluation functions on the component error behaviors namely *strongest-postcondition(sp)* and *weakest-precondition(wp)*. The sp is defined as:

$$sp(t, \sigma) = \sigma' \begin{cases} \sigma = \sigma' & \text{when } \sigma \not\vdash \text{a} \\ \sigma = \sigma \cup \text{target error state} & \text{when } \sigma \vdash \text{a and t is a state transition} \\ \sigma = \sigma \cup \text{propagation target} & \text{when } \sigma \vdash \text{a and t is a error propagation} \end{cases} \tag{5.4}$$

where,

- t $\in$ T the set of all behavior expressions involving either state transition or error propagation
- a is the interpretation of source error state and error condition in σ

Similarly wp is defined as:

$$wp(t,\sigma) = \sigma' \begin{cases} \sigma = \sigma' & \text{when } \sigma \not\vdash \text{range(t)} \\ \sigma = \sigma \cup domain(t) & \text{when } \sigma \vdash \text{target error state and t is a state transition} \\ \sigma = \sigma \cup domain(t) & \text{when } \sigma \vdash \text{propagation target and t is a error propagation} \end{cases} \tag{5.5}$$

where,

- domain(t) = source error state and error condition
- range(t) = either target error state or propagation target

The analysis first performs a simplified version of FPTC analysis named *forward* from the ι state using the *strongest-postcondition* function. Similarly, from the user-provided criterion, a backward analysis is performed using the *weakest-precondition*. Combining these two analyses results in a unified store representing a union of all states from the initial state to the criterion.

The forward analysis is defined as:

$$forward(\sigma) = \begin{cases} \bigcup_{t \epsilon T} sp(t,i) & \text{if } \sigma = \varnothing \\ \bigcup_{t \epsilon T} sp(t,\sigma) & else \end{cases} \tag{5.6}$$

The backward analysis is defined as:

$$backward(\sigma) = \begin{cases} \bigcup_{t \epsilon T} wp(t,c) & \text{if } \sigma = \varnothing \\ \bigcup_{t \epsilon T} wp(t,\sigma) & else \end{cases} \tag{5.7}$$

Algorithm 9: state_reach

```
input  : C : (P, ∑_Error, error_states)
Result: σ'
isForward ← false
σ ← iota
σ' ← forward(σ)
while σ ≠ σ' do
    σ' ← σ
    if isForward then
        σ' ← forward(σ) ∩ σ
        ¬isForward
    else
        σ' ← backward(σ) ∩ σ
        ¬isForward
    end
end
```

Algorithm 9 ensures that if the result is not empty, then the unsafe state is reachable from the initial state, and the transitions and intermediary states are reachable from an initial state and contribute to the causation of the unsafe state. This algorithm's intuition is similar to that of a program chopping[65] where the initial state ι acts as the source, and the unsafe state acts as the target. Performing a forward reachability after the backward reachability eliminates the state that is not reachable from the initial state. Similarly, a backward after a forward eliminates the states that are a superset of the final state.

The correctness is argued in two steps:

1. Forward and backward analysis terminates and the solution is optimal

2. State_reach analysis terminates and is optimal

Table 5.1 provides the important information for showing the correctness of the algorithm. The forward and backward analysis always adds information to the result. Because of the finite size of the store and the length of the ascending chain, the least upper bound on the forward/backward analysis is optimal, and it terminated. However, the state_reach is interesting because it computes the greatest lower bound (GLB). The first forward analysis computes all the facts that are reachable from the initial state. This result is considered as $\top$ for the rest of the analysis. Subsequently, the analysis alternated between the forward

	stateReach	Backward/Forward
Initial State	⊤ (greatest element)	⊥ (least element)
Join/meet operator	∩ (intersection)	∪ (union)
Fixpoint	Greatest lower bound	Least upper bound

Table 5.1: Insight into state reachability analysis

and backward in each step, but the results are restricted to the previous result. Meaning, the analysis can only remove the information, but it can never add more facts. Thus it will terminate, and the solution is optimal.

The time complexity of both forward and backward analysis is linear because they are bounded by $O(\sigma)$, where $\sigma = \sum_{\mathcal{P}} \times \sum_{Error} \cup \sum_{\mathcal{N}} \times$ error states. The state reach algorithm works on the same set of data, but it reduces the size at each step. Therefore the overall time complexity is bounded by $O(|\sigma|^2)$

Chapter 6

Awas Visualization and Querying

This chapter presents a high-level overview of the Awas dependency analysis and querying tool in section 6.1. The next section 6.2 presents the visualization of models in a web browser capable of focusing and viewing interested parts of the model. Finally, section 6.3 demonstrates different querying capability using the UAS model described in section 3.1.

6.1 Tool Architecture

Figure 6.1 presents the Awas tool implementation architecture. There are three parts to the Awas tool Architecture: (1) an OSATE plugin that consumes an AADL instance Model and translates it into a JavaScript Object Notation (JSON) formatted text representation called AIR (AADL Intermediate Representation), (2) Awas dependency graph builder and reachability analysis engine, and (3) model visualizer and Awas Query interpreter.

A key feature of representing the AADL instance model as an AIR model is facilitating the exchange of AADL models between front-end and back-end tools in a language-independent format. Furthermore, AIR enables loose coupling between Awas and OSATE modeling environment, thus providing the opportunity to support reachability on other modeling languages using Awas.

The Awas dependence graph is built using Slang[66] – a subset of the Scala programming

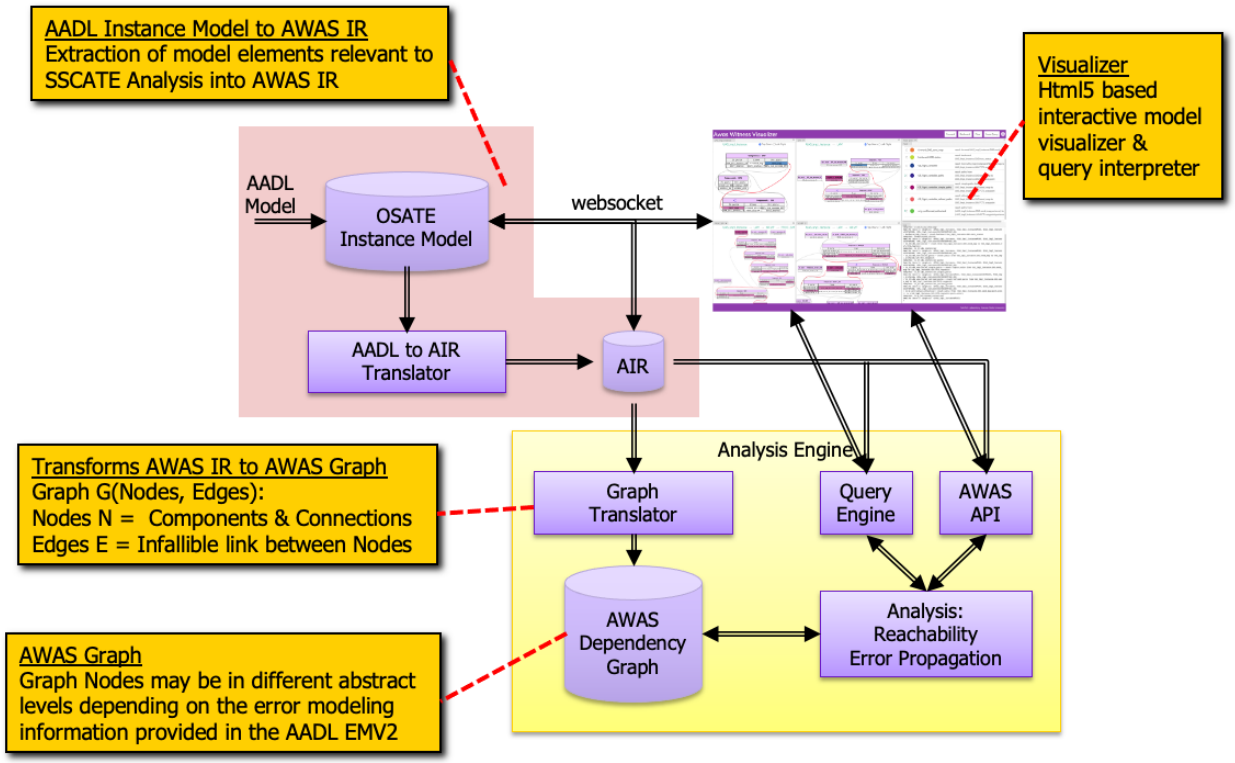


Figure 6.1: AADL reachability analysis tool architecture

language designed for engineering high assurance safety/security-critical systems. We use Scala[67] – a JVM-based language to implement the reachability analysis algorithms. Apart from Scala APIs, we also provide Java APIs to support our reachability engine within the OSATE development environment. Using the Java APIs, the user can perform regression testing on the model.

All of our analysis engines, including the Awas graph builder, query language interpreter, and the reachability algorithms, are translated into JavaScript using ScalaJS[68], a Scala to JavaScript translator.

6.2 Visualizer

For a given AADL instance model, Awas generates an HTML5-based interactive visualization. Awas algorithms, developed in Scala, are compiled to JavaScript and run directly in the browser – allowing queries and analyses to be executed independently of OSATE or other

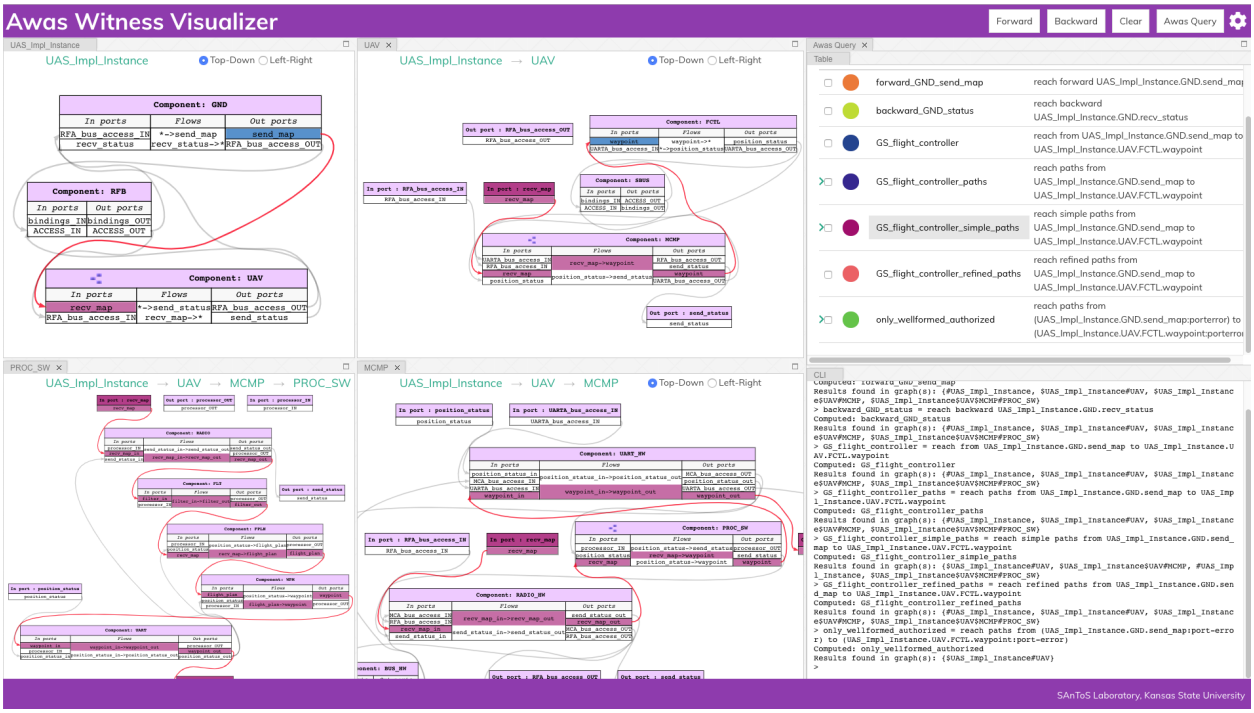


Figure 6.2: Awas reachability visualizer and query interpreter

tool components requiring installation. Figure 6.2 shows an example visualization. Multiple panes can be opened to show dependencies and analysis results at different levels of the system architecture. The user can immediately launch and view various forward and backward reachability analysis forms by selecting components, ports, and connections. Views can be configured at various level of detail (e.g., focusing on connection dependencies, adding dependencies related to AADL bindings, adding AADL EMv2 error flow information). In addition, Awas provides a query language (illustrated in the right pane of Figure 6.2) that allow complex queries to be specified and easily replayed with a single click. The visualizer provides a dynamic read-eval-print-loop for the Awas query language (bottom right). Our industrial partners have found the HTML5-based Awas visualizers to be especially useful because they allow a system description to be easily distributed via the web or a self-contained zip file so that stakeholders can browse the architecture and its dependencies without having to install the complete OSATE infrastructure and associated models.

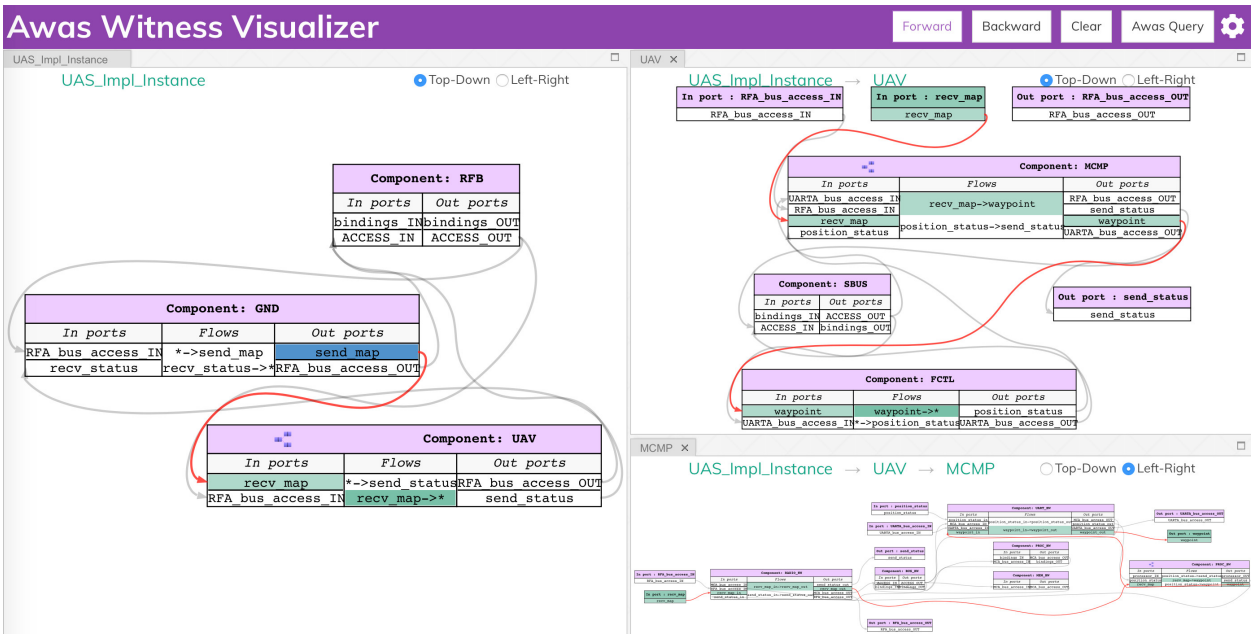


Figure 6.3: Awas Visualization of a Forward Slice (interactive forward dependence query)

6.2.1 Base Awas Dependence Graph

In the in-memory graph representation, both components and connections are represented as nodes. The edges connecting the nodes are considered infallible or passive similar to FPTC[69]. To interact with the visualization of the underlying dependence graph, users click on a component or a port and press a button to carry out basic queries such as "where in the system does information from this port/component flow?" (forward reachability in the dependence/constraint graph) or, "what system elements are contributing information that flows into this port/component?" (backward reachability).

The visualization in figure 6.3 shows the results of the user selecting thee `send_map` port (in blue) of the Ground Station and pressesing the *Forward* button to invoke the dependence analysis. The visualizer displays paths (in red) and associated ports and connections (in green) along which the information flows in the system. The visualization allows one to open multiple windows to show the results at different levels of the system hierarchy; the left window in Figure 6.3 shows the top-level of the system, while the right two windows show the UAV and its mission computer sub-systems). The scroll wheel on the mouse can be used to zoom into a particular system section or component of interest and double-clicking on a

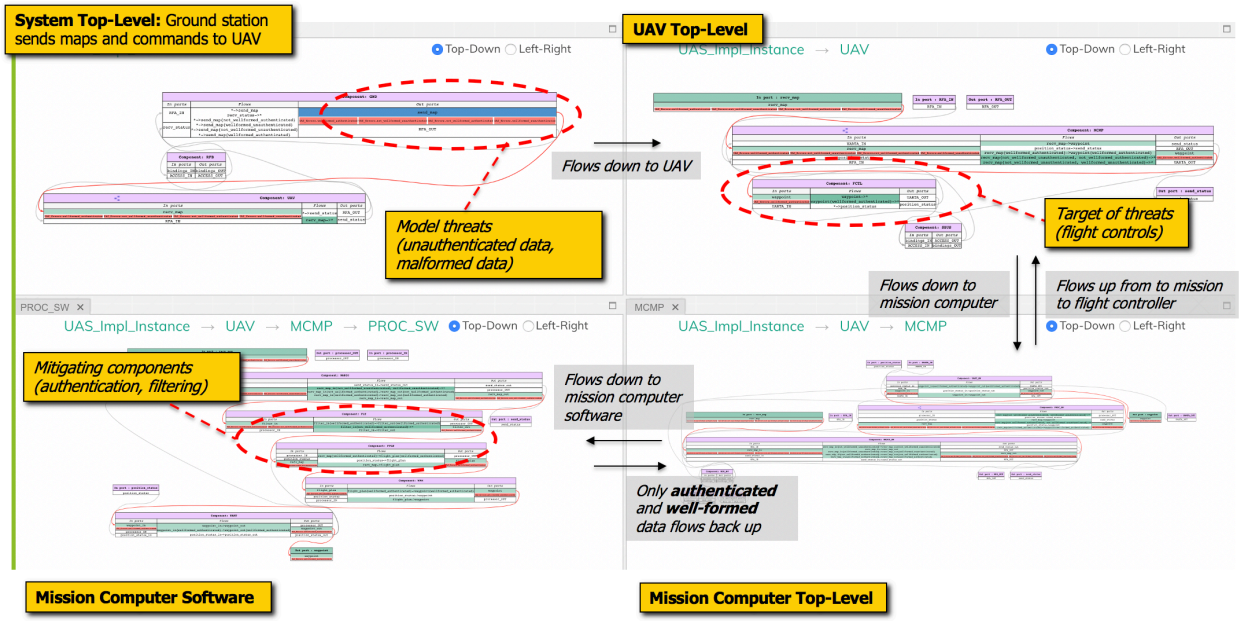


Figure 6.4: Awas Visualization of AADL EMv2-based Security Properties (Overview)

component opens the component's sub graph.

In AADL, a component can be refined by a sub-system where the information from a component's input port descend into the sub-system and ascend back through the output port. The intra-component flows defined in a component summarize the information flow in the sub-system. In Awas, each system is expressed by a graph. In the case of the sub-system, the Awas graph includes the parent component's ports as nodes in the graph. Using these port-nodes, a sub-system graph is connected to its parent component's graph.

6.2.2 Property Propagation Graph

Awas supports different forms of analyses that are layered on top of the base graphs and visualizations described in Section 6.2.1. One such layering is the support for the AADL Error Modeling (EMv2) annex. In previous work we illustrated how AADL EMv2 and Awas could support safety analysis and risk management of medical devices[70]. This section summarizes how Awas can support visualizations and analysis of security policies with properties of the policy captured using AADL EMv2.

Section 3.1.2 illustrated how AADL EMv2 specifications could be used to capture au-

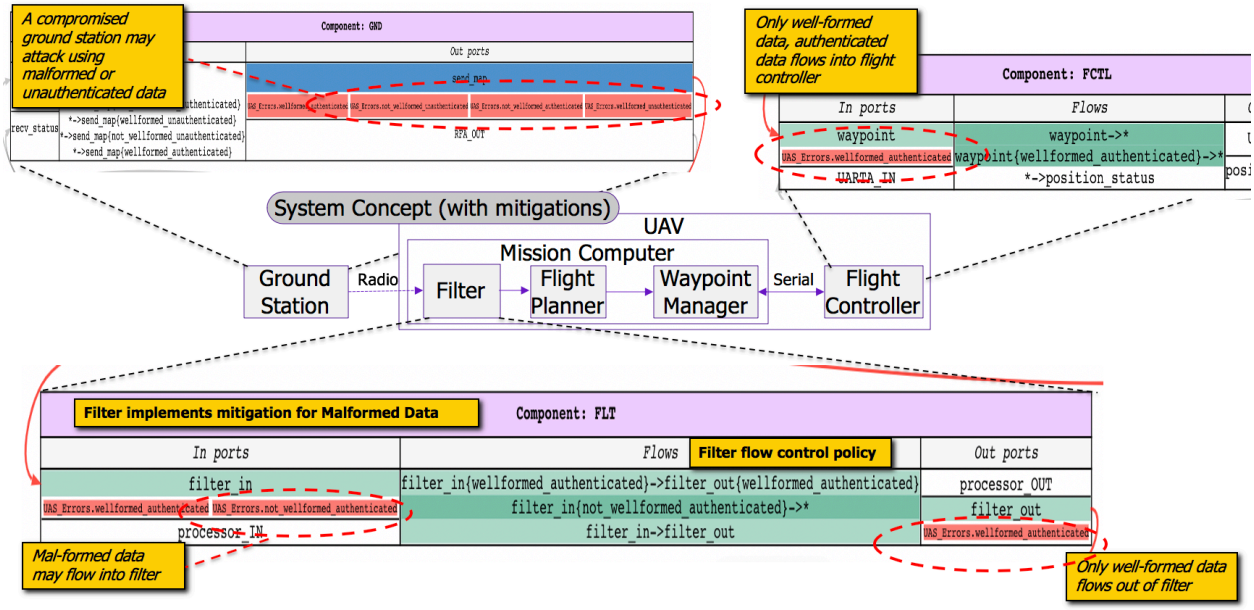


Figure 6.5: Awas Visualization of AADL EMv2-based Security Properties (Details)

thentication and message well-formedness properties related to the UAS example. Figure 6.4 shows an annotated screenshot of the Awas visualizer for the data security property analysis applied to this example. The figure illustrates the flow properties of the system after adding flow controls that authenticate commands and filter out malformed Ground Station data. The top-left quadrant shows the top-level system architecture. The colored markups highlight the *send_map* port of the Ground Station and information flow channels into the UAV.

Figure 6.5 shows a simplified version of the system architecture (for presentation sake), overlaid with portions of Figure 6.4 that capture key specifications and analysis results. The top left shows a magnification of the Ground Station visualization; the outgoing *send_map* port of Ground Station is annotated to indicate that a compromised Ground Station may send malformed map data or otherwise untrustworthy data or commands. Diving down into the UAS architecture in the top right of Figure 6.4, the visualization shows map information flowing through the UAV mission computer, with the bottom right showing the map information's path through mission computer components. The bottom left shows the details of the mission computer software architecture. To guard against the threats captured earlier in

annotations for the Ground Station, the radio driver component was modified to authenticate the map and commands, and a filter component was added that drops map malformed messages.

Figure 6.5 zooms in on a summary of the filter flow policy, indicating that while "not well-formed data" flows into the filter, such data does not flow out of the filter (indicated by the $*$). In Figure 6.4, the flow leaving the filter is visualized by highlighting the path through the remainder of the software (where the map data is converted into waypoints) and mission computer (bottom right) to the flight controls (top right). Figure 6.5 also shows a summary of the flows into the flight controller indicating that the desired properties are satisfied for the waypoint data; that is, only waypoint data derived from authenticated and well-formed map data flow into the flight controls.

These examples illustrate a broader capability enabled by the synergistic interaction between AADL, EMv2 and Awas. A system can be analyzed for different safety and data security concerns relevant to a particular application (*e.g.* authentication and message wellformed-ness). These analyses do not need to consider details from scratch, and instead, they "piggyback" on the base flow channels and reason about whether or not desired properties exist at different points along those channels.

6.3 Query Language

The previous sections focused on how users interacted with Awas by clicking and selecting various options within the visualization interface. This section gives a brief overview of the Awas query language, which can be used to codify commonly executed queries, architecture-oriented requirements, or audit objectives. Queries can be presented to Awas by loading a text file, entering text through the Awas visualizer REPL, or through the Awas APIs.

Given a graph, performing transitive closures can answer reachability questions. However, to compute the reachability of ports and errors, Awas refines the computed results in nodes-level to port-level further down to errors tokens. Additionally, we support forward reachability, backward reachability, source to target reachability, and source to target reach-

ability realized as paths. Finally, to access and compose the above-mentioned reachability analysis, we provide Awas Query, a simple named query language with set operators and filtering mechanisms. In this section, we demonstrate reachability analysis using the query language defined by the grammar in Appendix A by posing reachability queries on the UAV model described in Section 3.1.

6.3.1 Forward Reachability

Awas forward reachability analysis answers the general question of "what are all the components (ports, connections) that depend on a component (port) of interest?". In essence, if a component fails to produce valid output, how does the rest of the system behave? Forward reachability is similar to forward slicing as both expose where the information flows to. The Simple UAS query concept below is an example of forward reachability.

Query Concept 1

If the ground station sends the map, where does information regarding the map flow? Also, where is it getting consumed?

In architecture assessment activity, the query's name indicates a specific property in the architecture, and thus the failure of the query translates to the property failure in the architecture. All Awas queries consist of two parts. The part of the query before the equals sign is the query's name, such as *forward_GND_send_map* in figure 6.6. Subsequent queries can be composed using the query name. The part after the equal sign is the query expression which gets evaluated against the model. Query expressions that start with the keyword *reach* are reachability queries. The following keyword *forward* dictates the direction of the reachability analysis. Finally, the canonical name of the port *send_map* serves as the criteria for the query.

Figure 6.6 shows the result of the query concept 1. The graph contains highlighted ports representing that they belong to the reachable set of port *send_map*. The result also includes the port *send_map*, albeit in a different color to indicate that it is the provided criterion for the query. Finally, the result includes the flow *recv_map*$\rightarrow^*$ that consumes the information

```
forward_GND_send_map = reach forward
    UAS_Impl_Instance.GND.send_map
```

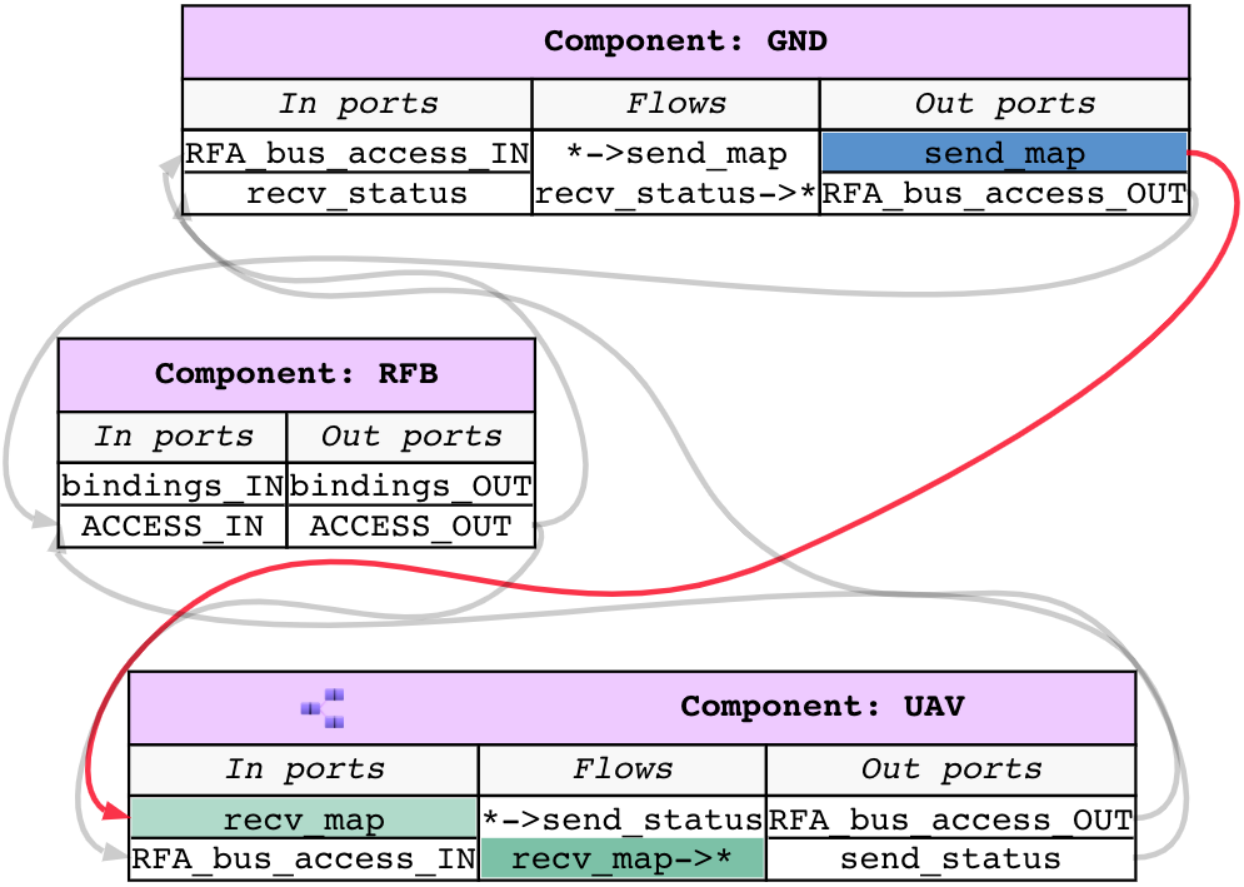


Figure 6.6: Forward reachability query and its result projected on the dependence graph

produced on the port *send_map*.

6.3.2 Backward Reachability

Similar to the forward reachability described above, the user can compute the reachability against the flow of information using the backward analysis. Backward analysis can answer the question of "where does the information needed by a component of interest flow from?". Additionally, in the case of safety analysis a backward analysis helps identify the root causes of a hazard or failure. Backward reachability analysis is analogous to a backward program slicing[71].

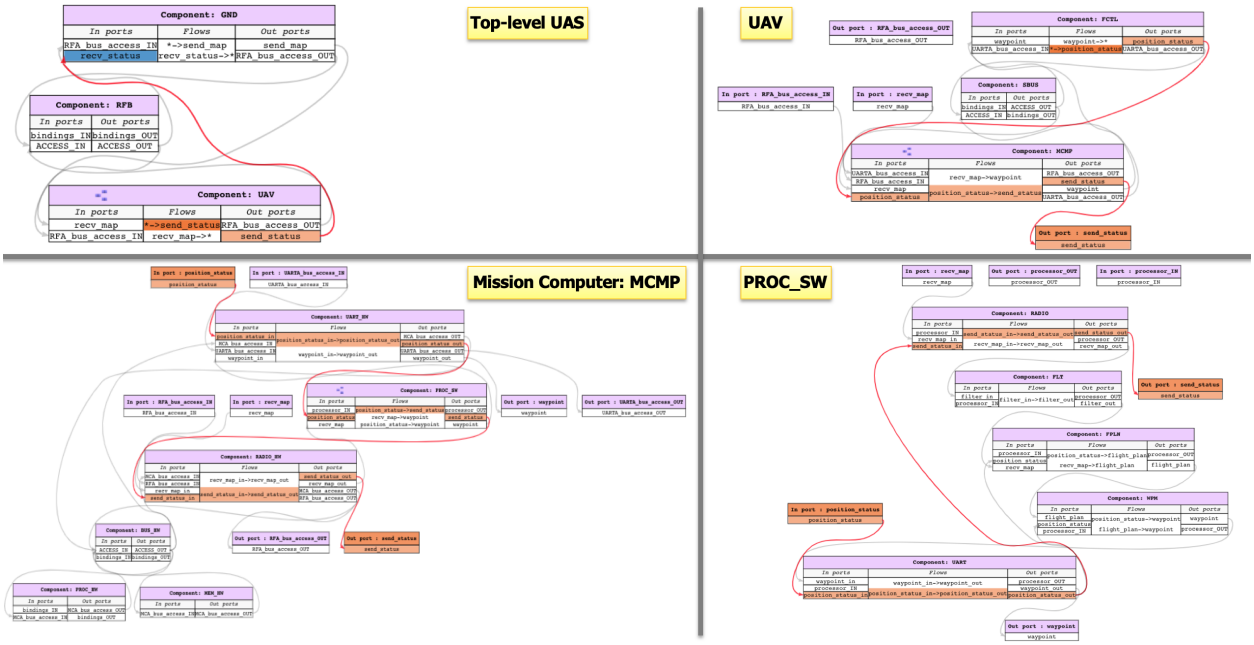


Figure 6.7: Backward reachability query and its result projected on the multiple graphs

Query Concept 2

From where does information needed to compute the recv_status flow from?

This query concept can be specified as follows:

```
backward_GND_status =

  reach backward UAS_Impl_Instance.GND.recv_status
```

The top of Figure 6.7 illustrates an Awas query describing Query 2. As Query 2 inquires about how the information reaches a port, the Awas query applies the *backward* analysis. The bottom of Figure 6.7 shows the result of Query 2. The result indicates that the information is flowing from the *UAV* component. Therefore, the reachability analysis uses the port *send_status* and descends into the *UAV* sub-system. In the *UAV* sub-system, on reaching the *MCMP* component, the reachability analysis has a couple of choices: *(i)* use the abstract flow defined in the *MCMP* component, or *(ii)* descent into the *MCMP* sub-system using the *send_status* port. Our reachability analysis takes both the paths and computes the reachability in the *MCMP* sub-system graph and its sub-system *PROC_SW*. Finally, our analysis finds the origin (flow source) of the information in the *FCTL* component.

6.3.3 Source and Target Reachability

Since forward and backward reachability analyses compute a transitive closure, a large system's forward or backward analysis results may overwhelm the user. If the user's concern is to check for the flow of information only up to a certain point in the system, then one can provide both the source and the target in the query to obtain a more focused set of results. The counterpart of this operation at the implementation level is program chopping[72;73].

Query Concept 3

When the ground station is sending the map, how does it get to the flight controller?

To formalize this query, the user intuitively asks if it is possible to reach the flight controller from the ground station and to show how if it is possible. In contrast, the previous queries only showed how the information flowed into or out of a specific port.

```
GS_flight_controller = reach from UAS_Impl_Instance.GND.send_map
                           to UAS_Impl_Instance.UAV.FCTL.waypoint
```

Figure 6.8: Query with both source and target

Figure 6.8 presents the reachability query for 'Query Concept 3'. This query illustrates that the user can specify both the source and sink of the information flow using the keywords *from* and *to*. The result includes all the connections, components, and ports responsible for propagating information between *send_map* and *waypoint*.

Figure 6.9 shows the result of executing the query in figure 6.8. The source and the target are highlighted in blue and the ports that are part of the information flow are highlighted in purple.

6.3.4 Path Reachability

In the chopping analysis above, the computed result includes all the nodes contributing to the reachability of a target node from a source node. However, the result does not distinguish each sequence of nodes that traces a path from source to target. In some instances, it is useful to realize the results as paths. We compute the path similar to the *meet-over-all-path*

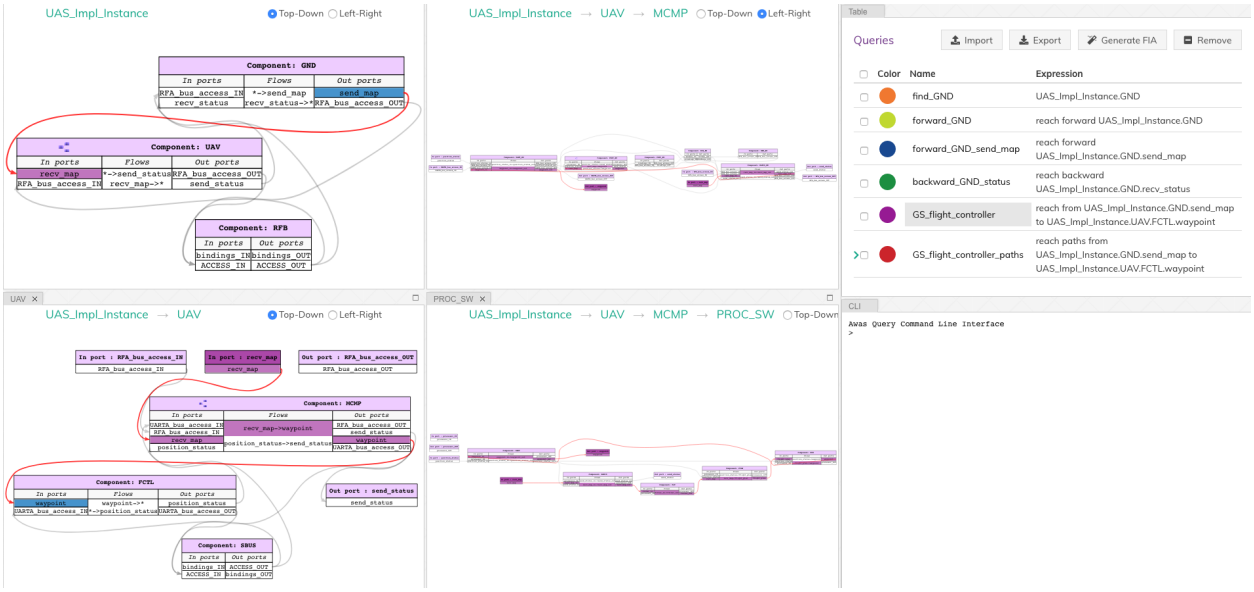


Figure 6.9: Result of query concept 3 6.8

solution in the program analysis by Kildall *et al.*[74]. We split a path whenever more than one intra-component flows are defined for a port and there are two or more edges leaving a port. In the absence of the intra-component flows, Awas conservatively assumes flows connect every pair of input and output ports. This causes an exponential number of paths and degrades the performance. In a typical workflow, this would motivate the developer to introduce flow specifications into the model (something that should be done in any case to support other assurance needs) to yield effective query times.

On the other hand, if the graph is strongly connected, there can be many paths. In that case, we provide a filtering mechanism based on the presence or absence of a node/port in the path using *all*, *some*, and *none* keywords.

Query Concept 4

When the ground station is sending the map, is it always flowing through the filter?

```
paths_FTL = reach refined paths from UAS_Impl_Instance.GND.send_map
                                 to UAS_Impl_Instance.UAV.FCTL.waypoint
                      with none(UAS_Impl_Instance.UAV.MCMP.PROC_SW.FTL:port)
```

Figure 6.10: Query with both source and target and path filters

This query concept is formalized in Figure 6.10. This query checks for the first require-

```
valid = reach paths from UAS_Impl_Instance.GND.send_map:port-error
                      to UAS_Impl_Instance.UAV.FCTL.waypoint:port-error
```

Figure 6.11: Reachability query with EMV2 errors

ment defined in Section 3.1.2. It identifies individual paths that can reach the port *waypoint* from port *send_map*, and along all paths, it checks for the existence of a path without any of the ports from the filter (*FTL*) component. The keyword *refined* informs the query evaluator to ignore the paths using summary flows in the parent components. An empty result for the above query indicates that all the paths from *send_map* to *waypoint* pass through the *FTL* component.

6.3.5 Error Reachability

Although OSATE provides several forms of hazard analysis to calculate the error propagation in an AADL model[36;75], it does not *explain* and *visualize* the propagation of errors in the system. In our approach, we overlay the error propagation information on the Awas graph to provide evidence of an error affecting other components. The Awas query language includes the ability to use EMv2 error tokens to issue queries with error tokens.

Awas computes the error propagation and transformation using a simplified version of Fault Propagation and Transformation Calculus (FPTC)[69]. Awas first computes reachability without the error information, and then refines the result with error propagation.

Query Concept 5
Do only authenticated and well-formed maps reach the flight controller?

This query concept poses the question of whether issues such as *not_wellformed* or *unauthenticated* information originating from the ground station reaches the flight controller. In essence, we are checking for the possibility of the situation where an adversary can take control over the UAV or the possibility of crashing the UAV due to corrupted data.

The query specification captures that even though the port *send_map* may propagate *wellformed_authenticated, wellformed_unauthenticated, not_wellformed_authenticated,* and *not_wellformed_unauthenticated* only *wellformed_authenticated* reaches the port *waypoint*

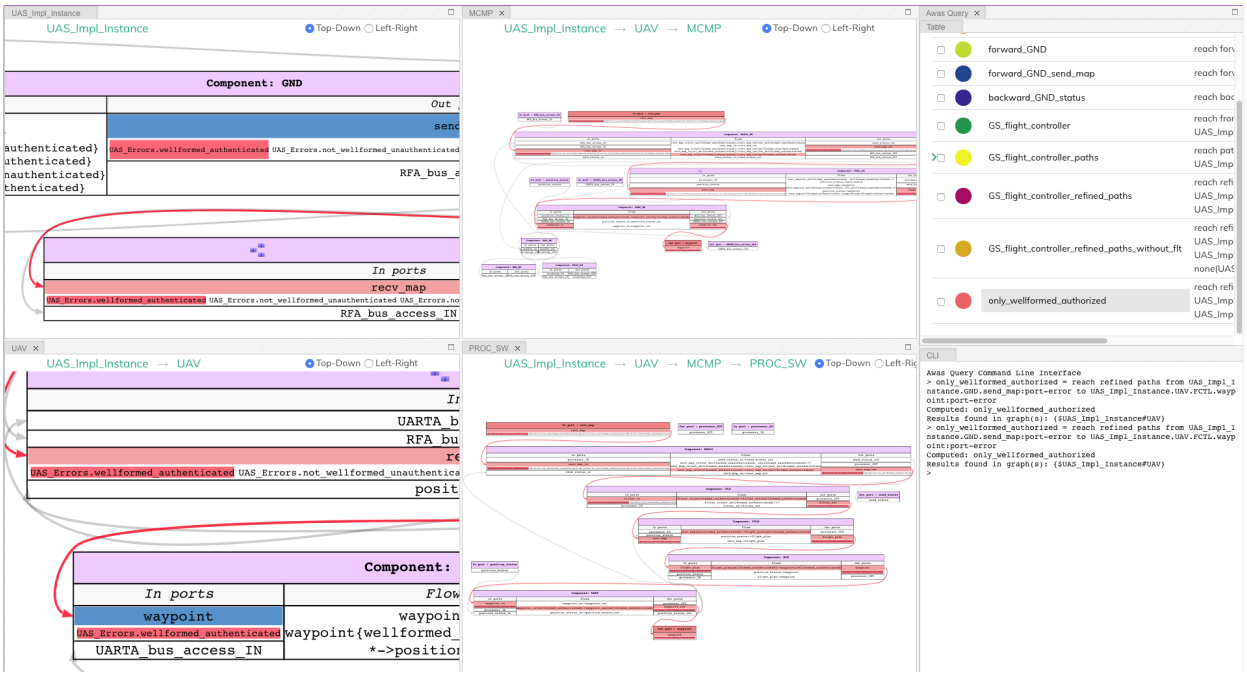


Figure 6.12: Result of query 6.11

as shown in figure 6.12.

The effect of mitigation mechanisms can be checked by selecting on error tokens *well-formed_unauthenticated*, *not_wellformed_authenticated*, or *not_wellformed_unauthenticated* and clicking the forward button. The result will show the propagation of the error and where it is mitigated.

Chapter 7

Application of Awas

> *"You cannot answer a question you cannot ask, and you cannot ask a question that you have no words for."*
>
> – *Judea Pearl, The Book of Why: The New Science of Cause and Effect*

This chapter presents two applications of the Awas dependence analysis. Section 7.1 describes the computation of the causal chain from an initiating cause to the hazardous situation using Awas. In section 7.4 we provide an annex for AADL to model secure information flow policies using security classification types. We use Awas to infer the security types and check the system against the policy.

7.1 Automating risk analysis of ISO 14971

ISO 14971 defines a number of concepts that need to be reflected in medical device MBSA. *Harm* is defined as "injury or damage to the health of people, or damage to property or the environment". *Hazard* is a "potential source of harm", and a *hazardous situation* is a "circumstance in which people, property or the environment is/are exposed to one or more hazards." *Initiating cause* is not a ISO 14971 defined term, but it is used in the standard to refer to faults or other issues that lead to a hazard.

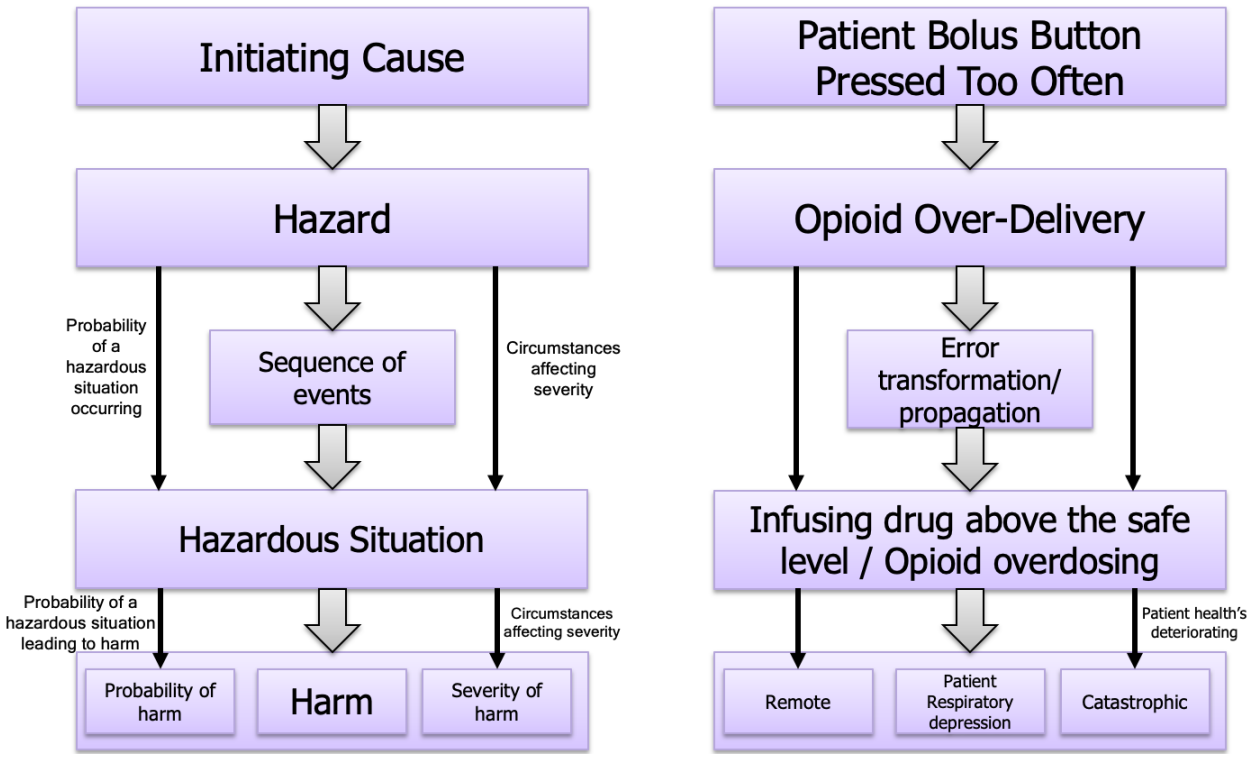


Figure 7.1: ISO 14971 Key Risk Analysis Terms and Relationships

The left side of Figure 7.1 (slightly adapted from ISO 14971 Annex C) illustrates the relationships between these terms. The scope of our work is *functional safety* – potential harms associated with incorrect function of software and hardware elements of the device, rather than physical, chemical, mechanical, electrical, and biological safety discussed in ISO 14971. Table 7.1 provides instances of the terms of Figure 7.1 related to the functional safety of the PCA Pump. The two primary hazards are opioid over-infusion and infusion of air bubbles into the patient blood stream – in both situations, severe consequences including death can be caused to the patient. Both of these hazards can have multiple initiating causes (excerpts are shown in Table 7.1), and our full models capture these along with associated mitigations. Due to space constraints, we limit detailed discussion to selected cases.

Harm	Hazardous Situation	Hazard	Initiating Cause
Cardiac arrest	Infusing Opioid when patient's respiration is deteriorating	Opioid overinfusion	a. Bolus button pressed too frequent b. Incorrect pump calibration
Tissue or organ damage	Infusing air bubbles into the patient's blood stream	Air embolism	Continue infusion i) after inappropriate priming, ii) with empty reservoir, or iii) under tubing leakage

Table 7.1: *ISO 14971 Risk Analysis Concepts Applied to the PCA Pump (excerpts)*

7.2 AADL Error Modeling for the OpenPCA System

In the discussions that follow, we use the following role names to distinguish the activities of different stakeholders of the framework: *tool designer* refers to authors of this paper and associated steps necessary to configure and extend AADL for our described ISO 14971 framework; medical device *manufacturer* refers to associated steps taken to configure the framework for their organization, which may include multiple medical devices; *analyst* refers to activities associated with using the configured framework to support risk analysis for a particular medical device.

Section 7.3 describes new analysis automation and reporting mechanisms that we have developed that aggregate all of these annotations into structure information that can be actively browsed, queried, and used to generate ISO 14971 aligned risk analysis reports with active elements that link directly to models and causality visualizations.

Preparing the ISO 14971 Framework – Tool Designer: Part of our effort to configure AADL EM for ISO 14971 involved defining schemas for related properties. The listing below shows the schema for the notion of harm. Schemas for Hazard, Hazardous Situation, and Cause are similar.

```
Harm: type record ( ID: aadlstring; -- unique ID used as reference to the
    harm
  Description: aadlstring; -- description used in report generation
  Severity: ISO14971_80001::SeverityScales;); -- associate severity
```

AADL EM includes a standard error type library that captures many of the notions

in the fault taxonomy of[35]. For this paper, we configured a simplified type library that is sufficiently general for supporting medical device risk analysis. Device manufacturers can further specialize the library to introduce notions of fault and error specific to their products. As an example of such customization, previous work from our research group created specializations for supporting risk analysis for interoperability and security related issues[50].

Instantiating the ISO 14971 Framework – Device Manufacturer: The device manufacturer may configure the framework for their general risk management process. This includes defining qualitative categories for severity and frequency (e.g., choosing among options listed in ISO 14971 2009 annexes and supporting technical reports, extending the AADL error library to reflect taxonomies of faults and hazards used within the manufacturer's risk management process.

Identifying Harms, Hazards, and Hazardous Situations for a Device – Analyst: Drawing on information gathered from the ISO 14971 process steps of "gathering characteristic related to safety" and "identifying hazards" (clauses 5.3 and 5.4) the analyst introduces model annotations to capture harms and hazards. The discovery of harms/hazards cannot be supported to any significant degree by automated tooling, but instead relies on domain knowledge, experience, clinical trials, and medical domain literature to identify relevant harms and hazards.

US FDA Guidance documents are often a good source for this type of information. [1] The listing below illustrates the definition of an over-infusion hazard and an associated harm of respiratory depression. In addition to identifiers used in reporting, the harm specification classifies severity of this harm as catastrophic (essentially uses an enumerated type value from a set of qualitative severity values configured by the manufacturer). Other harms/hazards (omitted here) that we captured for the PCA Pump related to functional safety include the

[1]The FDA Guidance on Infusion Pumps includes a number of example hazards including *Infusion Delivery Error* which is described as "Intended medication selected and delivery attempted, but failure to deliver within the right time, dose, volume, patient, or anatomical or physiologic site specifications. This can include over-delivery, under-delivery or delay in delivery situations." For PCA Pumps, over-delivery of opioids is the primary concern, so we create a specialization of infusion delivery error to "Over-infusion" as captured in the AADL property above.

air embolism and under-infusion as presented in Section 7.1.

```
--Harm
H1: constant ISO14971_80001::Harm => [
  ID => "H1";
  Description => "Respiratory Depression";
  Severity => Catastrophic; ];
--Hazards
Haz1: constant ISO14971_80001::Hazard => [
  ID => "Haz1";
  Description => "Drug over-infusion"; ];
```

a. Harm and Hazard instance

```
FrequentButtonPress : constant
  ISO14971_80001::Cause => [
   ID => "FrequentButtonPress";
   Description => "";
   Probability =>  Frequent; ];
```

b. Cause Instance

Similarly, the analyst introduces property that will label events/state in the device or environment that represent an initial step in a causality chain. The example below illustrates the introduction of a reporting label for a frequent bolus button press (one of the root causes of a potential over-infusion hazard). Additional causes may also be discovered and added in the process of the analysis (e.g., applying the tools of Section 7.3).

To configure the error propagation layer, based on their domain knowledge, analysts introduce EM error types representing different types of root causes and observable problematic device behaviors that may contribute to harms. The listing below shows excerpts

that define error types for problematic environment actions that (without mitigation) may cause harms as well as observable device behaviors that may lead to harms.

```
annex EMV2 {**
  error types --errors caused in the environment
  DrugKindError : type;  --wrong drug is loaded into reservoir
  TooSoonPress : type;   -- patient button pressed too soon/often
  ThirdPartyPress : type;  --someone other than the patient presses the
     button
     ...
   -- errors indicating patient harm
  AirEmbolism : type;  -- air bubble in fluid emitted from device
  DrugOverInfusion : type; -- too much drug, possibly harmful
  DrugUnderInfusion : type; -- too little drug, insufficient to reduce pain
**}
```

Now the analyst uses AADL EM annotations to connect the layers in the framework – linking the elements above to the architecture description. Such annotations are added throughout the architecture, but an especially important step is the treatment of the system boundary to reflect both environmental causes of hazards (typically associated with device inputs) and observable device behaviors that may lead to harm (typically associated with device outputs). The listing below shows excerpts of the system boundary model – focusing on annotations that address frequent bolus request / over-infusion.

```
system PCA_Pump_System extends Platform::Generic_System
 features
  sense: refined to feature group iPCA_Feature_Groups::Sensing_iPCA;
  act: refined to feature group iPCA_Feature_Groups::Actuation_iPCA;
  fill_drug: in data port Physical_Types::Fluid_Volume;
```

```
properties
 ISO14971_80001::SystemInfo => [
  Name => "Open PCA Pump";
  Description => "Patient-Controlled Analgesic infusion pump";
  IntendedUse => "patient for pain management"; ];
 ISO14971_80001::Hazardous_Situations =>
     (HazardousSituations::OverInfusion,
   HazardousSituations::UnderInfusion,
       HazardousSituations::IncorrectDrug);
 annex EMV2 {**
  use types iPCA_Error_Model, ErrorLibrary;  -- import error types
  error propagations
   -- drug output may be wrong flow rate, kind of drug, or air bubble
   act.drug_outlet: out propagation {DrugStopped, DrugOverInfusion,
      DrugUnderInfusion, DrugKindError, AirEmbolism};
   fill_drug: in propagation {DrugKindError};  -- wrong drug filled
   -- button pressed before next bolus permitted
   sense.patient_button_press: in propagation {TooSoonPress,
      ThirdPartyPress};
   sense.barcode_signal: in propagation {ValueError}; --barcode corruption
   sense.ui_touch: in propagation {OperatorError};  -- clinician error
  end propagations;
  properties  --  AADL properties specify error sources and resulting harms
   ISO14971_80001::causes => (Causes::FrequentButtonPress)
         applies to sense.patient_button_press.TooSoonPress;
   ISO14971_80001::causes => (Causes::IncorrectDrug)
         applies to fill_drug.DrugKindError;
```

```
    ISO14971_80001::Hazards => (Hazards::Haz1)
          applies to act.drug_outlet.DrugOverInfusion;
    ISO14971_80001::Hazards => (Hazards::Haz2)
          applies to act.drug_outlet.DrugUnderInfusion;
    ISO14971_80001::Hazards => (Hazards::Haz3)
          applies to act.drug_outlet.DrugKindError;
  **};
end PCA_Pump_System;
```

In particular, on the `patient_button_press` sensor input, an EM flow annotation of `ButtonError` models button presses that occur too often. The AADL EM `applies` construct associates the `Cause::FrequentButtonPress` cause with the `ButtonError` flow token, which has the effect of linking the error token (and flows proceeding) from the token to the reporting framework as a possible cause of (and causality chain leading to) a hazard. Similarly, the `DrugOverInfusionToken` is associated with `drug_outlet` output, and then associates flow leading into that token as well as the token itself with the `Haz1` annotation which is understood by the reporting framework.

Using the analysis framework to identify Sequences of Events – Analyst: ISO 14971 Clause 5.4 states that "For each identified hazard, the manufacturer shall consider the reasonably foreseeable sequences or combinations of events that can result in a hazardous situation, and shall identify and document the resulting hazardous situation(s)." To support this requirement, the analyst adds flow annotations to components The analyst adds flow annotations to components throughout the architecture to model causality paths and then uses the analysis capabilities in Section 7.3 to compute various forms of reachability and report generation.

The fragments in the listing below illustrates how flow annotations are added to capture error propagations indicating that a component (a) may be a *source* of an error, (b) may propagate errors (and possibly transform the type of error), and (c) may *sink* an error (i.e., serve as a mitigation for an error).

```
calibration_over : error source drug_outlet{DrugOverInfusion};
mp_err: error path drug_intake{DrugKindError} -> drug_outlet{DrugKindError};
over: error path bindings {HighValue} -> drug_outlet{DrugOverInfusion};
pbc: error sink patient_button_request {TooSoonPress, ThirdPartyPress};
```

In the component for the mechanical pump which takes actuation commands from the control logic (including setting the flow rate), the first line in the listing models the fact that a lack of calibration of the pump itself could cause fluid to be moved out of the `drug_outlet` port at a rate that exceeds the pump's specification, resulting in an drug over-infusion error (a corresponding under-infusion error is omitted). The second line models a situation where the wrong drug enters the `drug_intake` port (intuitively, because the nurse has entered a vial in the drug reservoir with the wrong drug) – in this case, the error propagates from the input to the output (i.e., the wrong drug flows through mechanical pump). The third line models a situation where the control logic has commanded a flow rate that is too high: the `HighValue` error is transformed to a `DrugOverInfusion` error indicating that the bad command causes a problematic high flow out of the mechanical pump. The final line models the patient bolus checker component that (partially) mitigates errors related to the bolus button being pushed too soon or by a third party by limiting the number of active button pushes over a time period. In this case, the component acts as a sink for the errors.

As flows are explored using the tools in Section 7.3, the analyst is interested in understanding the relationships between causes, hazards, and harms. A hazardous situation describes relationships between a hazard and a harm. The analyst records a hazardous situation by introducing a model property such as in the listing below. The hazardous situation instance below describes a scenario in which the Haz1 leads to the harm H1. During the analysis, the causality relationship between hazards and initiating causes are computed. Hence, providing a complete scenario of error flow from initiating cause to hazards to hazardous situation and finally leading to harms (this is reflected in the 14971 reports described in the following section).

```
OverInfusion : constant ISO14971_80001::Hazardous_Situation => [
```

Component: pump		
In ports	*Flows*	*Out ports*
gpio_IN	drug_intake->drug_outlet	drug_outlet
bindings_IN	bindings_IN{LowValue}->drug_outlet{DrugUnderInfusion}	rugKindError iPCA_Error_Model.I
r_Model.HighValue iPCA_Error_Model.LowValue	bindings_IN{HighValue}->drug_outlet{DrugOverInfusion}	bindings_OUT
drug_intake	*->drug_outlet{DrugOverInfusion}	power_OUT
iPCA_Error_Model.DrugKindError	*->drug_outlet{DrugUnderInfusion}	
power_IN	drug_intake{DrugKindError}->drug_outlet{DrugKindError}	gpio_OUT

Figure 7.2: Awas AADL Intra-component Error Flows Visualization

```
ID => "OverInfusion";
Description => "Infusing drug when the patient's health is deteriorating";
Hazard => Hazards::Haz1;
Paths_to_Harm => ([
 Harm => Harms::H1;
 Contributing_Factors => (ContributingFactors::HealthDeteriorating);
 Probability_of_Transition => Remote;]);
Risk => High;
Probability => Remote; ];
```

7.3 AADL Error Modeling Analysis Support

A variety of analyses can leverage the error flow and ISO 14971 property annotations in the previous section. The OSATE AADL EM plug-in provides several different forms of safety analysis including fault tree analysis and a simple functional hazard analysis. In this section, we illustrate Awas[76] which complements these existing analysis with a scalable interactive visualizations and queries of error flows. In the ISO 14971 context, these capabilities are applied to automated discover and visualize potential "sequence of events" leading from causes to hazards, to hazardous situations, to harms.

Awas builds *component* and *system* visualizations that are tailored to illustrating *flow-related* aspects. Figure 7.2 illustrates how Awas builds a component-level summary of flow properties that show component inputs (left side), outputs (right side), and the error flow

rules (middle) that the analyst has specified to capture how error tokens propagate from inputs to outputs.

Awas builds a dependence graph composed from intra-component flows (as in Figure 7.2) together with several forms of inter-component dependences including port connections, component bindings, etc. The flow graph representation and analysis algorithms are written in Scala and compiled to Javascript using the Scala.js framework[2]. This generates a highly navigable, dynamic visualization of flows integrated across all levels of the system hierarchy. The most basic capability is forward/backward reachability analysis. Analysts simply click on a component or port and press a button to carry out basic queries such as "where in the system do the modeled errors (and their subsequent impacts) from this port/component flow?" or "what system elements are contributing errors that flow into this port/component?".

In the example of Figure 7.3 [3], the analyst clicks on the system boundary `sense.patient_button_press` input port with an error token indicating a possible "too frequent" bolus button push and presses the Forward analysis button to have the tool discover and mark up to the where in the architecture the effects of this error may propagate (paths are shown in red, and components and ports along the path are shown in green and red). The Open PCA architecture includes approximately 19 sub-systems/component levels of hierarchy. Using the window-tiling capability of Awas, Figure 7.3 shows three such subsystems opened (system top-level, a portion of the functional architecture, and lower-level hardware resources). Behind the scenes, the reachability information is computed almost instantaneously across the entire system, A simple scroll of a mouse wheel zooms into a particular system section or component of interest. Double-clicking on components drills down to their subcomponent models. Projections of the system can be performed on components/flows of user-specified categories, or components along user-specified paths.

This supports expected ISO 14971 workflows as follows. Working in either a bottom up manner (from causes to hazardous situations) or top-down manner (from hazardous

[2] `www.scalajs.org`

[3] Note that the purpose of these screenshots is to illustrate application of the Awas tools at scale (capturing system-wide browsing across a large system with many complex components). The screen captures of the tool cannot capture both the scalability aspect while preserving the readability of the component/port/details, etc. In the Awas tool, mouse scrolling easily zooms in and out to reveal details.

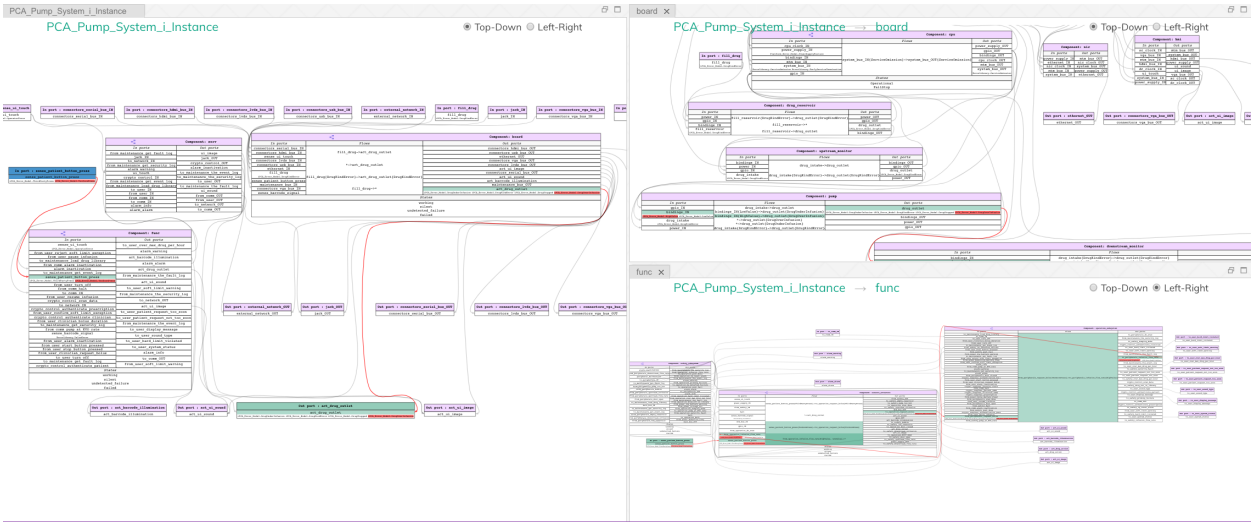
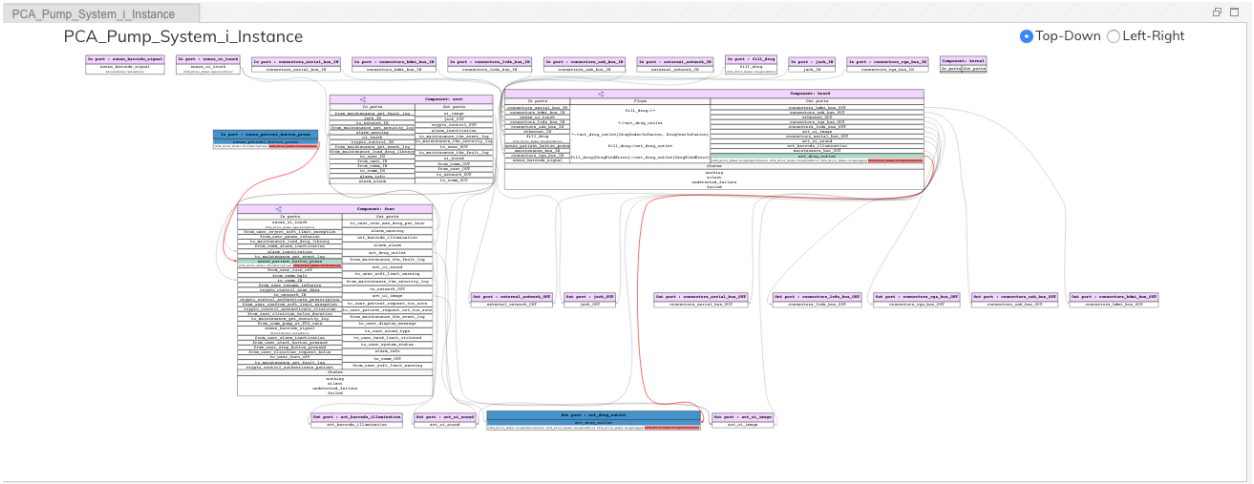


Figure 7.3: Awas AADL System-wide Error Flow Visualization (selected sub-systems)

situations to causes), the analyst uses both forward and backward Awas reachability to discover causality chains in the error-flow annotated architecture. Annotations marking causes and hazardous situations are incrementally added to the model as important aspects of error propagations are revealed in Awas. The web-site supporting this submission illustrates further capabilities in the browser-based deployment of Awas (no tool installation needed) including the ability to define and save more sophisticated queries written in a form of path logic. As the analyst discovers error propagations and begins to annotate the architecture for mitigation strategies, this enables common queries corresponding to hazardous situations to be replayed as mitigations are added to confirm that impacts of causes are eliminated or reduced.

On top of the general Awas capabilities, we have developed a reporting tool that produces information in the formats suggested by ISO 14971 and associated medical domain risk management guidance. Figure 7.4 illustrates an excerpt of this report that captures the association between hazardous situations and related concepts. This information is automatically extracted from the model based on the model annotations of Section 7.2 and the Awas reachability analysis. Both PDF and HTML versions of the report are produced. The HTML report (Figure 7.4) is "animated" in the sense that one can highlight a certain



Top Down Analysis

Hazards	Contributing Factors	Initiating Causes	Hazardous Situation	Harm
Haz1	PatientHealthDeteriorating	PumpIncorrectCalibration FrequentButtonPress	OverInfusion	H1
Haz2	PatientInGoodHealth	PumpIncorrectCalibration	UnderInfusion	H2
Haz3	ClinicianFailedToCheck	IncorrectDrug	IncorrectDrug	H3

Figure 7.4: Awas ISO 14971 Report (excerpts) illustrating Sequence of Events Leading to Hazardous Situation

hazardous situation (the selection is shown in blue), the Awas visualization for the causality chain from cause through hazardous situation to harm is automatically computed and displayed in the report, corresponding to the ISO 14971 requirement that the analyst uncover "series of events" (see Figure 7.1) along the causality pathway). Figure 7.4 shows excerpts capturing only a portion of information related to the over-infusion hazard. The website artifacts show a much expanded report capturing a number of other hazardous situations.

7.4 Security Modeling Framework

In critical systems, protecting confidential information is a long-standing problem. The popular technique to ensure confidentiality is through access control. Components or users require certain privileges to access confidential data. However, access control policies do not enforce how the acquired confidential data is manipulated or propagated. In large systems, manually tracking every sensitive data through the system is not feasible. Always keeping

the confidential data encrypted and decrypted only at the point of use puts strain on the system with a lot of encryption and decryption. Also, it is not safe to assume that the encryption and decryption components do not leak sensitive information. Formally verifying every context of crypto functions is costly.

It is important to analyze how information flows to check if the system satisfies its security policies. The checker transforms the security policies into information flow policies and checks them against the system[77]. In the last couple of decades, considerable efforts have been made in the use of type systems for information flow[78–81]. This work presents a security modeling annex to capture the security types in the AADL model.

The Security Modeling Framework (SMF) is an annex developed to support information flow types system in AADL. Information flow type system is used to specify parts of the system in different security classification and analyze the system to ensure it satisfies the security requirements. There are two parts to the SMF annex language.

- SMF Library, and
- SMF Model

```
1 package SecurityLevels
2 public
3   annex smf {**
4     domain types
5       trusted: type;
6       untrusted: type extends trusted;
7     end types;
8   **};
9 end SecurityLevels;
```

Figure 7.5: SMF Library example

The SMF library provides the adequate language constructs to define Denning-style security lattices as discussed in section 4.1.2. The figure 7.5 demonstrate the definition of security types *trusted* and *untrusted* on line 5 and 6. The "extend" keyword on line 6 enforces a partial ordering of *trusted* $\leqslant$ *untrusted* between the types. The policy checker is

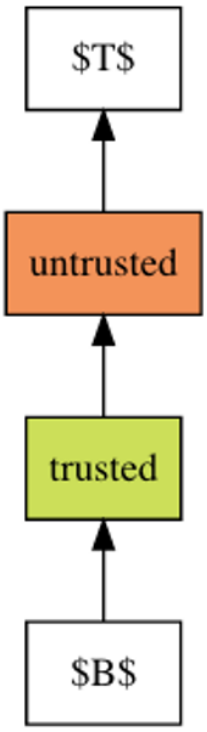


Figure 7.6: Generated Hass diagram of security type lattice

designed in a way to operate only on a complete lattice therefore, we introduces a unique greatest (T) and least (B) elements to the lattice. Figure 7.6 shows auto-generated hass diagram for the security types defined in figure 7.5

The partial ordering follows the Bell-LaPadula model of mandatory access control where the information from *trusted* level is allowed to flow into *untrusted* but not vice versa. In the absence of the extends keyword, the *trusted* and *untrusted* are considered to be disjoint security domains, and the information flow between them is not permitted. Figure 7.7 shows the generated lattice of disjoint security domains.

In the UAV system described in the section 3.1, there is a possibility that the Ground Station is sabotaged, and using the commands from the Ground Station, an adversary can capture or crash the UAV. All commands from the Ground Station go through a filter component before reaching the Flight Controller to avoid such situations. The Ground Station is considered an untrusted source, and the Flight Controller should operate only on the trusted commands. The two security domains defined using the SMF library are

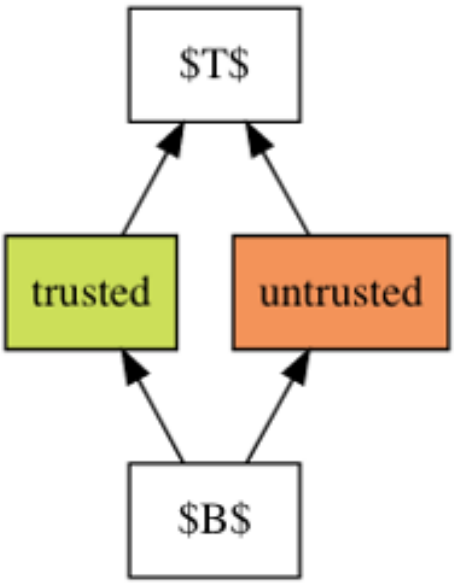


Figure 7.7: Security lattice with disjoint domains

associated with the 'send_map' port of the Ground Station and 'waypoint' port of the Flight Controller as shown in figure 7.8. The complete grammar for the SMF annex is provided in appendix C.

```
-- in ground station
annex smf{**
  classification
    send_map: untrusted;
**};

-- in flight controller
annex smf {**
  classification
    waypoint: trusted;
**};
```

Figure 7.8: Association of security types

The analysis happens in the Awas visualizer and projects the inferred type on the model. Figure 7.9 shows the result of the analysis. The 'send_map' port on the ground station component is marked with the *untrusted* type in solid highlight, indicating that the user

provides the type. Where else on the UAV's 'recv_map' port, the *untrusted* type is highlighted in stripes to differentiate that the type is an approximate inference of *geq untrusted*. Similarly, the 'waypoint' port on the flight control component is marked as *trusted* with a solid highlight. However, the red edge on the highlight indicates that there is a violation of the policy where *untrusted* data is leaking into *trusted* domain.

The table at the bottom list the violations of the security policy, on clicking each row, the violating path is highlighted on the model.

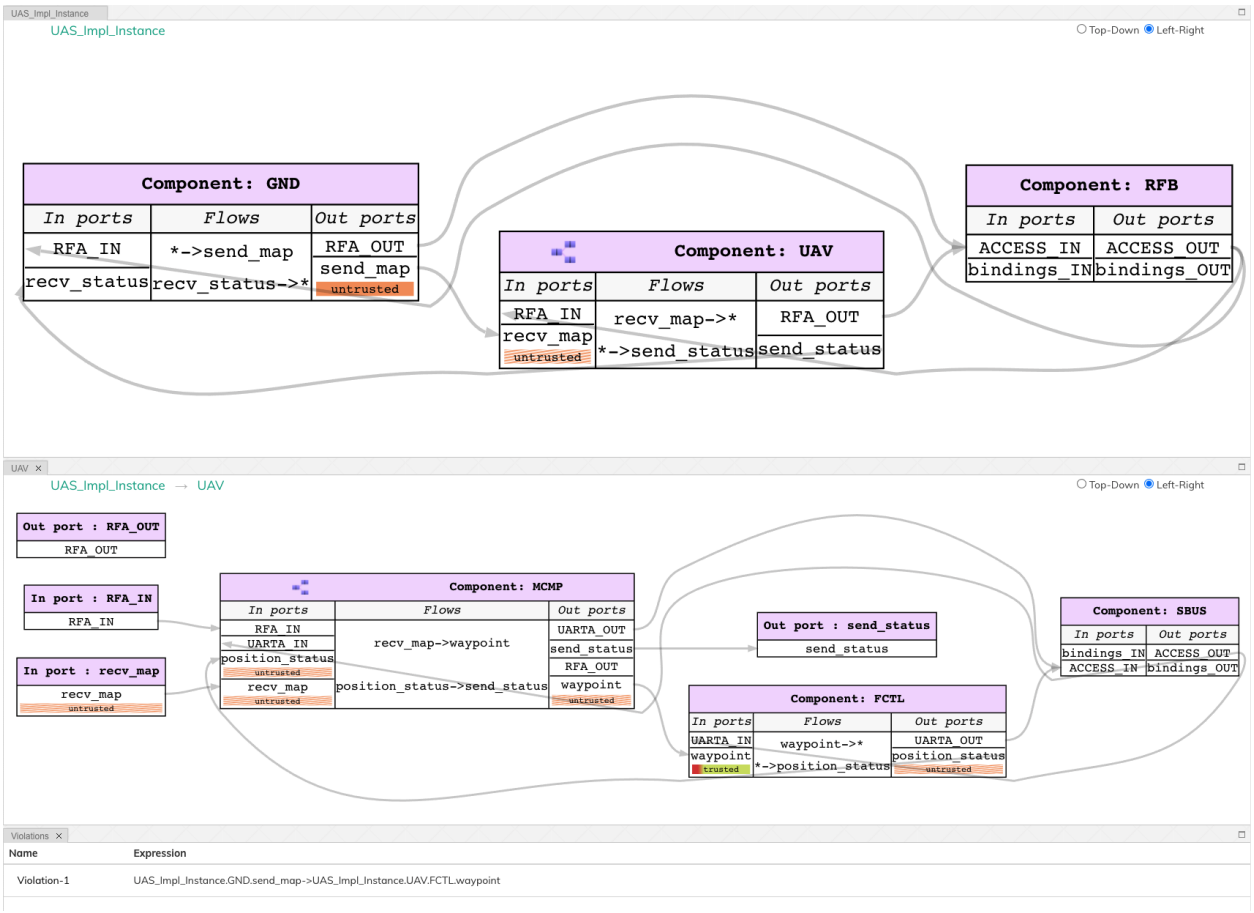


Figure 7.9: Result of SMF analysis on UAV system

To satisfy the security policy, the filter component in the software subsystem is equipped to de-classify the *untrusted* security type to *trusted* type. This is an abstraction of the behavior of the filter component with respect to the security policy.

The declassification is applied to the flow in the filter component that propagates the filtered commands from the ground station to the flight controller. After the filtering process

```
-- in filter component
annex smf {**
  de-classification
    filtered_cmd: untrusted -> trusted;
**};
```

Figure 7.10: De-classification of security types

the *untrusted* data is classified as *trusted*.

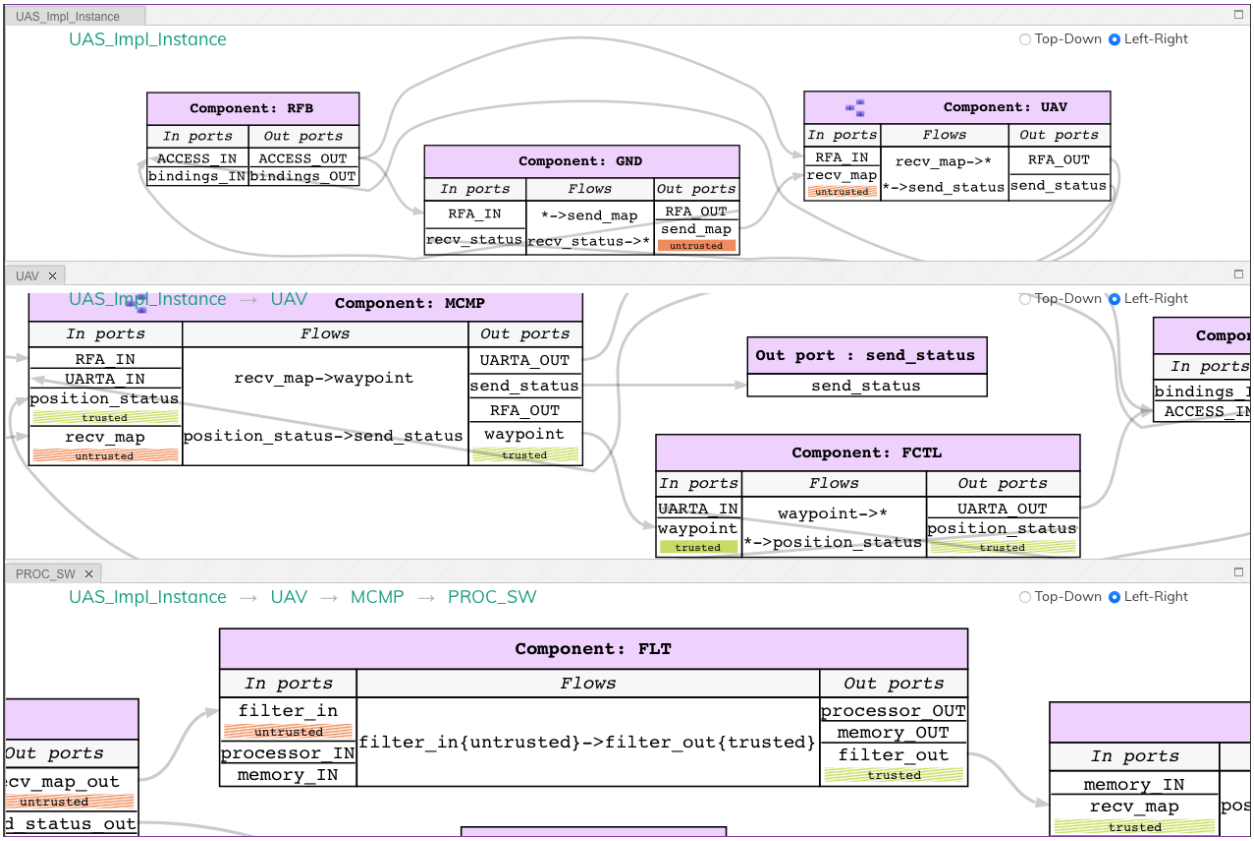


Figure 7.11: Filter component with declassification policy

Figure 7.11 illustrates the application of declassification on the filter component with the data from ground station 'send_map' to filter as *untrusted* and filter to flight controller as *trusted* flow of information.

7.4.1 Analysis

The analysis is an extension of my previous work on checking and inferring security policies on Spark Ada programs[82] which in itself is based on Hammer *et al.*[83] work on program analysis based secure information flow analysis. This technique is differs from the semantics-based security and security-type system by computing reachability on a graph.

The analysis operate on the Awas graph defined in section 6.2.1. Additionally, following are defined for the analysis

- S: partial ordered set of security types
- P: is a set of provided types of the form $\sum_{\mathcal{P}} \times S$, where a port is associated with a security type
- I: $(\sum_{\mathcal{P}}$ - $domain(P)) \times \bot$, initializing all other ports with security lattice's bottom
- D: is a declassification relation (f, s1, s2), where f $\in \rightarrow_{\mathcal{F}}$, and s1, s2 $\in S$

There are two parts in the analysis:

1. Based on the provided set, infer the types for all other ports in the graph
2. Search for policy violation and if exists calculate the violating path

The type inference algorithm works in two steps:

1. The security type is forward propagated from the provided set until either all reachable ports are covered, or another provided port is encountered. When a port receives a security type, the existing type associated with the port is replaced with the LUB of incoming and existing types.
2. The ports with | types are selected and replaced with $\top$ and from their successors the types are backward propagated by computing GLB until all the ports are explored at least once.

Algorithm 10: infer_types

input : ports : $\sum_{\mathcal{P}}$, types : S, provided : P, declass: D
Result: result : $\sum_{\mathcal{P}} \times S$
result ← (ports → |)
result ← result ∪ provided
worklist ← provided
while *exists* *current* ∈ *worklist* **do**
  worklist ← worklist − current
  (s, f, e) ← successor_port(*domain*(*current*))
  if *declass(f)* ≠ ∅ ***and*** *range(current)* = *domain*(*declass*(*f*)) **then**
    result(s) ← LUB(result(s), *range*(*declass*(*f*)))
  else
    result(s) ← LUB(result(s), *range*(*current*))
  end
  if *result modified* **then**
    worklist ← worklist ∪ modified elements of result
  end
end
I ← ∀ r ∈ result | *range*(*r*) = ⊥
gold ← result − I
worklist ← ∀ i ∈ I | successor_port(i) ∈ gold
while *exists* *current* ∈ *worklist* **do**
  worklist ← worklist − current
  ∀ p ∈ predecessor_port(*domain*(*current*)) | **if** *p* ∉ *gold* **then**
    result(p) ← GLB(result(p), *range*(*current*))
  end
end

The algorithm 10 computes a security type for all the ports in the graph. It does that efficiently by visiting each port and security type combination only once. The least upper bound and greatest lower bounds are computed on the security type lattice, which is essentially small compared to the size of the graph. The time complexity of this algorithm is bounded by $O(\sum_{\mathcal{P}} \times S)$.

The policy violations are identified by comparing every predecessor port type of a provided with its type. If they disagree with the policy, the violating path is computed by performing backward reachability from the provided until other provided are reached.

A downside of this technique is *label creep*. It is the effect of monotonically increasing the types. In the course of analysis, the type assigned to a port keeps increasing until a

fixed point is reached. Making too restrictive and forces the user to provide more types and negates the usefulness of inferring types.

Chapter 8

Integration and Evaluation

This chapter discusses the integration of Awas with the rest of the AADL tools. Our industrial partners (Adventium Labs, Software Engineering Institute, and Collins Aerospace) feedback guided the integration efforts. Finally, we present the practical performance evaluation for evaluating the queries against open-source AADL models.

8.1 Integration

Section 6.1 introduced the Eclipse-based OSATE environment for AADL and the translation of the AADL instance model to AIR. In this translation process, the internal OSATE representation of AADL resources is captured in the AIR. Furthermore, Awas develops a mapping from OSATE representation to Awas representation of components, connection, ports, and intra-component flows. These relationships enable the dependence analysis mentioned earlier performed on the graphical view of the instance model.

Figure 8.1 shows the forward dependence analysis on the AADL graphical view from the port `send_map`(selected by dashed border). On clicking the forward arrow toolbar button, Awas computes forward dependencies on the entire model and highlights the reachable ports on the graphical view of the instance model. This analysis is similar to the analysis performed on the visualizer presented in Figure 6.3.

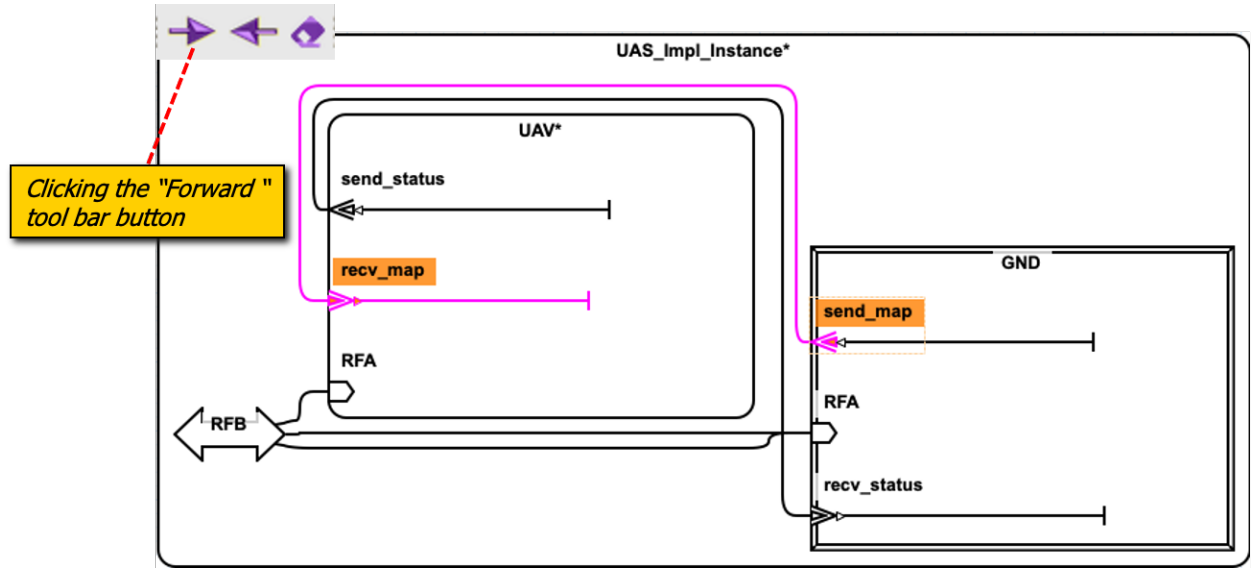


Figure 8.1: Forward Slice (interactive forward dependence query) on AADl Graphical view

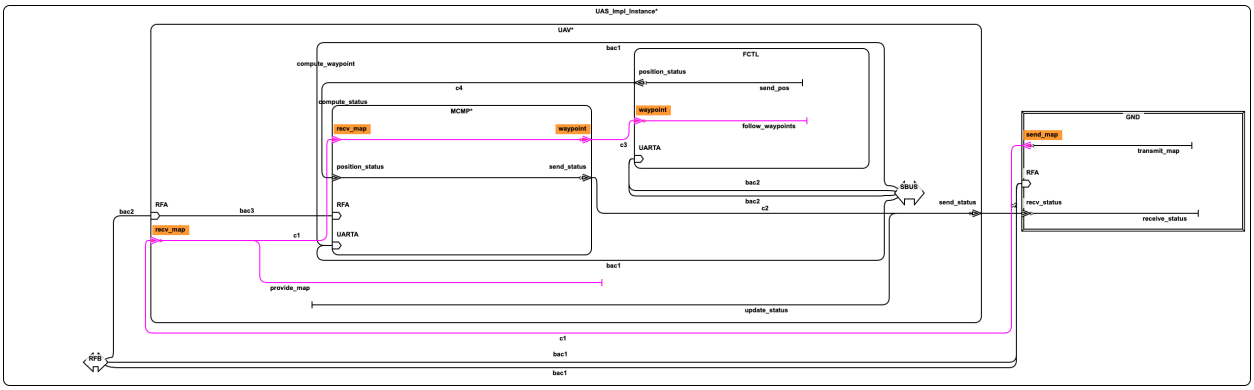


Figure 8.2: Forward Slice (interactive forward dependence query) on AADl Graphical view

The graphical view of the AADL Instance model can be customized to view subsystem components, intra-component flows, connections, and connections between the parent and its sub-components at a fine-grained level. OSATE integration can utilize all of these graphical elements to present the results of the dependence analysis. Figure 8.2 present the results of the same forward analysis, including the sub-components of the UAV component.

The ability to perform dependence analysis within the modeling environment provides quick feedback to the users. Albeit its advantages, the Awas visualizer provides a portable and interactive view of the system model without installing any other software.

8.1.1 Visualizer Integration

Section 6.1 describes the web-based interactive Awas visualizer and its capabilities. With the OSATE resources in the Awas graph, the visualizers can communicate with the OSATE Environment using the WebSocket protocol. This enables visualization of the query results in the AADL graphical view. Also, mimic all the interactions on the web-based viewer to the OSATE graphical viewer.

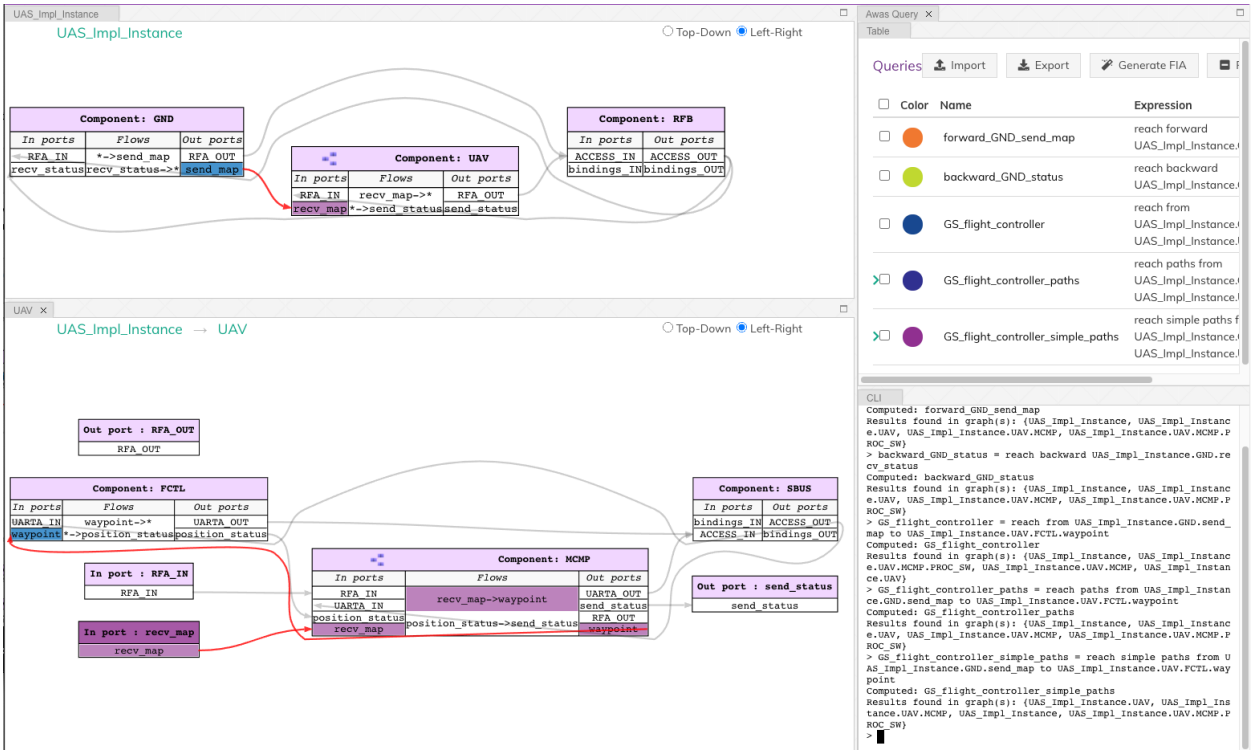


Figure 8.3: Result of Query Concept 4 in Awas Visualizer

Figure 8.3 shows the results executing the query concept 4 specified in section 6.3.4. When the WebSocket connection is active between the web page and the OSATE Environment. Awas furnishes the results in the AADL graphical view as shown in figure 8.3.

Apart from visualizing the results in the OSATE IDE, an active WebSocket connection can proactively update the visualizer when the AADL model changes. Similarly, it can navigate to the definition of an AADL element in the declarative model from the browser.

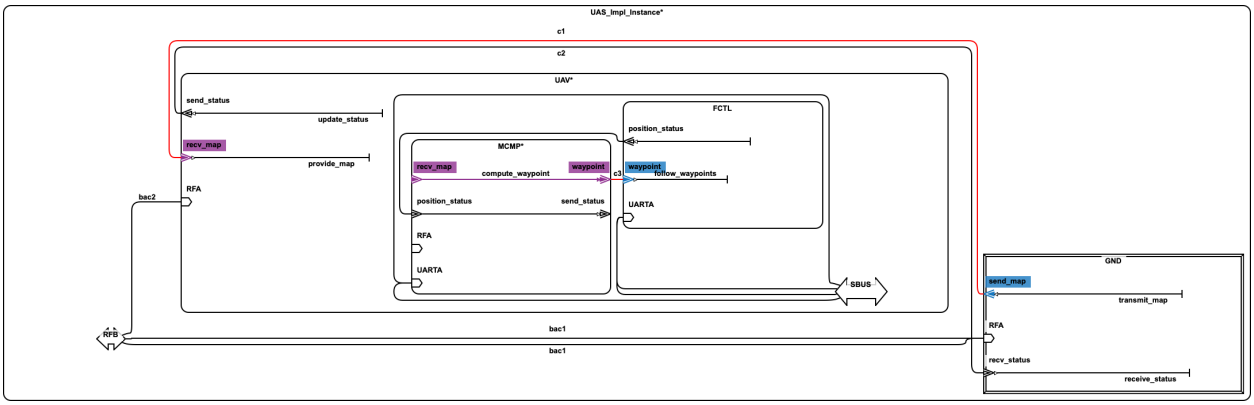


Figure 8.4: Result of Query Concept 4 in AADL graphical view

8.1.2 Alisa Integration

Architecture-Led Incremental System Assurance (ALISA) is a set of plug-ins to the OSATE toolset to support architecture-led system assurance. ALISA includes a requirement specification language called ReqSpec to support elicitation and modeling requirements for an AADL model. ReqSpec also supports the designing of verification plans and associates them with requirements. Finally, ALISA can execute these plans to produce evidence that the model satisfies the requirements.

ReqSpec supports two kinds of specification, (a) stakeholder goals representing individual stakeholder's requirements. There may be conflicts between goals and other requirements, and (b) system requirements intend to be verified and free of conflicts with other requirements. With the decomposition of the system into subsystem, the requirements may be refined into sub-requirements that are verifiable by the subsystem.

Figure 8.5 shows the requirements for the UAS model. The *AuthenticatedMap* captures that all map must go through a radio driver where the map is authenticated before reaching the flight controller. The second requirement states that the map shall not reach the flight controller without going through the filter component. The requirement also includes Awas queries corresponding to them.

The verification plan in figure 8.6 checks if there is a violating path. The requirements are satisfied if no violating path exists in the model and thus checks for an empty result for the query. Figure 8.7 shows the execution of the verification plan, which ensures that each of

```
requirement AuthenticatedMap [
    val authenticated = this.UAV.MCMP.PROC_SW.RADIO
    val flight_controller = this.UAV.FCTL
    val map = this.GND.send_map
    val authenticatedMapQuery = "authenticatedMapQuery = reach refined paths
                                            from UAS_Impl_Instance.GND.send_map
                                              to UAS_Impl_Instance.UAV.FCTL:port
                          with none(UAS_Impl_Instance.UAV.MCMP.PROC_SW.RADIO:port)"
    description "The map transmitted from ground station must be authenticated
                                            before reaching the flight_controller"
    category Quality.Security
]

requirement WellFormedMap [
    val wellformed = this.UAV.MCMP.PROC_SW.FLT
    val flight_controller = this.UAV.FCTL
    val map = this.GND.send_map
    val wellFormedMapQuery = "wellFormedMapQuery = reach refined paths
                                              from UAS_Impl_Instance.GND.send_map
                                                to UAS_Impl_Instance.UAV.FCTL:port
                                with none(UAS_Impl_Instance.UAV.MCMP.PROC_SW.FLT:port)"
    description "Only wellformed map from the ground station reaches
                                              the flight_controller"
```

Figure 8.5: ReqSpec requirement specification for the UAV model

the requirements is verified. A system analyst can develop a set of queries on the high-level model, and when the team of system designers develops the low-level details of the system, they can ensure that the requirements are not violated in the development process.

```
claim AuthenticatedMap [
    activities
        //UASQueries.AuthenticatedMap is empty
        AuthenticatedMap_empty: AwasMethods.isQueryEmpty(authenticatedMapQuery)
]

claim WellFormedMap [
    activities
       //UASQueries.WellFormedMap is empty
       WellFormedMap_empty: AwasMethods.isQueryEmpty(wellFormedMapQuery)
]
```

Figure 8.6: Verification plan for the UAV model

8.2 Evaluation

We evaluated Awas based on the reachability queries described in Section 6.3 by applying them to a collection of open source AADL models. As explained in Section 6.1, Awas al-

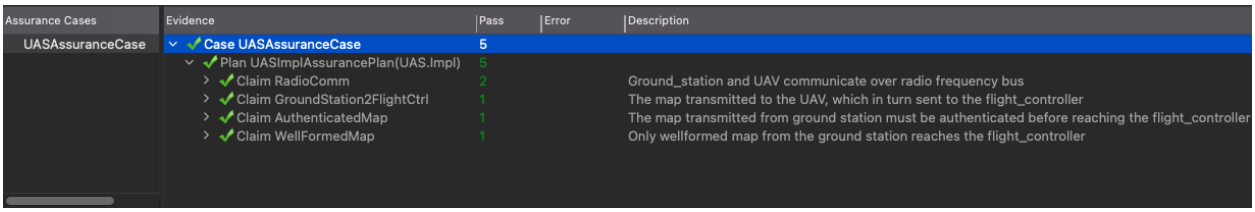


Figure 8.7: ALISA Assurance view

Models	No. of Graphs	Max Depth of Nested Graph	No. of Components	No. of Connections	No. of Ports	No. of Nodes	No. of Edges	No. of Intra-component Flows
Aircraft System	14	5	99	233	1052	399	466	22
Crazy Fly	2	2	16	25	125	50	50	24
Display Manager	11	3	39	48	284	154	96	0
Isolette	4	3	14	52	177	92	104	82
Open PCA Pump	19	5	80	457	1953	1018	914	250
Simple UAV	4	4	19	48	224	87	96	115
Speed Regulation	11	2	32	55	243	136	110	230
UAV - Phase 2	20	5	46	127	577	306	254	141
AFRL UxAS	2	2	37	102	803	142	204	0
Wheel Break System	26	6	168	674	2192	1177	1348	0

Table 8.1: Features of sample AADL models

gorithms are written in Scala and can be compiled via the Scala compiler to run on a Java Virtual Machine(JVM). This would be the typical Awas platform when Awas libraries are encapsulated and used by other AADL tools. In addition, when generating the HTML5 visualizations, the Scala code for Awas algorithms is translated to JavaScript using the Scala.js framework. Therefore, it is useful to understand the relative performance and scalability of the Awas algorithms running on the JVM and JavaScript platforms.

Table 8.1 presents the scale of each model and the characteristics that affect the queries'

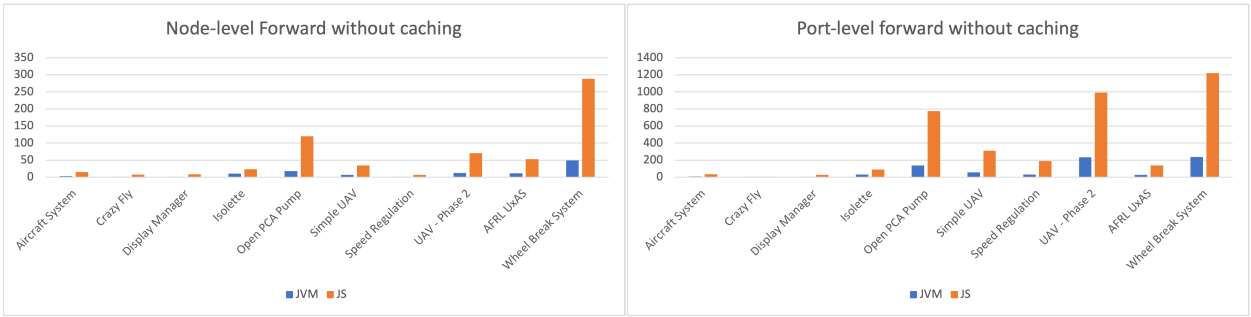


Figure 8.8: Forward Analysis

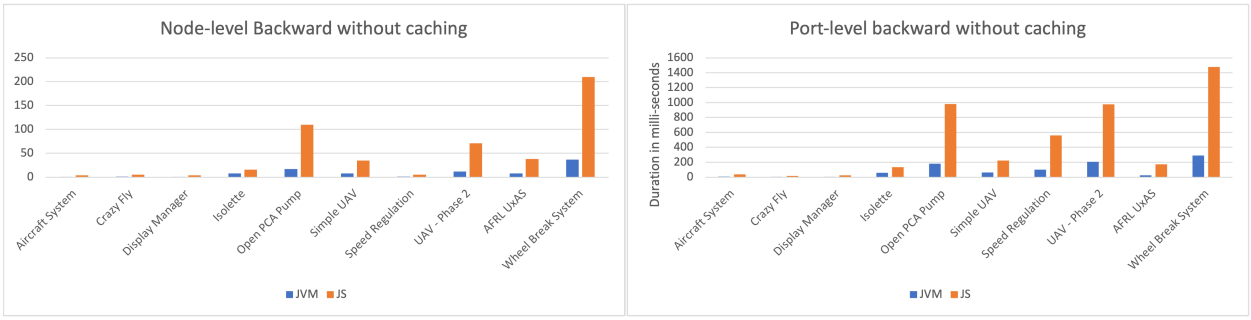


Figure 8.9: Backward Analysis

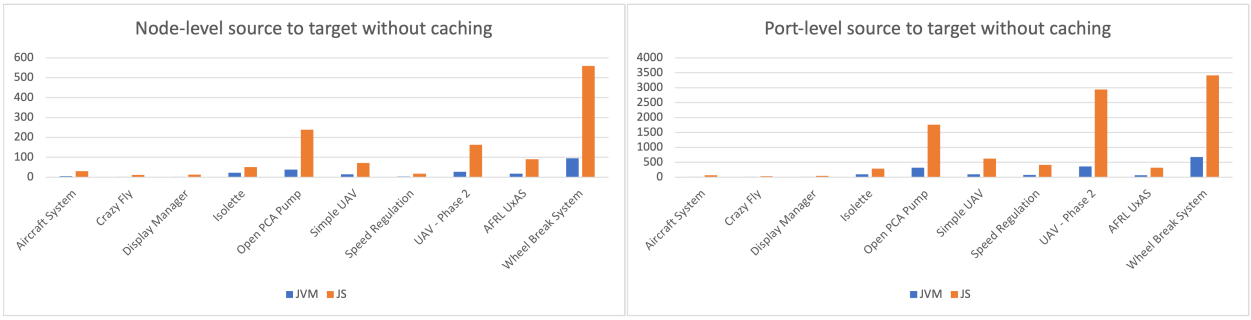


Figure 8.10: Source to Target Analysis

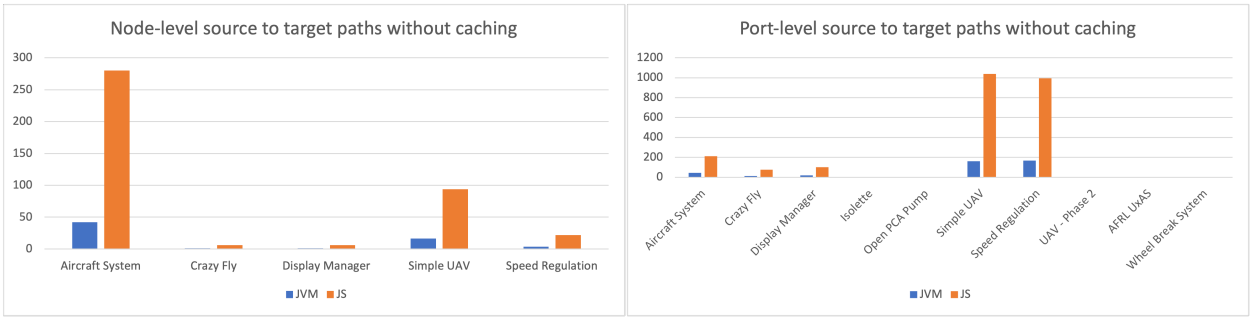


Figure 8.11: Source to Target With Paths Analysis

performance. We evaluated various queries in both the JVM and JavaScript platforms by generating thirty queries using a randomly picked criterion and evaluated each query thirty times to compute the average evaluation time for a model. We used the Google Chrome (version 95.0.4638.69) web browser for JavaScript as the execution platform. We evaluated both JVM and JavaScript versions on a MacBook Pro with a 2.4 GHz Intel Core i9 process and 64 GB memory.

As shown in Figures 8.8-8.10, Awas can perform reachability analysis instantaneously even on large industry models such as the *Wheel brake system*. Even on the largest models (*Wheel brake system* and *Open PCA Pump*), typical interactive queries complete in less than 3 seconds. The query category with the worst performance is path reachability, as shown in Figure 8.11 (lack of a bar indicates greater than one minute) due to the overall complexity of enumerating individual paths. The absence of intra-component flows, bus access features, and bindings relation are some of the factors that affect the performance of path reachability queries. When multiple components interact with a common bus, Awas computes the information flow from any component to all other components. The AADL core language cannot capture intra-component flows related to bindings.

In contrast, the EMv2 annex flows can capture the AADL binding flow relation. When EMv2 flows are provided, Awas utilizes this information to improve performance. In the model *Speed regulation*, path reachability computation is reasonably efficient due to the availability of intra-component flows and the lack of bindings. Elsewhere, in the *Isolette* model, port path reachability is noticeably slower due to the lack of intra-component flows (recall that in the absence of intra-component flow specifications, Awas soundly assumes that all inputs flow to all outputs).

Regarding the performance across both JVM and JavaScript platforms, our work demonstrates the feasibility of performing graph-based reachability analysis in a web browser due to the recent improvement in the JavaScript execution engines[84]. From Figures 8.8-8.10, the performance of Awas in JavaScript is approximately four times slower than the JVM. Although the performance difference is negligible in smaller models, in larger models, it is noticeable.

Figure 8.12: Performance Improvement in both JVM and JavaScript Platform

The algorithms presented in section 5.3 optimizes the reachability computation by caching the intermediate results of one query and using it in subsequent queries. The intermediate results correspond to the intra-component flows specified in the model. The effect of caching the flow of information in the subcomponent corresponding to the intra-component flow is more noticeable in forward port-level analysis than in backward port-level analysis. Figure 8.12 illustrates the performance improvement as a stacked bar chart. Models with subsystems and intra-component flows such as *Speed Regulation, Simple UAV, and UAV- Phase2* benefit the most, even up to 294% improvement. Overall, there is a 21% improvement among the ten models. In some models, performance declines amount to an average of 4.67% due to added checks for performing caching.

It is difficult to empirically evaluate the usability and effectiveness of a tool like Awas without a rigorous user case study. For anecdotal evidence, we note that Awas handles a large subset of AADL and has been applied to industrial scale models, including the Open PCA Pump models[85] – one of the largest and most complex publicly available AADL models (over 80 components, with 5-7 levels of architectural hierarchy). Once AADL flow annotations are added to a model, constructing an Awas visualization follows a very simple workflow: choose an option from an OSATE menu, specify a target folder, open the generated HTML index file in a browser. Our experience working with AADL on a number of projects is that even with small models, it is easy to lose "situational awareness" (e.g., "what other things is this port connected to and what does it influence within the system?"). That is, we have found Awas to be very useful to regain situational awareness and to support comprehension

of model structure. The Awas website[3] contains example models that can be immediately browsed, and these example artifacts are supported by detailed walkthroughs and videos.

Chapter 9

Future Work and Conclusions

AADL models capture many notions of dependence relevant for engineering safe and secure systems. However, the lack of tooling has been a barrier to effectively leveraging this information. With Awas, we have developed a framework that aggregates AADL's dependence information and provides analysis and visualization tools that enable engineers to better utilize that information for the development and assurance of realistic systems.

AADL and Awas can contribute to more rigorous engineering practices that address challenges in developing certified software, and systems[13]. This work demonstrated how the Awas AADL dependence analysis and visualization tool could be applied to support specific steps in AADL safety analysis and how reporting capabilities can be developed to support the ISO 14971 risk management process. We believe that the visualizations and error flow browsing capabilities can provide multiple practical benefits to practitioners working on full-scale systems.

Awas enabled both safety and security analysis under one tool to identify safety issues caused by security vulnerabilities. In security critical systems, this work provides a model specification and visualizations of flows between partitions in systems whose security properties are established using a micro-kernel and separation kernel foundations[52;53]

9.1 Extensions

Awas can be extended to support other forms of analysis such as timing and resource estimation and model and system information projection. This includes:

1. Supporting additional security analysis and threat modeling tasks that leverage model properties of components and connections added during security audits,

2. Visualization of coverage information (e.g., of ports, connections, and flow paths) from system tests and during live execution,

3. Visualizations of counter-example paths resulting from model-checking activities and deductive verification techniques in AADL[86],

4. Supporting probabilistic techniques can further enhance the precision of the analyses and provide quantitative risk evaluations, and

5. Integration of model-level information flows with source-code level information flows[87;88] and the ability to navigate freely between these.

9.2 Discussions

9.2.1 Is MBSE entirely model based?

The goal of MBSE is to support all the stages of the V-model using system architecture as illustrated in figure 9.1. However, in distributed development, the artifacts shared between the stakeholders of different organizations are mostly text-based documents. MBSE requires modeling software to view and interact with the model. The nontechnical member involved in the project either has to learn to use the modeling software or experience limited interaction with the model. In risk analysis, the report generated is independent of the model and opens up to the possibility of human error in understanding the safety concerns in the report and its implication on the model.

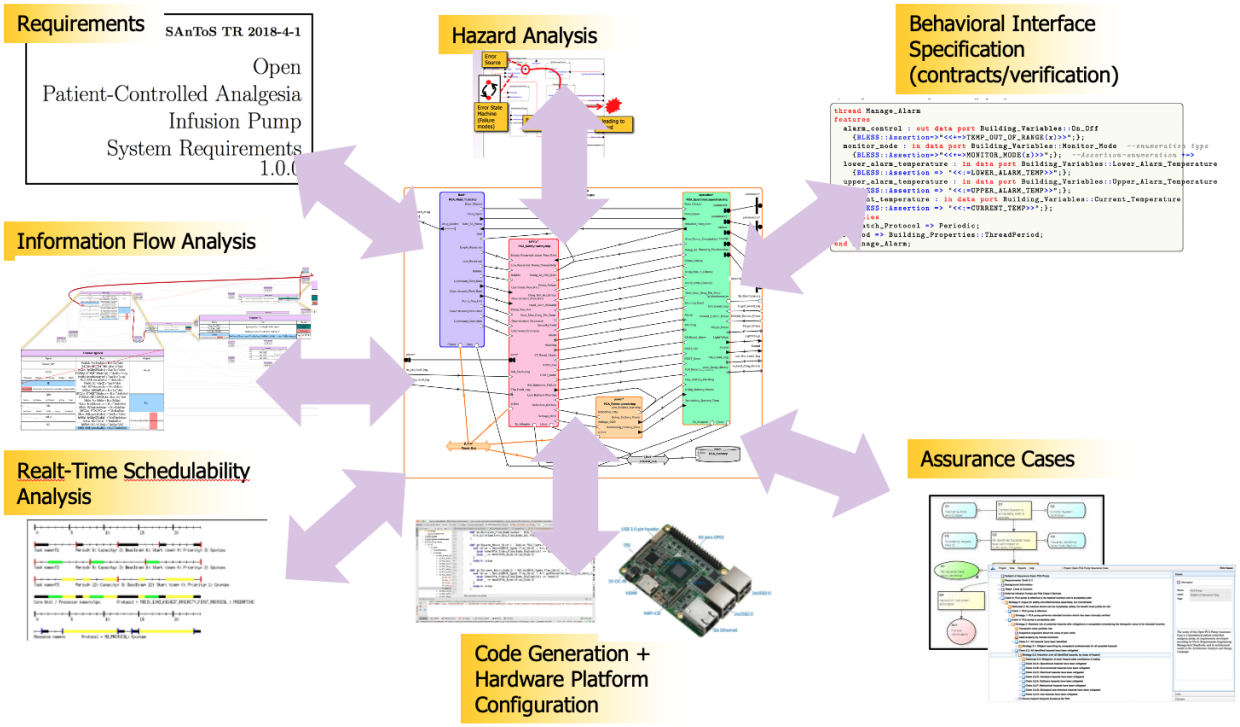


Figure 9.1: Model centric system engineering

A web-based modeling tool can eliminate some of the existing challenges in current MBSE practice. I consider Awas as the first step towards a web-based model analysis platform. Projects such as WebGME[89] provides a web-based modeling environment that is extensible to support other system engineering activities. Systems Engineering and Assurance Modeling (SEAM)[90] provides web-based assurance modeling and generation of Goal Structured Notation (GSN) diagrams. However, there is a need for an integrated web-based platform that supports requirements gathering and refinements, system modeling, code generation, hardware simulation, testing, and verification & validation of systems. A web-based MBSE platform with precise access control policies enables, multiple organizations to collaborate and communicate without compromising intellectual properties. Finally, the result of the risk management can be shared with the third-party evaluators without generating documents.

9.2.2 Can automated risk analysis tool be trusted?

As discussed in section 4.3, many of the automated risk analysis tools are based on the model checking technique. The risk analysis assumes that the underlying analysis platform will operate correctly. These model checking tools provide evidence in the form of a counter-example when the system violates a safety property. However, when the tool claims there is no violation of safety constraints, it gives little assurance. Past work from our research group specifically, Amtoft *et al.* has demonstrated an information flow analysis technique that is capable of providing a strong guarantee on the result of analysis[91].

Awas is part of the Kansas State University Sireum[92] framework, which includes the Slang language[66]. Slang is a subset of Scala that is designed for verification – it has a contract language with automated SMT-based verification support. We are planning to re-implement critical parts of the algorithms in Slang and use the Slang verification framework to establish fundamental correctness properties of the Awas algorithms. It is a significant challenge to develop an end-to-end verified, automated, and user-friendly tool.

Bibliography

[1] Peter H Feiler, Bruce Lewis, Steve Vestal, and Ed Colbert. An overview of the sae architecture analysis & design language (aadl) standard: A basis for model-based architecture-driven embedded systems engineering. In *IFIP World Computer Congress, TC 2*, pages 3–15. Springer, 2004. 2

[2] American Association of Medical Instruments (AAMI). AAMI TIR57: Principles for medical device information security risk management. https://www.aami.org/productspublications/ProductDetail.aspx?ItemNumber=3729, 2016. 3

[3] Hariharan Thiagarajan and John Hatcliff. Awas user documentation. URL http://awas.sireum.org/. https://awas.sireum.org. 5, 175

[4] Hariharan Thiagarajan, John Hatcliff, et al. Awas: Aadl information flow and error propagation analysis framework. *Innovations in Systems and Software Engineering*, pages 1–20, 2021. 6

[5] Hariharan Thiagarajan, John Hatcliff, and Robby. Awas: AADL information flow and error propagation analysis framework. In *ECSA Companion – Proceedings of the 2020 Workshop on "moDeling, vErification and Testing of dEpendable CriTical systems" (DETECT 2020).*, volume 1269 of *Communications in Computer and Information Science*, pages 294–310. Springer, 2020. 6

[6] Hariharan Thiagarajan, Brian Larson, John Hatcliff, and Yi Zhang. Model-based risk analysis for an open-source pca pump using aadl error modeling. In *International Symposium on Model-Based Safety and Assessment*, pages 34–50. Springer, 2020. 6

[7] Mark Weiser. The computer for the 21st century. *ACM SIGMOBILE mobile computing and communications review*, 3(3):3–11, 1999. 8

[8] Peter Marwedel. *Embedded system design: embedded systems foundations of cyber-physical systems, and the internet of things*. Springer Nature, 2021. 8

[9] Edward A Lee. Computing foundations and practice for cyber-physical systems: A preliminary report. *University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72*, 21, 2007. 9

[10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The internet of things: A survey. *Computer networks*, 54(15):2787–2805, 2010. 10

[11] Howard Eisner. *Essentials of project and systems engineering management*. John Wiley & Sons, 2008. 10

[12] Paul Rook. Controlling software projects. *Software engineering journal*, 1(1):7–16, 1986. 10

[13] John Hatcliff, Alan Wassyng, Tim Kelly, Cyrille Comar, and Paul L. Jones. Certifiably safe software-dependent systems: Challenges and directions. In *Proceedings of the on Future of Software Engineering (ICSE FOSE)*, pages 182–200, 2014. doi: 10.1145/2593882.2593895. 12, 176

[14] Nancy G. Leveson. *Engineering a Safer World*. Engineering Systems. MIT Press, 2011. ISBN 978-0-262-01662-9. URL http://mitpress.mit.edu/sites/default/files/titles/free_download/9780262016629_Engineering_a_Safer_World.pdf. 12, 39, 44

[15] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer, 2006. 14

[16] N. Shevchenko. An introduction to model-based systems engineering (mbse). Carnegie Mellon University's Software Engineering Institute Blog, 2020. URL http://insights.sei.cmu.edu/blog/introduction-model-based-systems-engineering-mbse/. 15

[17] Azad M Madni and Shatad Purohit. Economic analysis of model-based systems engineering. *Systems*, 7(1):12, 2019. 16

[18] John C Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th international conference on software engineering*, pages 547–550, 2002. 16

[19] Sam Procter, John Hatcliff, and Robby. Towards an AADL-based definition of app architecture for medical application platforms. In *SEHC Workshop*, 2014. URL http://samprocter.com/wp-content/uploads/2014/06/sehc14-aadl-for-map-apps.pdf. 17

[20] J. Hatcliff, A. King, I. Lee, M. Robkin, E. Vasserman, A. MacDonald, S. Weininger, A. Fernando, and J. M. Goldman. Rationale and architecture principles for medical application platforms. In *Proceedings of 2012 IEEE/ACM International Conference onCyber-Physical Systems (ICCPS)*, pages 3–12, 2012. 17

[21] J. Hatcliff, E. Y. Vasserman, T. Carpenter, and R. Whillock. Challenges of distributed risk management for medical application platforms. In *2018 IEEE Symposium on Product Compliance Engineering (ISPCE)*, pages 1–14, May 2018. 17, 21, 30

[22] ASTM Committee F-29, Anaesthetic and Respiratory Equipment, Subcommittee 21, Devices in the integrated clinical environment. Medical devices and medical systems — essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE), 2009. 17

[23] Center for Integration of Medicine and Innovative Technology (CIMIT). Strategic Initiative on Integrated Clinical Environments (ICE). URL http://www.cimit.org/programs-integrated-clinical-environments.html. 17

[24] MDPnP Program. Medical Device Plug-and-Play Program - ICE Details. URL http://www.mdpnp.org/mdice.html. http://www.mdpnp.org/mdice.html. 17

[25] Yu Jin Kim, Sam Procter, John Hatcliff, Venkatesh-Prasad Ranganath, and Robby.

Ecosphere principles for medical application platforms. In *IEEE International Conference on Healthcare Informatics (ICHI)*, 2015. 17, 30

[26] US Food and Drug Administration. Examples of Reported Infusion Pump Problems, . URL https://www.fda.gov/medical-devices/infusion-pumps/examples-reported-infusion-pump-problems. 19

[27] US Food and Drug Administration. Infusion Pump Improvement Initiative, . URL https://www.fda.gov/medical-devices/infusion-pumps/infusion-pump-improvement-initiative. 20

[28] Center for Devices and Radiological Health. Infusion Pumps Total Product Life Cycle–Guidance for Industry and FDA Staff. Technical Report FDA-2010-D-0194, US Food and Drug Administration, 2014. 20

[29] PE Macintyre. Safety and efficacy of patient-controlled analgesia. *British journal of anaesthesia*, 87(1):36–46, 2001. 20

[30] David Arney, Miroslav Pajic, Julian M. Goldman, Insup Lee, Rahul Mangharam, and Oleg Sokolsky. Toward patient safety in closed-loop medical device systems. In *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems*, ICCPS '10, pages 139–148, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0066-7. doi: 10.1145/1795194.1795214. URL http://doi.acm.org/10.1145/1795194.1795214. xiii, 21, 26

[31] David Arney, Sebastian Fischmeister, Julian M Goldman, Insup Lee, and Robert Trausmuth. Plug-and-play for medical devices: Experiences from a case study. *Biomedical Instrumentation & Technology*, 43(4):313–317, 2009. 21

[32] Object Management Group. Systems modeling language (bdd sysml) version 1.3. http://sysml.org/docs/specs/OMGSysML-v1.3-12-06-02.pdf, 2012. viii, 23

[33] Simulink Documentation. Simulation and model-based design, 2020. URL https://www.mathworks.com/products/simulink.html. ix, 23

[34] MATLAB. *version 7.10.0 (R2010a)*. The MathWorks Inc., Natick, Massachusetts, 2010. 23

[35] Algirdas Avizienis, J.-C. Laprie, Brian Randell, and Carl Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 33, 34, 35, 72, 80, 148

[36] Brian Larson, John Hatcliff, Kim Fowler, and Julien Delange. Illustrating the AADL error modeling annex (v. 2) using a simple safety-critical medical device. *ACM SIGAda Ada Letters*, 33(3):65–84, 2013. 34, 143

[37] SAE AS-2C Architecture Description Language Subcommittee. SAE Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Language. Technical report, SAE Aerospace, June 2014. 34

[38] John Hatcliff, Brian R. Larson, Jason Belt, Robby, and Yi Zhang. A unified approach for modeling, developing, and assuring critical systems. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification and Validation. Modeling*, pages 225–245, Cham, 2018. Springer International Publishing. ISBN 978-3-030-03418-4. 38

[39] Clifton A Ericson II. *Hazard analysis techniques for system safety*. John Wiley & Sons, 2005. 38

[40] S. Procter and J. Hatcliff. An architecturally-integrated, systems-based hazard analysis for medical applications. In *2014 Twelfth ACM/IEEE Conference on Formal Methods and Models for Codesign (MEMOCODE)*, pages 124–133, Oct 2014. xiii, 40

[41] B. Larson, J. Hatcliff, K. Fowler, and J. Delange. Illustrating the aadl error modeling annex (v.2) using a simple safety-critical medical device. In *Proceedings of the 2013 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '13, pages 65–84, New York, NY, 2013. ACM. 41

[42] Venkatesh-Prasad Ranganath, Yu Jin Kim, John Hatcliff, and Robby. Communication patterns for interconnecting and composing medical systems. In *37th Annual International Conference of the IEEE Engineering in Medicine and Biology Society, EMBC 2015, Milan, Italy, August 25-29, 2015*, pages 1711–1716, 2015. doi: 10.1109/EMBC.2015.7318707. 48

[43] AAMI ANSI. Aami/iec 80001-1: 2010, application of risk management for it network incorporating medical devices-part 1: Roles, responsibilities and activities. *Association for the Advancement of Medical Instrumentation. Arlington, Va*, 2010. xvii, 49, 72, 73, 75

[44] Danny Dolev and Andrew Yao. On the security of public key protocols. *IEEE Transactions on information theory*, 29(2):198–208, 1983. 50

[45] Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The MILS component integration approach to secure information sharing. In *Digital Avionics Systems Conference, 2008. DASC 2008. IEEE/AIAA 27th*, pages 1–C. IEEE. URL http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=4702758. 51

[46] Jim Alves-Foss, Paul W. Oman, Carol Taylor, and W. Scott Harrison. The MILS architecture for high-assurance embedded systems. *International Journal of Embedded Systems*, 2(3):239–247, 2006. URL http://inderscience.metapress.com/index/C42N6765P7016074.pdf. 51

[47] Carolyn Boettcher, Rance DeLong, John Rushby, and Wilmar Sifre. The mils component integration approach to secure information sharing. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1–C. IEEE, 2008. 52

[48] Steven Rasmussen, Derek Kingston, and Laura R. Humphrey. A brief introduction to unmanned systems autonomy services (uxas). *2018 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 257–268, 2018. 55

[49] Andera Bondavalli and Luca Simoncini. Failure classification with respect to detection. In *Distributed Computing Systems, 1990. Proceedings., Second IEEE Workshop on Future Trends of*, pages 47–53. IEEE, 1990. 65

[50] S. Procter, E. Y. Vasserman, and J. Hatcliff. Safe and secure: Deeply integrating security in a new hazard analysis. In *Proceedings of ASSURE 2018 international workshop on assurance cases for software-intensive systems*, pages 1–10, September 2018. xvii, 67, 68, 148

[51] Sam Procter, Eugene Y. Vasserman, and John Hatcliff. SAFE and secure: Deeply integrating security in a new hazard analysis. In *Proceedings of the 12th International Conference on Availability, Reliability and Security, Reggio Calabria, Italy, August 29 - September 01, 2017*, pages 66:1–66:10, 2017. doi: 10.1145/3098954.3105823. URL http://doi.acm.org/10.1145/3098954.3105823. 68

[52] Todd Carpenter, John Hatcliff, and Eugene Y. Vasserman. A reference separation architecture for mixed-criticality medical and iot devices. In *Proceedings of the ACM Workshop on the Internet of Safe Things (SafeThings)*. ACM, November 2017. 86, 176

[53] Brian R. Larson, Paul Jones, Yi Zhang, and John Hatcliff. Principles and benefits of explicitly designed medical device safety architecture. *Biomedical Instrumentation & Technology*, 51(5):380–389, 2017. 89, 176

[54] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific journal of Mathematics*, 5(2):285–309, 1955. 101

[55] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993. 101

[56] Dorothy E Denning. A lattice model of secure information flow. *Communications of the ACM*, 19(5):236–243, 1976. 102

[57] E Allen Emerson. Temporal and modal logic. In *Formal Models and Semantics*, pages 995–1072. Elsevier, 1990. 104

[58] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, 1991. 105

[59] Andrew Gacek, John Backes, Mike Whalen, Lucas Wagner, and Elaheh Ghassabani. The jk ind model checker. In *International Conference on Computer Aided Verification*, pages 20–27. Springer, 2018. 105

[60] Kenneth L McMillan. Circular compositional reasoning about liveness. In *Advanced Research Working Conference on Correct Hardware Design and Verification Methods*, pages 342–346. Springer, 1999. 105

[61] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: an assurance case language for architecture models. *ACM SIGAda Ada Letters*, 34(3): 19–28, 2014. 106

[62] Pierre Bieber, Christian Bougnol, Charles Castel, Jean-Pierre Heckmann Christophe Kehren, Sylvain Metge, and Christel Seguin. Safety assessment with altarica. In *Building the Information Society*, pages 505–510. Springer, 2004. 107

[63] J. Brunel, P. Feiler, J. Hugues, B. Lewis, T. Prosvirnova, C. Seguin, and L. Wrage. Performing safety analyses with aadl and altarica. In *Proceedings of 4th International Symposium on Model-Based Safety and Assessment*, pages 67–81, 2017. 107

[64] Kryštof Hoder and Nikolaj Bjørner. Generalized property directed reachability. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 157–171. Springer, 2012. 108

[65] Thomas Reps and Genevieve Rosay. Precise interprocedural chopping. *ACM SIGSOFT Software Engineering Notes*, 20(4):41–52, 1995. 129

[66] John Hatcliff et al. Slang: The sireum programming language. In *International Symposium on Leveraging Applications of Formal Methods*, pages 253–273. Springer, 2021. 131, 179

[67] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Sebastian Maneth, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. An overview of the scala programming language. Technical report, Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004. 132

[68] Sébastien Doeraene. Scala. js: Type-directed interoperability with dynamically typed languages. *Rapport technique EPFL-REPORT-190834, École polytechnique fédérale de Lausanne*, 2013. 132

[69] Malcolm Wallace. Modular architectural representation and analysis of fault propagation and transformation. *Electronic Notes in Theoretical Computer Science*, 141(3): 53–71, 2005. 134, 143

[70] Hariharan Thiagarajan, Brian Larson, John Hatcliff, and Yi Zhang. Model-based risk analysis for an open-source PCA pump using AADL error modeling. In *Proceedings of the International Conference on Model-based Safety Analysis*, Sep 2020. 135

[71] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981. 139

[72] Daniel Jackson and Eugene J Rollins. Chopping: A generalization of slicing. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA DEPT OF COMPUTER SCIENCE, 1994. 141

[73] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 49–61. ACM, 1995. 141

[74] Gary A Kildall. A unified approach to global program optimization. In *Proceedings of the 1st annual ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 194–206. ACM, 1973. 142

[75] Julien Delange and Peter Feiler. Architecture fault modeling with the AADL error-model annex. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 361–368. IEEE, 2014. 143

[76] Awas. Sireum Awas web site. https://awas.sireum.org, 2018. 154

[77] Andrei Sabelfeld and Andrew C Myers. Language-based information-flow security. *IEEE Journal on selected areas in communications*, 21(1):5–19, 2003. 158

[78] Andrew C Myers and Barbara Liskov. A decentralized model for information flow control. *ACM SIGOPS Operating Systems Review*, 31(5):129–142, 1997. 158

[79] Anindya Banerjee and David A Naumann. Secure information flow and pointer confinement in a java-like language. In *CSFW*, volume 2, page 253, 2002.

[80] Steve Zdancewic and Andrew C Myers. Secure information flow and cps. In *European Symposium on Programming*, pages 46–61. Springer, 2001.

[81] Andrei Sabelfeld and David Sands. Probabilistic noninterference for multi-threaded programs. In *Proceedings 13th IEEE Computer Security Foundations Workshop. CSFW-13*, pages 200–214. IEEE, 2000. 158

[82] Hariharan Thiagarajan. *Dependence analysis for inferring information flow properties in Spark ADA programs*. PhD thesis, Kansas State University, 2011. 163

[83] Christian Hammer, Jens Krinke, and Frank Nodes. Intransitive noninterference in dependence graphs. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (isola 2006)*, pages 119–128. IEEE, 2006. 163

[84] David Herrera, Hangfen Chen, Erick Lavoie, and Laurie Hendren. Webassembly and javascript challenge: Numerical program performance using modern browser technologies and devices. Technical report, Technical Report. Technical report SABLE-TR-2018-2. Montréal, Québec, Canada . . . , 2018. 173

[85] J. Hatcliff, B. Larson, T. Carpenter, P. Jones, Y. Zhang, and J. Jorgens. The Open PCA Pump Project: An exemplar open source medical device as a community resource. *SIGBED Rev.*, page 8–13, August 2019. 174

[86] Brian Larson, Patrice Chalin, and John Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. In *Proceedings of the 2013 NASA Formal Methods Conference*, volume 7871 of *Lecture Notes in Computer Science*, pages 276–290. Springer, 2013. 177

[87] Venkatesh Prasad Ranganath and John Hatcliff. Slicing concurrent java programs using Indus and Kaveri. *STTT*, 9(5-6):489–504, 2007. 177

[88] Hariharn Thiagarajan, John Hatcliff, Jason Belt, and Robby. Bakar Alir: Supporting developers in construction of information flow contracts in SPARK. In *2012 IEEE 12th International Working Conference on Source Code Analysis and Manipulation*, pages 132–137, 2012. 177

[89] WebGME. URL https://webgme.org/. 178

[90] Kaitlyn L Ryder, Ryan Alles, Gabor Karsai, Nagabhushan Mahadevan, John W Evans, Arthur F Witulski, Michael J Campola, Rebekah A Austin, and Ronald D Schrimpf. Systems engineering and assurance modeling (seam): A web-based solution for integrated mission assurance. *Facta Universitatis, Series: Electronics and Energetics*, 34 (1):001–020, 2021. 178

[91] Torben Amtoft, Josiah Dodds, Zhi Zhang, Andrew Appel, Lennart Beringer, John Hatcliff, Xinming Ou, and Andrew Cousino. A certificate infrastructure for machine-checked proofs of conditional information flow. In *International Conference on Principles of Security and Trust*, pages 369–389. Springer, 2012. 179

[92] Sireum. URL http://sireum.org/. 179

Appendix A

Query Language Grammar

```
modelFile : model EOF

model : queryStatement+

queryStatement : ID '=' queryExpr

setOp : '-' | 'union' | 'intersect'

queryExpr : 'reach' reachExpr ((setOp queryExpr) | ':' filter)?
            | primaryExpr ((setOp queryExpr) | ':' filter)?

reachExpr : ('forward' | 'backward') queryExpr
            | 'from' queryExpr 'to' queryExpr
            | ('refined')? ('simple')? 'paths from' queryExpr 'to' queryExpr ('with' withExpr)*

withExpr : ('some' | 'all' | 'none') '(' queryExpr ')'

primaryExpr : nodeNameError
            | '(' queryExpr ')'
            | '{' nodeNameError (',' nodeNameError)+ '}'
```

```
filter : node
       | port-error
       | port
       | in-port
       | out-port
       | error
       | source
       | sink

nodeNameError :  nodeName error?

nodeName : ID('.' ID)*

error : '{' errorId (',' errorId)* '}'

errorId : ID ('.' ID)*

ID : ([A-Z] | [a-z]) ([A-Z] | [a-z] | [0-9]| '_')*
```

Appendix B

ISO 14971

```
property set ISO14971_80001 is
with EMV2;

  SystemInfo: record (
    Name : aadlstring;                                          --required
    Description: aadlstring;                                    --optional
    IntendedUse: aadlstring;                                    --optional
  ) applies to (system);

  Causes: list of ISO14971_80001::Cause applies to ({emv2}**error source, {emv2}**error
      type, {emv2}**error behavior state, {emv2}**error event);

  Hazardous_Situations: list of ISO14971_80001::Hazardous_Situation applies to (system);

  Hazards: list of ISO14971_80001::Hazard applies to ({emv2}**error type, {emv2}**error
      behavior state);

  Harm: type record (
    ID: aadlstring;                                             -- required
    Description: aadlstring;                                    -- optional
    Severity: ISO14971_80001::SeverityScales;                   -- optional
```

```
            -- e.g. Catastrophic, High, Medium, Low, Negligible
);

Cause: type record (
  ID : aadlstring;                                                    -- required
  Description : aadlstring;                                           -- optional
  Basis :  ISO14971_80001::ProbabilityBasis;
          -- Causes per hour, or causes per number of occurrences
  NumberOfOccurrencesPerCause : aadlinteger;
          -- how many occurrences are expected to produce one hazard?
  Probability : ISO14971_80001::ProbabilityScales;                    -- optional
);        -- Frequent, Probable, occasional, remote, improbable

Hazard: type record (
  ID : aadlstring;
          -- hazard unique identifier
  Description : aadlstring;                                           -- optional
          -- description of the hazard eg: opioid
  Causes : list of ISO14971_80001::Cause;                             -- Computed
);

--circumstance in which people, property, or the environment are exposed to one or
    more hazard(s)
Hazardous_Situation: type record (
  ID: aadlstring;
  Description : aadlstring;                                           -- optional
  Hazard :  ISO14971_80001::Hazard;
  Severity : ISO14971_80001::SeverityScales;                          -- computed
  Paths_to_harm : list of record (
    Harm: ISO14971_80001::Harm;
        -- e.g. patient overdosed/fatality
    Contributing_Factors: list of ISO14971_80001::Contributing_Factor;
        -- e.g. Patient vitals are deteriorating
```

```
52      Probability_of_Transition: ISO14971_80001::ProbabilityScales;          -- optional
53    );                                                                       -- optional
54    Risk : ISO14971_80001::RiskLevels;                                       -- computed
55    Probability : ISO14971_80001::ProbabilityScales;                         -- optional
56  );
57
58  Contributing_Factor : type record (
59    ID : aadlstring;
60    Description : aadlstring;                                                --optional
61  );
62
63  Risk_Control : record (
64    ID : aadlstring;
65    Description : aadlstring;
66    Effective_Probability: ISO14971_80001::ProbabilityScales;
67          -- probability that the risk control mitigates an incoming error;
68  ) applies to ({emv2}**error behavior state,
69                 {emv2}**error event, {emv2}**error flow, {emv2}**error propagation);
70
71 ------------------------------------------------------
72 ----------------------Scales--------------------------
73 ------------------------------------------------------
74
75  SeverityScales: type enumeration (Catastrophic, High, Medium, Low, Negligible,
        Critical, Serious, Minor, NoEffect);
76            --for ISO 80001
77  Catatrophic: constant EMV2::SeverityRange => 1;
78  High: constant EMV2::SeverityRange => 2;
79  Medium: constant EMV2::SeverityRange => 3;
80  Low : constant EMV2::SeverityRange => 4;
81  Negligible  : constant EMV2::SeverityRange => 5;
82            --ISO 14971 uses some different terms for severity
83  Critical : constant EMV2::SeverityRange => 2;
```

```
84              -- Results in permanent impairment or life-threatening injury
85    Serious : constant EMV2::SeverityRange => 3;
86              -- Results in injury or impairment requiring professional medical
                    intervention
87    Minor : constant EMV2::SeverityRange => 4;
88              -- Results in temporary injury or impairment not requiring professional
                    medical intervention
89    NoEffect : constant EMV2::SeverityRange => 5;
90              -- same as Negligible because EMV2::SeverityRange = [1..5]
91
92    ProbabilityScales: type enumeration (Frequent, Probable, Occasional, Remote,
          Improbable);
93
94    Frequent: constant EMV2::LikelihoodLabels => A;
95    Probable: constant EMV2::LikelihoodLabels => B;
96    Occasional: constant EMV2::LikelihoodLabels => C;
97    Remote: constant EMV2::LikelihoodLabels => D;
98    Improbable: constant EMV2::LikelihoodLabels => E;
99
100   RiskLevels : type enumeration (High, Moderate, Low);
101             -- ISO 14971 allows risks to be quantified as the number of uses for each
                    adverse event, on average
102   ProbabilityBasis: type enumeration (CausesPerHour, NumberOfOccurrencesPerCause);
103
104 end ISO14971_80001;
```

Appendix C

Security Modeling Framework Annex Grammar

```
SecMFRoot returns aadl2::NamedElement:
  {SecMFRoot} (library=SMFLibrary | subclauses+=SMFSubclause*)

AnnexLibrary returns aadl2::AnnexLibrary:
  SecModelLibrary

AnnexSubclause returns aadl2::AnnexSubclause:
  SecModelSubclause

NamedElement returns aadl2::NamedElement:
  SecModelLibrary   |  SMFClassification | SMFDeclassification  | SMFTypeDef | SMFTypeRef

SecModelLibrary returns SecModelLibrary:
  {SecModelLibrary}
  (DomainTypesKeywords
    (types+=SMFTypeDef)*
  EndTypesKeywords ';'
  )?
```

```
SMFLibrary returns SecModelLibrary:
  {SecModelLibrary}
  ('library' name=QEMREF
    (
      DomainTypesKeywords
      (types+=SMFTypeDef)*
      EndTypesKeywords ';'
    )?
  ) |
  ('package' name=QEMREF 'public'
  'annex' ID  '{**'
  (
    DomainTypesKeywords
    (type+=SMFTypeDef)*
    EndTypesKeywords ';'
  )?
  '**}' ';' 'end' QEMREF ';'
  )

Element returns aadl2::Element:
   SMFTypeRef

ModalElement returns aadl2::ModalElement:
  SecModelSubclause

SMFSubclause returns SecModelSubclause:
  {SecModelSubclause} 'subclause' name=QCREF
  (ClassificationKeywords (classification += SMFClassification)+)?
  (DeclassificationKeywords (declassification += SMFDeclassification)+)?

SecModelSubclause returns SecModelSubclause:
  {SecModelSubclause}
  (ClassificationKeywords (classification += SMFClassification)+)?
```

```
  (DeclassificationKeywords (declassification += SMFDeclassification)+)?

SMFClassification returns SMFClassification:
  {SMFClassification}
  (feature=[aadl2::NamedElement| ID])
  ':' typeRef = [SMFTypeRef|ID]
  ';'

SMFTypeRef returns SMFTypeRef:
  SMFTypeDef

SMFDeclassification returns SMFDeclassification:
  {SMFDeclassification}
  (flow=[aadl2::NamedElement])
  ':' (srcName=[SMFTypeRef|ID] | any?='any') '->' snkName=[SMFTypeRef|ID] ';'

SMFTypeDef returns SmfTypeDef:
  name=ID (
    (':' 'type')
    |(':' 'type' 'extends' type+=[SMFTypeRef|ID] (','type+=[SMFTypeRef|ID])*)
  )
  ';'

ClassificationKeywords:
  'classification'

DeclassificationKeywords:
  'de-classification'

DomainTypesKeywords:
  'domain' 'types'

EndTypesKeywords:
```

```
  'end' 'types'

terminal SL_COMMENT:
  '--' !('\n' | '\r')* ('\r'? '\n')?;

terminal INTEGER_LIT : ('0'..'9')+;

QUALIFIEDNAME: ID ('.' ID)+;

QEMREF: ID ('::' ID)* ;//('.' ID )?;
```