A DEVELOPMENT AND ASSURANCE PROCESS FOR MEDICAL APPLICATION PLATFORM APPS

by

SAM PROCTER

B.S., University of Nebraska – Lincoln, 2009

M.S., Kansas State University, 2011

AN ABSTRACT OF A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Abstract

Medical devices have traditionally been designed, built, and certified for use as monolithic units. A new vision of "Medical Application Platforms" (MAPs) is emerging that would enable compositional medical systems to be instantiated at the point of care from a collection of trusted components. This work details efforts to create a development environment for applications that run on these MAPs.

The first contribution of this effort is a language and code generator that can be used to model and implement MAP applications. The language is a subset of the Architecture, Analysis and Design Language (AADL) that has been tailored to the platform-based environment of MAPs. Accompanying the language is software tooling that provides automated code generation targeting an existing MAP implementation.

The second contribution is a new hazard analysis process called the Systematic Analysis of Faults and Errors (SAFE). SAFE is a modified version of the previously-existing System Theoretic Process Analysis (STPA), that has been made more rigorous, partially compositional, and easier. SAFE is not a replacement for STPA, however, rather it more effectively analyzes the hardware- and software-based elements of a full safety-critical system. SAFE has both manual and tool-assisted formats; the latter consists of AADL annotations that are designed to be used with the language subset from the first contribution. An automated report generator has also been implemented to accelerate the hazard analysis process.

Third, this work examines how, independent of its place in the system hierarchy or the precise configuration of its environment, a component may contribute to the safety (or lack thereof) of an entire system. Based on this, we propose a reference model which generalizes notions of harm and the role of components in their environment so that they can be applied to components either in isolation or as part of a complete system. Connections between these formalisms and existing approaches for system composition and fault propagation are also established.

This dissertation presents these contributions along with a review of relevant literature, evaluation of the SAFE process, and concludes with discussion of potential future work.

A DEVELOPMENT AND ASSURANCE PROCESS FOR MEDICAL APPLICATION PLATFORM APPS

by

Sam Procter

B.S., University of Nebraska – Lincoln, 2009

M.S., Kansas State University, 2011

A DISSERTATION

submitted in partial fulfillment of the
requirements for the degree

DOCTOR OF PHILOSOPHY

Department of Computing and Information Sciences
College of Engineering

KANSAS STATE UNIVERSITY
Manhattan, Kansas

2016

Approved by:

Major Professor
John Hatcliff

Abstract

Medical devices have traditionally been designed, built, and certified for use as monolithic units. A new vision of "Medical Application Platforms" (MAPs) is emerging that would enable compositional medical systems to be instantiated at the point of care from a collection of trusted components. This work details efforts to create a development environment for applications that run on these MAPs.

The first contribution of this effort is a language and code generator that can be used to model and implement MAP applications. The language is a subset of the Architecture, Analysis and Design Language (AADL) that has been tailored to the platform-based environment of MAPs. Accompanying the language is software tooling that provides automated code generation targeting an existing MAP implementation.

The second contribution is a new hazard analysis process called the Systematic Analysis of Faults and Errors (SAFE). SAFE is a modified version of the previously-existing System Theoretic Process Analysis (STPA), that has been made more rigorous, partially compositional, and easier. SAFE is not a replacement for STPA, however, rather it more effectively analyzes the hardware- and software-based elements of a full safety-critical system. SAFE has both manual and tool-assisted formats; the latter consists of AADL annotations that are designed to be used with the language subset from the first contribution. An automated report generator has also been implemented to accelerate the hazard analysis process.

Third, this work examines how, independent of its place in the system hierarchy or the precise configuration of its environment, a component may contribute to the safety (or lack thereof) of an entire system. Based on this, we propose a reference model which generalizes notions of harm and the role of components in their environment so that they can be applied to components either in isolation or as part of a complete system. Connections between these formalisms and existing approaches for system composition and fault propagation are also established.

This dissertation presents these contributions along with a review of relevant literature, evaluation of the SAFE process, and concludes with discussion of potential future work.

Table of Contents

List of Figures

List of Tables

Acknowledgments

First and foremost I would like to thank my major professor, Dr. John Hatcliff, for all the advice, help, and support he has given me throughout the course of my PhD work. I feel very fortunate to have been able to study under him, Dr. Robby, and the other faculty in the SAnToS Laboratory, and to have worked with all the students that have been members. I would also like to thank my committee members for their feedback on this research; I have in particular appreciated Dr. Eugene Vasserman's kind words and unique perspective on the challenges of system safety. Similarly, the advice of Dr. David Schmidt has been a great help at various points throughout my graduate studies.

My family—both my parents, as well as my brother and his husband—have been a source of great support and inspiration, and I would not have made it nearly as far as I have without them. Similarly, my friends—in town and across the country, in graduate school and in industry—have listened to my complaints and cheered my successes, and for that I am extremely grateful.

This work was supported in part by the US National Science Foundation (NSF) awards CNS-1239543, CNS-1355778, and CNS-1446544.

Chapter 1

Introduction

John Knight wrote that "Safety-critical systems are those systems whose failure could result in loss of life, significant property damage, or damage to the environment" [8]. Safety-critical software engineering, then, is the practice of building software used in these systems. Building software correctly is challenging regardless of application, but it is especially difficult when the impact of a system failure is so high.

One domain where many systems are safety-critical is in medicine. Traditionally, medical devices have been built as standalone units, following similarly standalone safety-engineering processes. Increasingly, though, there is a desire for medical devices to work together—through some form of automated coordination—to perform various types of clinical procedures. These range from the straightforward, e.g., updating the electronic medical record automatically with the readings from a physiological monitor, to the advanced, e.g., automatically disabling the administration of an analgesic when a patient shows signs of respiratory distress.

Typically, the safety evaluations of these medical devices also treat them as monoliths: existing safety procedures (like hazard analyses) considered only completed systems. These procedures are not only de facto standards, known and used in industry, but are de jure standards as well: current evaluation processes for the United States Food and Drug Administration (FDA) only consider completed systems.

There are a number of concepts that have enabled more rapid construction (though not necessarily analysis) of software- and hardware-based systems, including:

- *Interoperability:* Systems are increasingly thought of not as standalone units but as components that must interoperate in order to achieve the system's overall goal. Such an approach also enables component *reuse*.

- *Reuse:* In response to rising demands on development resources, previously developed components are often reused in new systems. Some are even developed for a number of potential uses, rather than for one particular system.

- *Component-/Model-Driven Development:* Rather than have developers directly implement required functionality, system construction now often takes place via models of system components. In this paradigm, components can be thought of as building blocks that are specified and integrated before being automatically translated into system components.

- *Platforms:* Components need access to underlying services that, e.g., enable real-time communication and govern access to shared computational resources. These underlying services are collectively referred to as a platform; many modern development approaches first establish the platform to be used and then build system functionality around its capabilities and limitations.

- *Systems-of-Systems:* In many cases each component and the platform itself can be built, reasoned about, and sometimes used independently as standalone systems. Thus, what are referred to as "systems" are often actually hierarchical systems-of-systems that have interesting, recursive properties. Though some researchers object to the use of this term [9], we feel that it speaks to the concept of systems as a collection of cooperating elements that can all (to varying extents) also stand alone.

All of these concepts can combine to enable a number of beneficial, “marketplace”-like aspects for system development within medicine, but with this ease of development comes a corresponding increase in the difficulty of safety assurance. Safety assessment in a multi-vendor, component-based domain can be not only challenging but quite slow if performed using traditional techniques, which removes much of the benefit of these more modern development approaches.

Work in this area has, to some extent, addressed the challenges of safety-critical and distributed systems. Many of these techniques prioritize architectural specifications as a “single source of truth” upon which various annotations targeting particular “quality attributes” can be added [10, 11]. Once all or part of the system’s architecture is annotated, supported analyses (either manual or tool-supported) can be performed.

What is needed, then, is an examination and demonstration of the suitability of these software engineering techniques to the challenge of safely constructing and evaluating medical device integration software. This work describes the motivations and results of an effort to perform this task.

Specifically, the contributions of this work are:

1. *Theory – Hazard Environment Hierarchy:* I examine the relationship between a component and its environment in the context of system safety. I believe that a pattern exists in the vertical decomposition of systems which makes the component-environment relationship a hierarchically repetitive one, and this repetition can be exploited to increase the formality of existing hazard analyses. I propose a collection of definitions that, while by no means a full formalization or end-to-end theory, move the previously disparate topics of system safety analysis and compositional verification one step closer together.

2. *Process – Hazard Analysis:* Leveraging the realizations from 1, I propose significant modifications to an existing systems-based hazard analysis. These modifications result in a process that, I believe, requires less user-expertise and is more repeatable than the

status quo. Additionally, a subset of these modifications are designed to be integrated with a semi-formal description of a system's architecture, the increased rigor of which may bring additional analytic power and reduced analysis time.

3. *Tooling:*

   (a) *Code Generation:* I demonstrate existing approaches to architectural specification and automated system construction by developing a software prototype that will take a specification of a medical system as input and produce as output a runnable version of that system. This enables a very close alignment of system model and executable artifacts, saving developer time and enhancing the quality of model-based analyses.

   (b) *Report Generation:* Leveraging the code generation described in 3.(a), I will describe additional annotations that align with and support the hazard analysis process from 2, fully integrating it with a semiformal description of a system's architecture. The software prototype from 3.(a) will then be extended to support automated generation of a hazard analysis report usable by a number of system stakeholders.

Throughout this work we use as a running example the "PCA Interlock" clinical scenario, which is introduced in Section 2.1.5. This work is being done—in collaboration with engineers from UL and the United States Food and Drug Administration—in the context of an existing standard governing integrated clinical environments as well as an upcoming standard addressing medical device interoperability. [2]

Chapter 2

Literature Review

In this section, I examine the three areas that this work draws from. First, I review the literature surrounding the integration of medical devices, primarily in the context of *Medical Application Platforms*. Second, I review the relevant literature in the system safety domain, with particular attention paid to hazard analysis techniques and medical system safety standardization efforts. Third, I review the literature associated with system and software architecture modeling, focusing on formal architecture descriptions that enable automated system construction and analyses.

2.1 Integrating Medical Devices

2.1.1 Medical Application Platforms

Though the concept of integrating medical devices has been circulating for some time (e.g., [12, 2]), in [13], Hatcliff et al. introduce the term *Medical Application Platform* (MAP) and define it as "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 (i.e.*apps*) that provide medical utility via the ability to both acquire information from and update/control integrated devices, IT systems, and displays." They go on to explain the need for the concept, mentioning an aging population that is projected to increase demands for healthcare services and a focus on outcome-based medicine driving "quality enhancements and cost reductions."

Hatcliff et al. further explain that in other, non-safety-critical domains, there are three trends which are driving integration efforts: increasing levels of *device interoperability*, a *platform approach* (where the entire ecosphere of hardware and software components are governed by a single entity, e.g., Apple's governance of iOS), and the embrace of a *systems perspective*, which encourages the conception of components as (sub)systems, i.e., taking a systems-of-systems approach. This fits well with Checkland's definition of a system, which is "A set of elements connected together, which form a whole, this showing properties which are properties of the whole, rather than properties of its component parts." [14]

These platforms allow the dynamic integration of medical devices and purpose-built software in order to serve some clinical need. Hatcliff et al. explain that there are four missing elements, the absence of which block the implementation and industrial adoption of MAPs:

- *Interoperability Standards:* Coordinating the actions of disparate entities in a marketplace requires some form of incentives. Hatcliff et al. point out that "there has been significant activity on standards" targeting the exchange of information, but existing works "do not address the technical requirements for device connectivity, safety, and security."

- *Appropriate Architectures:* Architectures addressing the challenges faced by MAPs exist in other domains, such as Integrated Modular Avionics (IMA) for aviation, or Multiple Independent Levels of Security (MILS) in security-oriented fields [15, 16]. These are not, however, immediately applicable to the medical field.

- *Regulatory Pathway:* Regulatory agencies, like the US Food and Drug Administration, approve medical devices in a monolithic fashion. Approval of each permutation of a composable system like those enabled by MAP technology, though, would be an arduous task indeed: the number of possible combinations given only a small number of devices and applications would still be quite large.

- *Interoperability Ecosystem:* The phrase "interoperability ecosystem" (introduced in [13], renamed to *ecosphere* and greatly expanded in [17, 1]) refers to the complete environment surrounding a platform. King et al. define it as the "set of devices, software applications and computational platforms intended to interact with one [another]; the stakeholders that organize, manufacture, and use these products; as well as the explicitly defined processes that are followed to develop, certify, and use these products."

In addition to the removal of these roadblocks, Hatcliff et al. also identify a number of technologies necessary to enable the MAP vision. They group these technologies into four main areas:

1. *Architecture and Interfacing:* Interoperable architecture, compliance-checkable interfaces, rich interface description language, etc.

2. *Platform Technology:* Static verification, composability, global resource management, automated trust and security services, etc.

3. *Safety, Effectiveness, and Certification:* Component-wise regulation, third-party certification, etc.

4. *Ecosystem Support:* Consensus-driven component specifications, development environment for MAP apps, etc.

It is in the fourth area, specifically in the development environment, that this work makes the bulk of its contribution. There is a great deal of overlap between goal areas, though, so a common development environment would include support for a number of other goals as

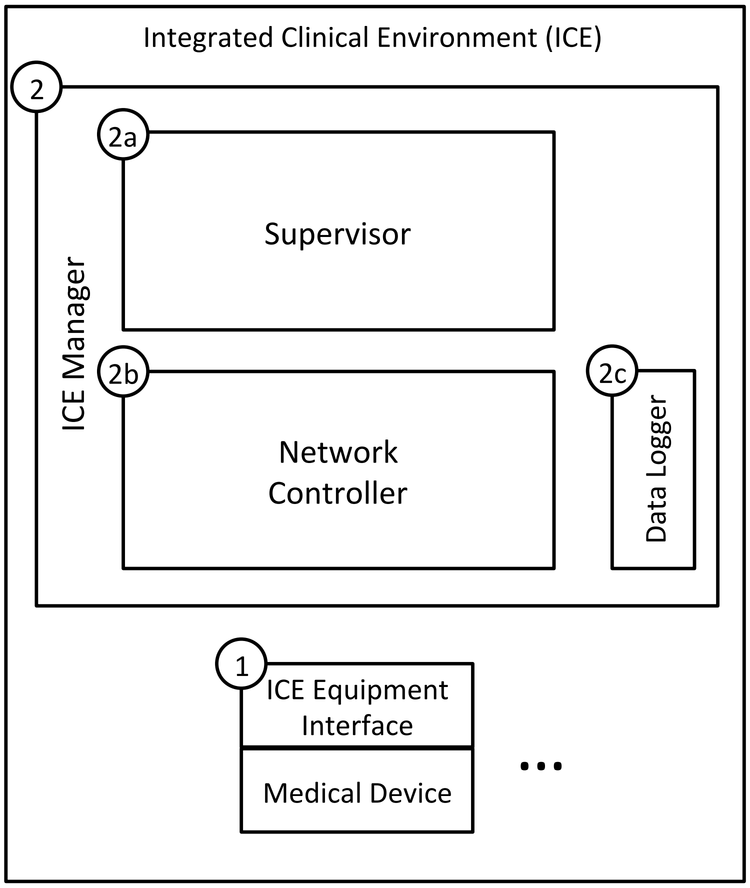


Figure 2.1: *The ICE Architecture, figure adapted from [1, 2]*

well by supporting, e.g., interface checking; static verification tooling; repeatable analysis processes for analysts, regulators and certifiers, etc.

2.1.2 The Integrated Clinical Environment

In 2009, ASTM International released the standard ASTM-F2761 "Medical Devices and Medical Systems – Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) – Part 1: General requirements and conceptual model" [2]. The standard describes the high-level functional (i.e., non-architectural/non-implementation) requirements for a possible MAP implementation. Figure 2.1 shows the ICE components (descriptions adapted from [2]):

1. *Equipment Interface:* Though no description is given of the ICE Equipment Interface (only examples), it is understood to enable a piece of equipment (e.g., a medical device) to communicate with the ICE system

2. *Manager:* The ICE Manager is the name used to refer to the collection of the ICE Supervisor, Network Controller, and Data Logger.

   (a) *Supervisor:* The ICE Supervisor is responsible for either ensuring that the requirements (both functional and non-functional) provided by the Network Controller can deliver the intended use of the ICE Supervisor, or for generating an alarm if the requirements of a running system can no longer be met.

   (b) *Network Controller:* The ICE Network Controller is responsible for ensuring the delivery of the functional capabilities, "in accordance with non-functional requirements" of the connected devices to the ICE Supervisor, or generating an alarm if the connection fails in some way.

   (c) *Data Logger:* The data logger provides a time-indexed recording of "the accessible 'state-of-the-clinical environment'" This enables any problems that are encountered to be diagnosed post-hoc, be they related to safety, security, or functional concerns.

Though there are a small number of different ICE implementations from both academic and commercial suppliers, this focuses on one in particular, the *Medical Device Coordination Framework*.

2.1.3 The Medical Device Coordination Framework

In their 2009 work, King et al. described the Medical Device Coordination Framework (MDCF), which is an open-source implementation of the ICE standard developed at Kansas State University and later the University of Pennsylvania [18]. The MDCF, and its associated tooling, has been under near-continuous development since.

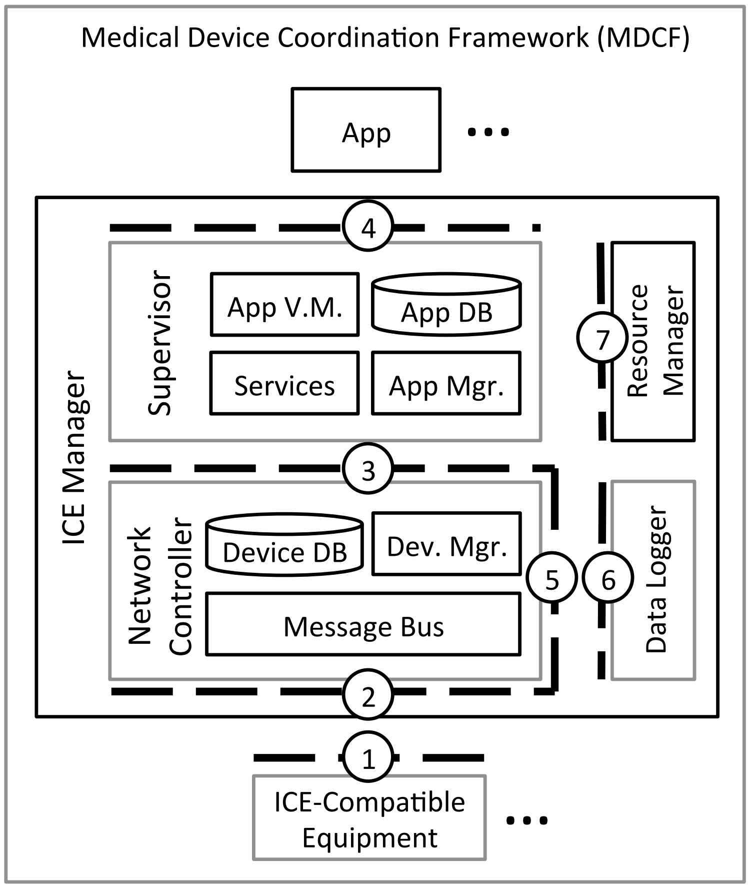


Figure 2.2: *The MDCF Architecture, figure adapted from [1]*

Kim et al. recently described how the MDCF implements the ICE Architecture [1][1]. Figure 2.2 shows a decomposition of the ICE components, the addition of separate architectural elements representing application logic and a global resource manager, and the seven interfaces over which the ICE components interact. MDCF-specific components are outlined in black, interfaces are thick-dashed lines, and ICE components are outlined in grey.

The MDCF specific component additions Kim et al. describe are:

1. *App:* Software applications that guide the integration of medical devices to do something clinically meaningful. Though these are executed in the Supervisor, they are developed and typically analyzed as architecturally independent. Note that the term

[1]An earlier description can be found in Hatcliff et al.'s ICCPS 2012 paper. [13]

used in the ICE standard is "Application logic," rather than *app*.

2. *Resource Manager:* Enforcing the quality-of-service specifications of both apps and devices requires global resource management involving the allocation and monitoring of networking and computing resources (e.g., channels and CPU time, respectively). The MDCF relies on the resource manager provided by King's MIDAS [19].

3. *Supervisor:*

   (a) *App Virtual Machine:* Apps are envisioned to be executed in isolated environments (e.g., separation kernels [16]). As of this writing, this feature is not implemented – apps are executed as written on the shared Java virtual machine.

   (b) *App Database:* Stores the names and configuration schemata of app implementations. Executable logic is looked up at launch time based on the app's name.

   (c) *App Manager:* Handles the instantiation/connection (including the verification of digital certificates), monitoring, and termination of instantiated applications.

   (d) *Services:* A collection of services that expose the functionality of the app manager (e.g., app launch) to the clinical user.

4. *Network Controller:*

   (a) *Device Database:* Stores identity information and capability descriptions of devices that have been approved to work with the MAP. This allows individual healthcare delivery organizations (HDOs, e.g., hospital) to restrict which devices are usable on their network.

   (b) *Device Manager:* Similar to the app manager, this module handles the authenticated connections of ICE compatible equipment.

   (c) *Message Bus:* The actual communication infrastructure which handles the routing of messages between components.

Kim et al. also specify seven interfaces (identified by circled numbers in Figure 2.2) that the components use to communicate with one another. We restate these interfaces here, and adopt the notation $A \rightarrow B$ which signifies that component A uses an interface provided by component B:

1. *Platform → Devices:* The platform uses this interface to creates instances of the device's supported communication exchanges. For example, if the device supports a requester/responder mode of interaction, this interface would allow its creation and binding to appropriate network channels.

2. *Devices → Platform:* Devices use this interface to connect to and authenticate with the platform.

3. *Supervisor → Devices via Network Controller:* The supervisor uses this interface to query available devices. An example of this usage would be when an app is requesting some set of device capabilities; the check to see if those capabilities are present would go through this interface.

4. *Apps → Supervisor:* This interface allows apps to request platform services, like the reservation of computing or networking resources.

5. *Data Logger → Network Controller:* This interface is used by the data logger to retrieve messages sent across the network from the network controller.

6. *Network Controller → Data Logger:* This interface is used by the network controller to get messages stored by the data logger for playback. This is used for, e.g., post-hoc/forensic analysis.

7. *Supervisor → Resource Manager:* This interface is used by the supervisor to request the allocation of computing or networking resources, typically on behalf of an application that a clinician would like to launch.

2.1.4 Connecting Medical Devices

When the term "device interconnectivity" is used in the medical domain, its precise definition is rarely made explicit; rather, it is typically understood to mean "interoperability." Work in the field of simulation theory has led to the proposal of different "types" or "levels" of interoperability, but most recognize at a minimum the distinction between two components being able to "interconnect" (e.g., use a similar communication protocol, data format, etc.) and to "interoperate" (e.g., shared understanding of assumptions and semantics).

Petty and Weisel (who focus on simulation systems, though we believe their definitions work in this domain as well) discuss *composability*, which they define as "the capability to select and assemble... components in various combinations into valid... systems." They further explain that there are two perspectives from which composability can be understood: the *engineering* perspective, which focuses on *syntactic* composability (i.e., "whether the components can be connected") and the *modeling* perspective, which focuses on *semantic* composability (i.e., whether the computation of the composed system is semantically valid") [20].

Syntactic Interoperability

The primary concern of syntactic interoperability is the connection between two devices, or between a single device and some middleware. Unfortunately efforts in this area have not yet converged onto a single technology or style. Early work, like the first versions of the MDCF, used the Java Messaging Service (JMS) [18], which is an API that describes a peer-to-peer messaging service; JMS has both open-source and hardened commercial implementations [21]. Modern implementations of the MDCF can be configured to use either a purpose-built message-bus provided by MIDAS or the Data Distribution Service (DDS) [19, 22]. Some commercial MAP implementations (e.g., Docbox) use DDS, while others (e.g., Dräger) use a purpose-built web-service framework [23].

One of the early developments of research in this area was the recognition that MAP apps were in general well suited to publish-subscribe (pub-sub) architectures. Pub-sub works well because a number of MAP apps may be operating on the same patient at the same time, so allowing multiple apps to efficiently read from the same physiological monitoring device efficiently was a high priority. See, for example, [18] for a discussion of the performance aspects of a JMS implementation of the MDCF in various fan-in and fan-out topologies using a range of message sizes and types. Typical MAP implementations use an explicit invocation style of pub-sub, in that the underlying networking middleware routes messages to particular components, but those components themselves are responsible for implementing handlers for event or message arrival. Another advantage of pub-sub is the decoupling of data producers from consumers [11] (all but a necessity in a compositional plug-and-play system like MAPs), though competing concerns like data privacy and security can, at times, require some level of coupling.

While pub-sub works well for reading physiological data from a number of sensors, it is less straightforward to use in more advanced MAP apps that require device control. Since it is unnecessary (or possibly even inappropriate) to have commands sent from a logic module to a specific device broadcast widely, point-to-point communication is, in these cases, preferable to pub-sub. While it is possible to have per-device topics (a strategy employed by, e.g., the MDCF); one advantage of Dräger's web-service architecture is the explicit inclusion of point-to-point (discussed in their work as request-response) style connections for private communications and device control [23].

Semantic Interoperability

The issue of semantic interoperability, or the challenge of ensuring equivalency between similar values and concepts across disparate components (e.g., medical devices, app logic modules, etc.), is one best addressed by domain (i.e., medical) experts. To that end, we have looked to relevant medical standards—in particular, IEEE 11073[2]—which has two goals: to

[2]Note that while IEEE 11073 contains specifications for both semantic and syntactic interoperability, syntactic interoperability is quite well treated by software engineering literature. We use the standard

“provide real-time plug-and-play interoperability for patient-connected medical devices” and to “facilitate the efficient exchange of vital signs and medical device data, acquired at the point-of-care, in all health care environments.” [24]

At a high level, the standard focuses on two topics: nomenclature and the “domain information model” (DIM). As the name implies, the nomenclature sections of the standard give a semantic coding for a range of physiological categorizations (e.g., alerts, body sites, metrics, etc.). The DIM, on the other hand, “is an object-oriented model that consists of objects, their attributes, and their methods, which are abstractions of real-world entities in the domain of... medical devices.” [25]. The full family of standards includes refinements for specific device families (e.g., pulse oximeters) and is broadly split into those devices used for personal health and those used at the point of care.

While our examples draw inspiration from—and where possible align with—IEEE 11073, we do not use it outright for two main reasons. First, device interconnectivity is only a part of the full MAP vision, and IEEE 11073 does not include facilities for, e.g., hosting/execution of application logic. Second, and more practically, the standard follows an object-oriented, dynamic discovery style of capability description, which both prevents static knowledge (and thus static analysis) of a system and makes makes the creation of robust application logic more complex. That is, applications will have to be responsible for implementing a potentially large number of code paths, depending on the result of the various dynamic capability queries.

2.1.5 A PCA Interlock App

One of the most well-studied MAP apps is the *PCA interlock scenario* [26, 27, 28]. The app deals with an unfortunately commonplace problem in clinical settings, where following surgery or major trauma, a patient is given a patient-controlled analgesia (PCA) pump to manage her pain. The pump, which may or may not administer a constant, low rate

primarily as a source of knowledge of domain-specific semantic aspects.

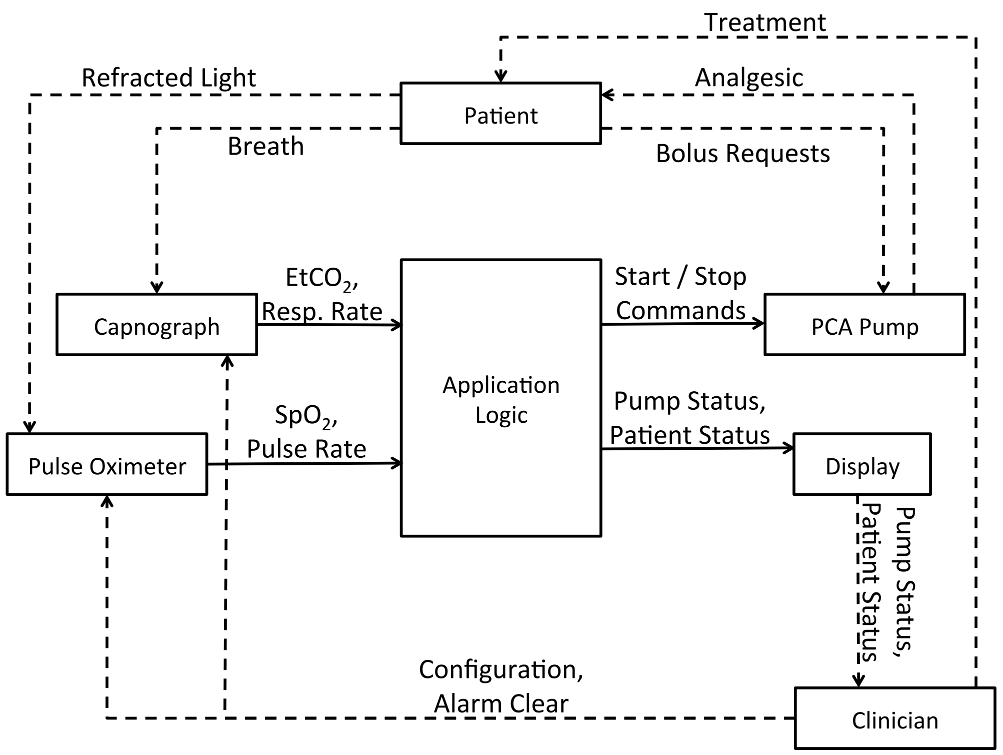


Figure 2.3: *The App Developer's view of the PCA Interlock Application*

(sometimes referred to as the "basal" rate) allows a the patient to press a button (sometimes referred to as a "bolus trigger") to receive a larger dose of analgesic. This interaction is designed to be self-limiting as patients will typically fall unconscious and be unable to self-administer more analgesic. Various problems exist, however, which can lead to accidental overdoses, e.g., atypical patient pharmacokinetics; a visitor pressing the button for the patient (so-called "PCA-by-proxy"); the patient rolling over onto the trigger while asleep; or incorrect medication, dosage, or pump settings by a clinician [29]. Regardless of cause, an overdose can lead to respiratory depression which can cause serious injury or death.

The PCA interlock scenario involves the coordination of one or (typically) several patient monitoring devices, e.g., a capnograph and pulse-oximeter. The app developer's view of one possible configuration is shown in Figure 2.3. This figure shows the primary elements of the app (medical devices, application logic, etc.) and their connections; note that the solid arrows represent connections that are explicitly created as part of the app's instantiation (i.e., network connections) and the dashed arrows represent communication supported, but not directly enabled, by the app. A number of physiological parameters are monitored (e.g., SpO_2, $ETCO_2$, and respiratory rate) by application logic for signs of respiratory depression according to some pharmacokinetic model. This model, which is embedded in the app's logic, can be configured either automatically or by the clinician according to patient history, medical conditions, and treatment status (e.g., the presence of supplemental oxygen). When the model indicates signs of respiratory depression, the PCA pump can be disabled, via a `stop` command. More sophisticated implementations, where the pump is only enabled for certain windows of time (which precludes situations where the PCA pump is left enabled inappropriately due to network failure) have also been proposed [27]. Note that the app-developer-centric view of Figure 2.3 excludes some MDCF features that are transparent to the developer, e.g., the data logger or the fact that the physiological parameters may come from separate devices. A different view of the application—one made to align with the ICE/MDCF-centric view—is shown in Figure 2.4.

2.2 System Safety

2.2.1 A Note on Terminology

The vocabulary used when discussing safety, unfortunately, varies somewhat between sources. Leveson, who developed the hazard analysis technique this work hews most closely to, uses a set of the standard terms (accident, hazard, etc.) and gives her own definitions [30]. Ericson, who produced a handbook covering a number of techniques uses similar terms, but gives his own set of definitions [7]. Avižienis et al. give a full taxonomy of terms in [3], the definitions of which in general correspond Leveson and Ericson's use, though they do not explicitly cite his work. Many standards, e.g., ISO 14971, IEC 80001, and AADL's EMV2 Annex define their own terms (sourced either from other standards or the standardization

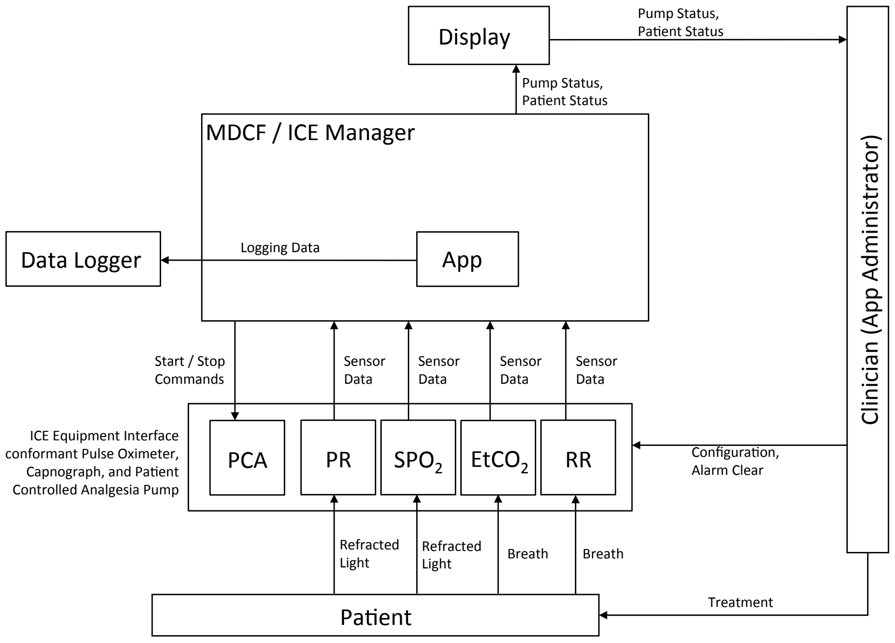

Display
Pump Status, Patient Status
Pump Status, Patient Status
MDCF / ICE Manager
Data Logger
Logging Data
App
Start / Stop Commands
Sensor Data
Sensor Data
Sensor Data
Sensor Data
ICE Equipment Interface conformant Pulse Oximeter, Capnograph, and Patient Controlled Analgesia Pump
PCA
PR
SPO2
EtCO2
RR
Configuration, Alarm Clear
Clinician (App Administrator)
Refracted Light
Refracted Light
Breath
Breath
Patient
Treatment

Figure 2.4: *The MDCF view of the PCA Interlock Application*

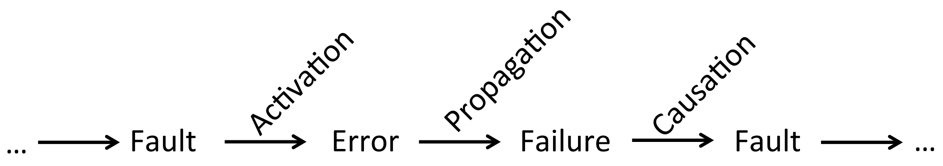


Figure 2.5: *The relationship between the terms "Fault," "Error," and "Failure," reproduced from Figure 11 of [3]*

body's style guide) [31, 32, 33]. The end result of this is a considerable difficulty in communicating precisely about safety as terms may have subtly different meanings to different readers.

Though a full reconciliation of the various terminologies is beyond the scope of this work we recognize the challenges posed and, unless otherwise noted, adopt the terminology and definitions used by Leveson. We adopt the specific terminology (found in [33] and shown graphically in Figure 2.5) that *errors* are "the difference between a computed, observed, or measured value or condition and the true, specified, or theoretically correct value or condition," *faults* are root causes of errors, and *failures* are the "termination of the ability of a product to perform a required function." Put another way, faults are the *root causes* of errors and failures are the *observable effects* of an error.

2.2.2 Hazard Analyses

For as long as systems have been built, some consideration has been paid to their safety. It wasn't until the middle of the twentieth century, though, that these attempts began to be formalized into more rigorous processes. We discuss three such processes here, the first two, Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA) as they are predominant in industry (using [7] as our main source); the third, System Theoretic Process Analysis (STPA) is a newer and more modern approach (detailed in [30]) that forms the basis of much of this work.

Fault Tree Analysis

Fault Tree Analysis, commonly referred to by the initialism FTA, was created by H. Watson and Allison B. Mearns at Bell Labs when they were working on the Minuteman Guidance System [7]. It is a top-down analysis technique that asks the analyst to first select an undesirable event and then consider all the contributory errors that could combine in various ways (typically via AND/OR gates) to cause that event. This process is then repeated recursively until an acceptable level of specificity is met.

An example fault tree for the PCA interlock scenario is shown in Figure 2.6. The top event corresponds to the main accident (or mishap, in the language of [7]) that the PCA interlock app is designed to avoid—an overdose of analgesic being administered to the patient. The figure shows that this could come about if two errors occur simultaneously: the application logic receiving bad physiological data *and* the error that caused the bad physiological data being undetected. The block that joins these two errors—labelled G1—

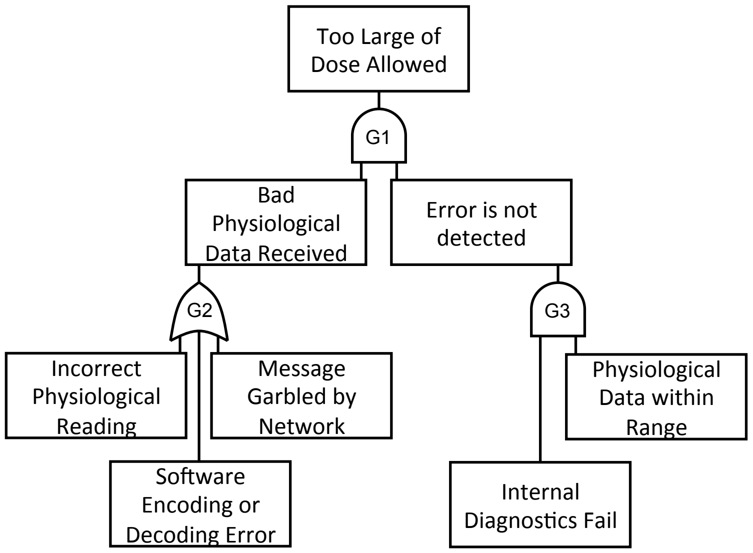

Too Large of Dose Allowed
G1
Bad Physiological Data Received
Error is not detected
G2
G3
Incorrect Physiological Reading
Message Garbled by Network
Physiological Data within Range
Software Encoding or Decoding Error
Internal Diagnostics Fail

Figure 2.6: *An example FTA for the PCA interlock scenario, adapted from [4]*

dictates the style of this join. Block G2, on the other hand, is an *or* join, so any of its children (incorrect physiological reading, software encoding or decoding error, or message garbling by the network) could independently cause the parent error: receipt of bad physiological data by the application logic. Figure 2.6 is, of course, only a fragment of a full FTA analysis, which would consider not only more top-level accidents (e.g., underinfusion causing the patient to the be in unnecessary pain, damage to the medical devices caused by inappropriate commands, etc.) and also a deeper analysis of the causes of overinfusion as well.

As FTA has considerable industry support and a history of use, Ericson details extensive theory beyond the basic example given above [7]. Fault tree analyses can be annotated with the likelihoods of particular errors; various statistical methods (Fussell-Vesely importance, Risk Reduction Worth, Birnbaum's importance measure, etc.) then detail how these likelihoods might be combined to arrive at overall probabilities of failure or success. There are also various algorithms for determining subsets of the full fault tree—so-called *cut sets*—which allow an analyst to focus on particular event chains. Additionally, while the simple graphical notation used in Figure 2.6 forms the core of all fault tree analyses, it has been extended with a number of constructs to allow things like prioritization between gates, exclusivity, inhibition, m-of-n combinations, phasing, timing, dynamism, or even user-specified extensions.

Failure Mode and Effects Analysis

Failure Mode and Effects Analysis is commonly referred to via the initialism FMEA. If the analysis is annotated with criticality information, the technique is sometimes referred to as Failure Mode, Effects, and Criticality Analysis, or FMECA. The technique, which was detailed in MIL-STD-1629A, was developed in the late 1940's by the United States military [7]. It is a bottom-up analysis technique that asks the analyst to examine the various elements of his system (i.e., hardware components, functional components, software components, etc.) and consider how the element's failure would impact the behavior of the overall system. The set of hardware failures might include terms like "breakage" or "collapse," while the set of software failures might include terms like "deadlock" or "garbled message." This process would then be repeated for each other element in the system.

An example FMECA, following a worksheet provided by Ericson in [7] for the PCA interlock scenario is shown in Table 2.1. This example takes a functional approach (i.e., failure of system *functions* rather than hardware or software components are considered): the function examined is the provision of SpO_2 physiological data to the app. The first row considers the results of this function failing outright, while the second and third consider late delivery or incorrect value delivery. The third column asks for a failure rate, which is unknown for this function but may in some cases be available (typically those involving hardware failures). Causal factors are listed, as are immediate (or local) effects, and then system effects are considered separately. After the method of detection, there is sometimes a column for "current controls" though in this example we skip to the hazard, which should be identified and explained elsewhere. Though the entire worksheet will likely be customized by an organization or standards committee, the "Risk" number is especially likely to be tailored to the domain's needs. As a general-purpose index, Ericson mentions that the risk index

Failure Mode and Effects Analysis

System: PCA Interlock Scenario				Subsystem: Pulse Oximeter Device				Mode/Phase: Execution	
Function	Failure Mode	Fail Rate	Causal Factors	Immediate Effect	System Effect	Method of Detection	Hazard	Risk	Recommended Action
Provide SpO_2	Fails to provide	N/A	Network failure, device failure	SpO_2 not reported	Unknown patient state	App Logic	Potential for overdose	3D	Add safety timeout
	Provides late	N/A	Network congestion, transient device failure	SpO_2 not reported	Unknown patient state	App Logic	Potential for overdose	3C	Add safety timeout
	Provides wrong	N/A	Device error	SpO_2 value incorrect	Incorrect patient state	None	Potential for overdose	1E	Have device report data quality with sensor reading
Analyst: Sam Procter				Date: September 26, 2014					Page: 3/14

Table 2.1: *An example FMEA Worksheet for the PCA Interlock scenario, adapted from [4, 7]*

from MIL-STD-882 is “generally followed.” The first character is a severity rating, which ranges from 1 (Catastrophic, resulting “in one or more of the following: death, permanent total disability, irreversible significant environmental impact, or monetary loss equal to or exceeding $10,000,000”) to 4 (Negligible, resulting “one or more of the following: injury or occupational illness not resulting in a lost work day, minimal environmental impact, or monetary loss less than $100,000”). The second character is a probability level, ranging from A (Frequent, “Likely to occur often in the life of an item”) to E (Improbable, “So unlikely, it can be assumed occurrence may not be experienced in the life of an item). Probability level F is also sometimes used to signify accidents that have been completely eliminated from possibility. The last column of the worksheet is the recommended action, which specifies what is to be done to eliminate or mitigate the failure.

System Theoretic Process Analysis

After examining a number of failings with safety-critical systems (and the engineering processes that guided their designs) Leveson recently described the *Systems Theoretic Accident Model and Processes* (STAMP) causality model and an associated hazard analysis, *System Theoretic Process Analysis* (STPA) [30]. As their names imply, STAMP and STPA differ most significantly from previous hazard analyses in their use of systems theory, which is defined by Leveson as an approach that “focuses on systems taken as a whole, not on the parts taken separately.” This integration leads to the key realization that safety is an emergent property that can be viewed as a control problem. Put another way, unsafe events are the result of inadequate control.

Unlike FMEA, which has the failure of certain system components as its central notion, or FTA, which focuses on avoidance of certain events, STPA is driven by the enforcement of *safety constraints*, which are rules that, if properly enforced, prevent inadequate control of the system.

Once a list of potentially unsafe control actions has been identified, causes can be an-

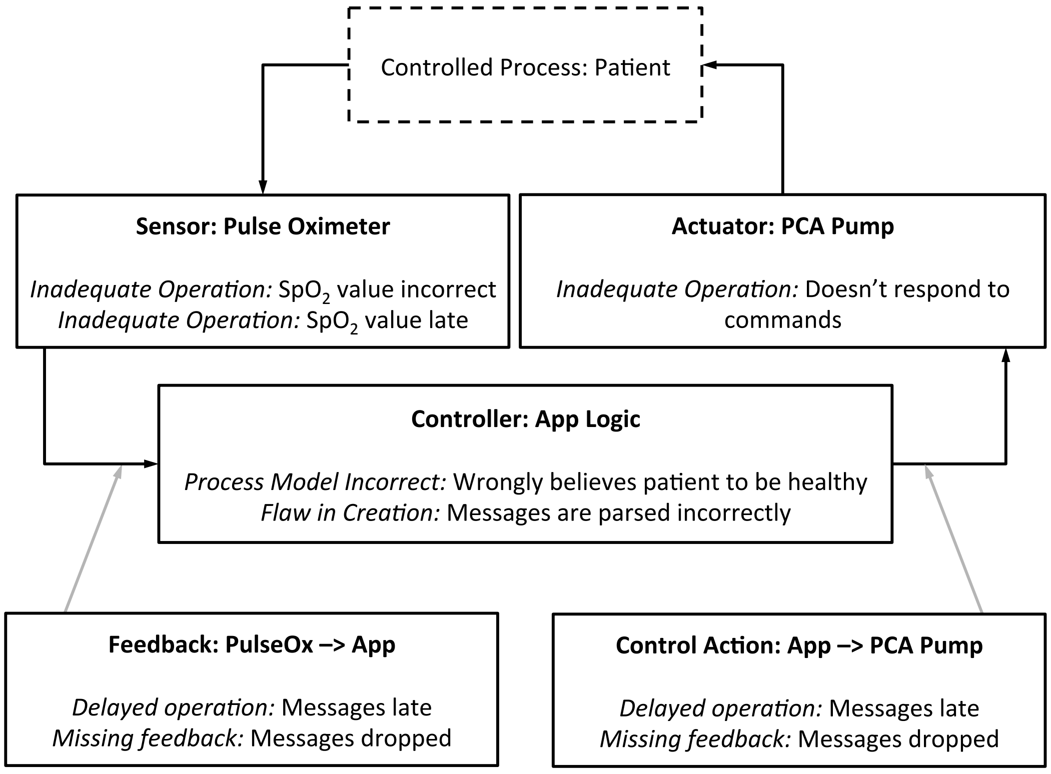

Controlled Process: Patient
Sensor: Pulse Oximeter
Inadequate Operation: SpO$_2$ value incorrect
Inadequate Operation: SpO$_2$ value late
Actuator: PCA Pump
Inadequate Operation: Doesn't respond to commands
Controller: App Logic
Process Model Incorrect: Wrongly believes patient to be healthy
Flaw in Creation: Messages are parsed incorrectly
Feedback: PulseOx –> App
Delayed operation: Messages late
Missing feedback: Messages dropped
Control Action: App –> PCA Pump
Delayed operation: Messages late
Missing feedback: Messages dropped

Figure 2.7: *A control loop from the PCA Interlock example, annotated according to STPA. Adapted from [4]*

alyzed and prevented or mitigated. STPA contextualizes control actions by placing them into a hierarchical control structure. The most canonical control structure is the *control loop*, an example of which is shown in Figure 2.7. Elements of this control loop are labelled as "Component Role: Component Name" (e.g., the pulse oximeter plays the role of a "sensor" in this control loop). The lower half of the components are then labelled with ways that they can contribute to the system being unsafe, in the format "*Causality Guideword:* Error." Procter and Hatcliff describe an early version of the work in this dissertation as (among other goals) an attempt to develop a canonical report format for STPA [4].

2.2.3 The Fault Propagation and Transformation Calculus

Each of the previously designed hazard analyses have as their goal to find the causes of accidents so that they can be eliminated (by a more thoughtful system design) or mitigated by some control. FMEA does this perhaps most explicitly by starting from causes and working up to effects, FTA's cut sets are essentially top-down projections over a range of possible causes to find a causal chain, and STPA's second step focuses on finding "causal scenarios" which might allow safety constraints to be violated.

It should be noted that Leveson explicitly calls out the assumption that "Accidents are caused by chains of directly related events" and instead offers the replacement "Accidents are complex processes involving the entire socio-technical system. Traditional event-chain models cannot describe this process adequately." [30] Masys presents an expanded version of this argument in his dissertation, writing that "...linear thinking is a myopic perspective that does not recognize the multiple interrelations and entanglement that characterizes the [problem] space and therefore is not an effective mode for understanding [the] complex socio-technical domain." [34] We argue that, though explicitly rejecting the idea of an event-chain, by asking the analyst to identify "causal scenarios" in which safety-constraints are violated Leveson retains the idea and importance of linear causality in STPA.

Wallace has formalized this notion of (linear) causality by introducing the Fault Propagation and Transformation Calculus, or FPTC [35]. He gives a full syntax and evaluation semantics for the calculus where a directed graph, made up of nodes corresponding to components and communication links, is seeded with behavior tokens (i.e., a combination of "correct" behaviors and "faults"). Then, the tokens propagate and transform through the various components (a process which is fully specified as a fixpoint algorithm) repeatedly until the graph stabilizes. Note that Wallace uses the term fault to include both root causes and resultant errors as, in his calculus, they are to some extent interchangeable.

Nodes in the system's graph may produce new behavior tokens (such a component would be a *source* of the behavior), representing some independent action. For example, if the pulse oximeter in our example could produce incorrect values, this would be represented in the FPTC by having the PCA pump component be a source for a token labelled “SpO_2 Wrong.” Nodes may also consume tokens (i.e., the component is a *sink*), representing its ability to compensate for some failure. Consider a sophisticated app logic implementation that cross-checked the SpO_2 values with $ETCO_2$ and respiratory rate values—such an implementation would consume the SpO_2 Wrong faults and behave normally (as long as the other parameters were correct). Nodes may also propagate tokens (representing, e.g., the propagation of incorrect values) or transform them. For example, the app logic might enable the pump if it does not get any inputs from the sensors, this would be represented in the FPTC as a transformation from a “SpO_2 Late” token into an “Inadvertent Pump Enable” token. We provide an in-depth example of FPTC in Section 5.5.1.

2.2.4 Documenting Safety: Assurance Cases

Once a hazard analysis has been performed, the resulting artifacts (e.g., worksheet, report, etc.) can be sent to various stakeholders for evaluation. Often, though, a hazard analysis is situated as part of a larger argument of the overall safety of a system, and though this argument can take various formats, arguments are typically presented in a structured format called an *assurance case*. Assurance cases typically contain, at a minimum, claims, arguments, and evidence, though Rushby explains in [36] that “standards and guidelines on which some certification processes are based often specify only the evidence to be produced; the claims and argument are implicit, but presumably informed the deliberations that produced the guidelines.”

Assurance case notation, construction and evaluation are themselves well-studied, large subjects. While we will not go into great detail here, the interested reader is directed to [5] which is an excellent primer on the state-of-the-art.

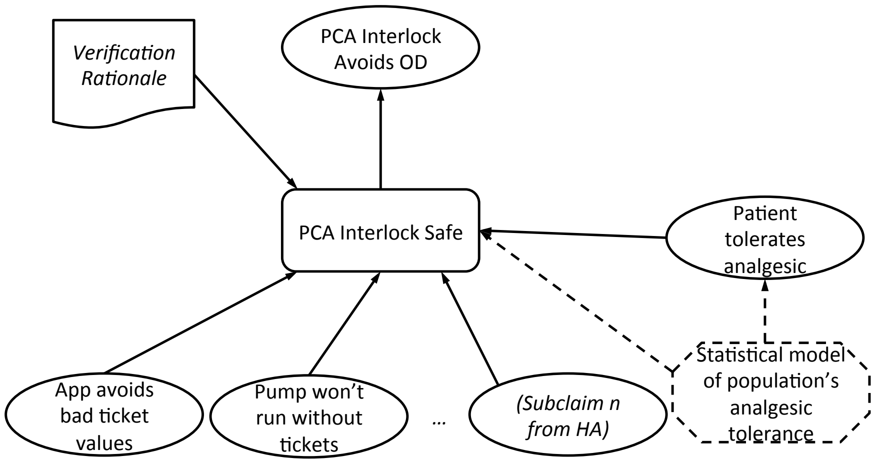

Verification Rationale
PCA Interlock Avoids OD
PCA Interlock Safe
Patient tolerates analgesic
Statistical model of population's analgesic tolerance
App avoids bad ticket values
Pump won't run without tickets
...
(Subclaim n from HA)

Figure 2.8: *An example of the Claims-Argument-Evidence assurance case format's graphical notation that is equivalent to Figure 2.9. Adapted from [5, pg. 56]*

```
ARGUMENT PCA Interlock Avoids Overdose
CLAIM
 The PCA Interlock is safe
ASSUMPTIONS
 The patient has a standard tolerance for analgesic
PREMISES
 App avoids bad ticket values,
 PCA pump won't run without tickets,
 ...
 Subclaim n from Hazard Analysis
JUSTIFICATION
 Verification Rationale
CONFIDENCE
 Statistical models have been used in the app's construction
END ARGUMENT
```

Figure 2.9: *An example of the Claims-Argument-Evidence assurance case format's textual notation that is equivalent to Figure 2.8. Adapted from [5, pg. 56]*

Notations

Rushby explains that there are two primary notations for assurance cases, with some degree of tool support for both the notations themselves and converting between the two [5].

Claims-Argument-Evidence (CAE) This notation, developed by Adelard, has both textual and graphical representations. An example from Rushby [5] that has been adapted for the PCA Interlock is shown graphically in Figure 2.8 and textually in Figure 2.9. The figures show some of the more common elements of assurance cases (claim, justification, etc.) as well as less common ones, like assumptions (or "side-warrants,") and statements of confidence (or "backing,") which Rushby explains are derived from Toulmin's argument structure [37]. CAE has tool support from Adelard itself in the form of the Assurance and Safety Case Environment (ASCE) software.

Goal Structuring Notation (GSN) A second popular notation was introduced by Kelly and Weaver in [38] and then explained in great detail in Kelly's doctoral dissertation [39]. A great example of this notation can be found in Figure 2 of Feng et al.'s fragment of a safety case for the PCA interlock scenario [40]. The primary elements here are goals (squares), strategies (parallelograms), contexts (stadiums), and solutions (circles). Helpfully, goals can be affixed with diamonds to denote that they are "undeveloped" and still need to be completed.

ISO/IEC 15026 In addition to academic sources, there is regulatory guidance on assurance cases as well in the form of ISO/IEC 15026, which is titled *Systems and Software Engineering – Systems and Software Assurance*. The standard has four parts:

1. *Concepts and Vocabulary:* This part clarifies the meanings of the terms as they are used in the standard.
2. *[The] Assurance Case:* This explains the structure and content of an assurance case. Note that unlike CAE and GSN, this standard uses a textual format.
3. *System Integrity Levels:* Since different systems have different levels of criticality, they also should be evaluated to different levels of integrity. This part discusses the definition and use of levels of assurance integrity.

4. *Assurance in the Life Cycle:* This final part explains how assurance cases can be used throughout a product's life cycle by aligning with the normatively-referenced IEEE 15288 and IEC 12207 (System and Software Life Cycle Processes, respectively) [41, 42].

Evaluation

Evaluating an assurance case is no simple task. Not only is the system being evaluated large and complex except in all but the most trivial of cases, but also, as Rushby points out, an evaluator is also faced with evaluating the case itself [5]. That is, a great system may have a bad assurance case, or a bad system may have an excellent assurance case.

Worse still, many of the arguments in an assurance case are by nature subjective, so in addition to evaluating soundness, an evaluator is also tasked with evaluating confidence. Deciding how to explicitly include this confidence evaluation is an ongoing challenge, and initial attempts to model it statistically resulted in less-than-practicable solutions. An alternative solution coming primarily from researchers at City University in London, results from the admission of the possibility of a system's perfection and it has yielded more promising results—see for example [43, 44, 45].

2.2.5 Standardization Efforts

While the academic literature contains a great deal of knowledge regarding system safety, there is also a substantial amount of information, with which much of industry aligns, in various standards. Standards specific to particular portions of this effort are discussed throughout this chapter (i.e., IEEE 11073 in Section 2.1.4 or AS 5506 in Section 2.3.3), but we discuss and summarize a handful of relevant standards here.

General System Safety Standards

Though there are a number of standards specific to the (software-driven) medical device domain, there are two more general standards we discuss first in order to orient the reader to the area and align with non-medical safety efforts.

IEC 61508 Perhaps the most relevant general system safety standard is IEC 61508, titled *Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems* [46]. The seven-part standard discusses safety requirements for electrical, electronic, and/or programmable electronic (E/E/PE) systems throughout their lifecycle. The standard is divided into seven parts. Those parts provide both requirements (e.g., general requirements (part 1), requirements for E/E/PE safety-related systems (part 2), software requirements (part 3)) and guidance on meeting those requirements (e.g., examples of methods for the determination of safety integrity levels (part 5). It also provides standardized definitions and abbreviations (part 4), guidance on applying parts 2 and 3 of the standard (part 6), and an overview of techniques and measures (part 7)).

ARP 4761 A second notable standard is ARP 4761, titled *Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment* [47]. Though this standard applies to the construction of aircraft rather than generic systems or medical devices, it is valuable because it has been analyzed and referenced by a number of important authors in the safety literature, e.g., Leveson and Rushby [48, 5]. The process consists of three main analyses (explained in detail in the standard's first three appendices) which are designed to roughly correspond to an aircraft's primary development phases:

1. *Functional Hazard Assessment (FHA):* This analysis, which aligns with the requirements phase of an aircraft's development, should "identify and classify the failure condition(s) associated with the aircraft functions and combinations of aircraft functions." It "is a high level, qualitative assessment of the basic aircraft functions as defined at the beginning of aircraft development."

2. *Preliminary System Safety Assessment (PSSA):* This analysis uses as input the output of the FHA, and "is a systematic examination of the proposed system architecture(s)

to determine how failures can cause the functional hazards identified by the FHA." It corresponds to the design phase of aircraft development.

3. *System Safety Assessment (SSA):* This analysis uses as input the output of the PSSA, and "is a systematic, comprehensive evaluation of the implemented system to show that the safety objectives from the FHA and derived safety requirements from the PSSA are met." It corresponds to the test phase of aircraft development.

While ARP 4761 has so far been used with some level of success, Leveson et al. explain in [48] that "In the reality of increasing aircraft complexity and software control, we believe the traditional safety assessment process used in ARP 4761 omits important causes of aircraft accidents." The authors argue that STPA (which is a product of their laboratory) or other approaches are necessary to remedy this omission.

Medical System Safety Standards

Much as ARP4761 applies specifically to the construction of aircraft, so too are there medical device safety standards, which we consider here. These standards typically apply to device vendors and, at times, healthcare delivery organizations as well. They are created by domain experts and published by standards organizations like the International Organization for Standardization (ISO), International Electrotechnical Commission (IEC), American National Standards Institute (ANSI), and the Association for the Advancement of Medical Instrumentation (AAMI). Adherence is evaluated by certification authorities like UL (formerly known as Underwriters Laboratories), and required by regulatory authorities like (in the United States) the Food and Drug Administration (FDA). These organizations derive their authority from legislative or executive actions. The entire "ecosphere" of stakeholders envisioned for the ICE project is more fully explained by Kim et al. in [1]. Here, we summarize four particularly relevant standards.

IEC 60601 As medical devices have been electromechanical in nature for some time, so too have there been standardization efforts in the area for several decades. The first edition of one of the oldest standards, ANSI/AAMI/IEC 60601: *Safety and Effectiveness of Medical Electrical Equipment* was introduced in 1977 [49]. The standard's scope "applies to the BASIC SAFETY [3] and ESSENTIAL PERFORMANCE of MEDICAL ELECTRICAL EQUIPMENT and MEDICAL ELECTRICAL SYSTEMS..." Like most of the standards discussed in this section, IEC 60601 is technically a family of standards, rather than a single monolithic standard. The family is divided into three general areas:

1. *60601: General Requirements:* The aim of this standard is to "specify general requirements and to serve as the basis for particular standards."
2. *60601-1-XX: Collateral Standards:* These apply to a family of medical electrical equipment/systems. That is, these standards "specify general requirements for BASIC SAFETY and ESSENTIAL PERFORMANCE applicable to:
   - A subgroup of [MEDICAL ELECTRICAL] EQUIPMENT (e.g., radiological equipment)
   - A specific characteristic of all [MEDICAL ELECTRICAL] EQUIPMENT not fully addressed in [the general requirements]."
3. *60601-2-XX: Particular Standards:* These apply to a specific type of device (e.g., cardiac defibrillators), and they "may modify, replace or delete requirements contained in [the general requirements] as appropriate for the particular [MEDICAL ELECTRICAL] EQUIPMENT under consideration..."

ISO 14971 A second relevant standard, is ANSI/AAMI/ISO 14971: *Medical Devices—Application of Risk Management to Medical Devices* [31]. The authors specify that its scope

[3]Terms in SMALL CAPITAL letters have specific definitions as part of a number of standards in this section. This convention is part of the standard; the quotes containing this notation are verbatim.

is to “[specify] a process for a manufacturer to identify the hazards associated with medical devices. . . to estimate and evaluate the associated risks, to control these risks, and to monitor the effectiveness of the controls.”

While the standard’s definition of terms is very useful (though see Section 2.2.1 for a caveat), its main contribution is not a process, as the scope might imply, but rather requirements for whatever process is used. These requirements are divided into four top-level steps, which reference a number of informative annexes:

1. *Risk Analysis:* Performing any of a number of processes could meet the requirements of the standard. Annex G lists several options: Preliminary Hazard Analysis (PHA), Fault Tree Analysis (FTA), Failure Modes and Effects Analysis (FMEA), Hazard and Operability Study (HAZOP), and Hazard Analysis and Critical Control Point (HACCP). We believe that the process described Chapter 4 of this dissertation could be used as well. Three important components of risk analysis, that are generic to the process used, are:

   (a) *Intended Use:* The notion of intended use is a particularly important one in medical device safety analysis. Section C.2.1 of the standard poses questions that may help identify the intended use of a device, i.e., “[W]hat is the medical device’s role relative to diagnosis. . . of disease?” or “What are the indications for use?”

   (b) *Identification of Hazards:* The identification of hazards, as opposed to how hazards could occur, is the focus of some processes (i.e., those types classified as “Conceptual (or Preliminary) Design Hazard Analysis Types” by [7], e.g., PHA).

   (c) *Estimation of Risks:* This involves estimating both the likelihood of the hazard occurring as well as the severity if it does occur. Section 3 of Annex D provides excellent guidance.

2. *Risk Evaluation:* Here the analyst is required to determine if a risk is great enough to require some sort of compensation.

3. *Risk Control:* This set of requirements governs the compensation for a given hazard, if required, and entails:

   (a) *Option Analysis:* Here the analyst simply identifies the possible compensatory actions.

   (b) *Implementation of Control Measures:* One or more of the options is then implemented.

   (c) *Residual Risk Evaluation:* The new design, with the compensatory actions in place, is then re-evaluated as in step 1(c). Annex J provides additional guidance.

   (d) *Risk/Benefit Analysis:* If the compensatory action(s) cannot completely remove the risk of the hazard, the analyst is then guided in weighing the residual risk against the benefits of the device.

   (e) *Risks Arising from Control Measures:* As the compensatory actions may themselves be unsafe, the analyst is instructed to evaluate risks resulting from their use.

   (f) *Completeness of Risk Control:* The analyst is required to "ensure that the risk(s) from all identified hazardous situations have been considered."

4. *Evaluation of Overall Risk Acceptability:* Finally, the overall risk of the device is evaluated. Section 7 of Annex D provides more guidance here, including identifying some candidate analyses (i.e., Fault and/or Event Tree Analysis).

The standard then discusses the need for a report which collects the above information. The report will also grow to include production or post-production information, as appropriate.

IEC 80001 A third relevant standard is ANSI/AAMI/IEC 80001: *Application of Risk Management for IT Networks Incorporating Medical Devices* [32]. The standard has two primary sections: roles and responsibilities, and risk management throughout the life cycle of a component of a medical IT network. Additionally, as in ISO 14971, the standard's definitions are particularly useful, perhaps none moreso than the three KEY PROPERTIES[4] of SAFETY, EFFECTIVENESS, and DATA AND SYSTEMS SECURITY. Note that the risk management processes in this standard apply to medical devices after they have been acquired by a purchaser (i.e., the "standard does not cover pre-market [activities]").

Roles and Responsibilities The standard identifies several important roles that must be performed:

- *Responsible Organization:* The organization (e.g., a hospital) that is "entirely accountable for the use and maintenance of a MEDICAL IT-NETWORK" is responsible for the "RISK MANAGEMENT PROCESS for the MEDICAL IT-NETWORK, spanning planning, design, installation, [etc.]" Two elements of the organization have more specific responsibilities:
  - TOP MANAGEMENT*:* A number of responsibilities cannot be delegated beyond the organization's management, e.g., establishing risk-management policies, ensuring the provision of resources, and ensuring participation.
  - *Medical IT-Network Risk Manager:* Supervision of the overall RISK MANAGEMENT process itself, however, is to be led by one person, who has a number of responsibilities, e.g., managing and reporting on the process to top-level managers.
- *Medical Device Manufacturer:* Though the standard does not apply to pre-market activities (i.e., device development), various pieces of information (termed "ACCOMPANYING DOCUMENTS") need to be provided by a device's manufacturer.

[4]See Footnote 3 on page 32

- *Providers of other information technology:* The providers of, e.g., servers, software, middleware, etc., are also expected to make available certain pieces of information like requirements, known vulnerabilities, etc.

Risk Management The top management of the responsible organization is responsible for producing a policy and process for balancing the three key properties across four main areas:

- *Risk Management Planning and Documentation:* This involves determining the "risk-relevant assets" (e.g., hardware, software, etc.), producing supporting documentation (and enforceable responsibility agreements), which combine to form the risk management plan.

- *Risk Management:* Broadly, documentation of four steps are required (note that these steps—section 4.4 of the standard—resemble/reference ISO 14971):

  - *Risk Analysis:* First, hazards arising from the network are identified, as are risks of their occurrence.
  - *Risk Evaluation:* Then, the responsible organization determines whether or not risk control measures are required.
  - *Risk Control:* Next, a number of substeps collectively comprise the risk control process including considering options for, implementation of, and verification of risk control measures.
  - *Residual Risk Evaluation:* Finally, the risks of the hazards are reconsidered with the appropriate compensatory measures in place.

- *Change-Release and Configuration Management:* Responsible organizations are required to have a process in place for changes. This process should involve risk management of proposed changes unless a change permit exists (and such a permit can only be produced through previous risk management activities.)

- *Live Network* Risk Management*:* Responsible organizations are also required to monitor their network and document "negative events" as well as compensatory actions.

IEC 62304 A fourth standard that should be considered is ANSI/AAMI/IEC 62304 *Medical Device Software—Software Life Cycle Processes* [50]. It provides an excellent, high-level view of the full software development process for medical devices as well as providing definitions for standard terms and concepts. Note that it exists at a higher level than ARP4761, in that all four phases of 4761's process (Concept Development, Preliminary Design, Detailed Design, and Design Validation and Verification) are contained in IEC 62304's first step (Development).

There are two important concepts in IEC 62304:

- *Software Safety Classification:* The standard introduces three classes of software-driven medical device safety:

Class A	"No injury or damage to health is possible"
Class B	"Non-serious injury[5] is possible"
Class C	"Death or serious injury is possible."

- *Software of Unknown Provenance:* More commonly used as the initialism "SOUP," this term refers to a "software item that is already developed... for which adequate records of the development processes are not available." Considering the behavior of such software which is typically acquired in a non-bespoke, off-the-shelf manner can pose a significant challenge to a safety assessment.

In addition to the concepts, there are detailed descriptions of five processes which collectively comprise the software development lifecycle. The processes, which will overlap to some extent, are:

[5]See Footnote 3 on page 32

1. *Development:* This process entails everything from planning and analysis through implementation, testing, and release.

2. *Maintenance:* This focuses on planning for and implementing modifications for problems discovered after release.

3. *Risk Management:* This largely aligns with (and refers to) ISO 14971, though it contains additional steps to deal with SOUP (e.g., the standard's section 7.1.3).

4. *Configuration Management:* This deals with processes and documentation for configuring both bespoke and SOUP software items.

5. *Problem Resolution:* This process requires things like investigating software problems, advising relevant parties, and maintaining records.

2.3 Architecture Modeling

We now turn our focus to documentation of software architectures, the "set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both." This documentation can be created for a number of reasons—and used in a number of ways—but we focus here on that documentation which is created to enable automated analyses and used as a basis for (machine-readable) code and (human-readable) report generation.

2.3.1 Why Model System Architecture?

In the text *Documenting Software Architectures*, Clements et al. explain that there are three primary reasons for documenting a system's architecture [11]. First, the documentation "serves as a means of education," primarily of people who are previously unexposed to a system. For example, when a new developer has been hired to work with an existing system, the documentation of that system's architecture is a good place to start his training. Second, the documentation "serves as a primary vehicle for communication among stakeholders." Stakeholders here can refer to a number of people, both the traditional client, whose money is being used to pay for the system's construction and also the developer(s) themselves. Third, documentation "serves as the basis for system analysis and construction;" this work closely examines the use of architecture documentation for both of these purposes. Our eventual solution should be a process and accompanying tool that facilitates these goals.

Using Architecture Documentation for System Analysis

Documentation is necessarily a reduced version of a full system—if every aspect of a given system were fully documented, the system would essentially be reproduced in the language of the documentation (e.g., narrative english, box-and-line diagrams) rather than the language of the implementation (e.g., Java). How can an analyst decide which aspects of the system should be included in the documentation, and which should not? She should focus on those aspects that would differentiate the system from a competitor; Clements et al. term these *quality attributes*. The primary quality attribute the developers of MAP apps are concerned with is, unsurprisingly, safety; though that actually entails a number of other attributes, like real-time aspects (i.e., predictable performance) and correctness (even in the face of errors, i.e., fault-tolerance). Thus, any architecture documentation a MAP app developer would create should clearly highlight issues related predictable performance and behavior in the presence of errors. Which properties are necessary for analysis of real-time and safety-related concerns are well established (see, e.g., [51, 19] for real-time properties and Section 2.2.2 for safety-related properties).

An analyst's job of determining what exactly to document, even after establishing the set of relevant quality attributes, is not a straightforward task. Clements et al. explain that architectural documentation has three obligations related to the attributes:

- *Indicate which Quality Attribute Drove the Design:* Since safety is driving the design of (likely) every MAP app, this obligation is less significant than it might be in other systems. That said, if it is at all unclear why a particular design decision is safer than other options, that should be documentable via our process and tooling.

- *Capture Solutions Chosen to Satisfy the Quality Attribute Requirements:* This obligation coincides nicely with the next topic of using the architecture documentation for system construction. Capturing the solution that, when constructed, successfully realizes the goal of being safe is—as a truism—necessary for the construction of that system.

- *Capture a Convincing Argument why the Solutions Provide the Necessary Quality Attributes:* Meeting this obligation will lead to a preference for more rigorous (or even semi-formal) documentation formats, since making a convincing argument is typically easier in such styles. This obligation also fits nicely with our goal of using the system's architecture documentation for system construction, since automation of that relies heavily on the formality of the documentation.

Using Architecture Documentation for System Construction

Sufficiently descriptive architecture documentation should be usable—by a developer with working domain knowledge (or at a minimum a reasonable set of assumptions)—as the sole guide in the creation of a system's architecture. That is, while architecture documentation will necessarily leave out the non-architectural aspects of a system (i.e., the logic that converts inputs to outputs, sometimes referred to as *business logic*), a developer should be able to create *shells* (sometimes referred to as "skeletons") for each component. These shells typically consist, in object-oriented software, of things like the classes and methods of a program (though not the logic within the methods) as well as the creation of application program interfaces (APIs) that enable access to necessary services: e.g., networking, resource management, etc.

This process can be automated if architecture documentation is done in a machine readable language through the process of *code generation*. Code generation—the process of translating some machine-readable input to some machine-readable output—is a well-known technique in computer science (and is typically associated with the compilation process, see e.g., [52]). In this work, however, we used an analyzable architecture modeling language called AADL and created a code generator that produced shells of MAP apps which, after business logic was added, are runnable on the MDCF. This language and tooling are described in Chapter 3.

2.3.2 Architecture Modeling Techniques

Techniques for modeling the architecture of a software-based system are numerous, and their study is its own substantial field within software engineering. We briefly summarize current thinking here in order to situate the work, but do not go into great depth.

Clements et al.: Documenting Software Architectures

Clements et al. state that architecture documentation consists of *elements*, their *relations* to one another, and their *properties* [11]. An architectural style is comprised of a "specialization of element and relation types, together with a set of constraints on how they can be used", and the result of applying the style to a system is a *view* of that system's architecture. That is, a view is "a representation of a set of system elements and the relationships associated with them." Clements et al. identify three main categories of architectural styles:

- *Module Views:* focus on *modules*, which are "implementation [units] of software that [provide] a coherent set of responsibilities."
- *Component and Connector Views:* focus on *components*, which are runtime entities that are processing units in an executing system, and connectors, which are runtime pathways of interaction between two or more components.

- *Allocation Views:* focus on "the mapping of software units to elements of an environment in which the software is developed or executes."

Standards-Based Approaches

As in the system and medical-device safety domains, there are standards that address the techniques used to document software-based systems. Two particularly relevant standards are IEC 10746 and IEEE 42010.

IEC 10746 ISO/IEC 10746, titled *Information Technology – Open Distributed Processing* (ODP) focuses on five viewpoints of a system and the correspondences between them [53]. Note that, unlike Clements et al., these viewpoints do not necessarily correspond cleanly to design- or run-time constructs (e.g., modules or components, respectively) but rather "to five clear groups of users of a whole family of standards." [54] The five viewpoints, which are summarized by Linington et al. in [54] are:

1. *Enterprise:* This viewpoint covers the needs of the enterprise (where enterprise is defined as "any activity of interest"), including the "objectives, business rules, and policies that need to be supported by the system being designed."

2. *Information:* This viewpoint aims to synchronize the information stored and manipulated in the system between the other perspectives, so as to prevent incongruous definitions from cropping up amongst the viewpoints.

3. *Computational:* This viewpoint considers the "high-level [object-oriented] design of the processes and applications supporting the enterprise activities" and refers to the information viewpoint for the actual data stored in the particular objects. Allocation to runtime resources is not specified in this viewpoint.

4. *Engineering:* This viewpoint describes the interactions between the various computational constructs. These descriptions form a set of guarantees provided to the compu-

tational viewpoint, which are referred to as *transparencies*.

5. *Technology:* This viewpoint considers the realization of the system in the real world, and "represents the hardware and software components of the implemented system, and the communication technology that provides links between these components."

Additionally, Linington et al. discuss the language(s) for representing ODP viewpoints, and note that textual, graphical, or machine-readable formats may be desirable for various stakeholders/tasks. The different languages and notations used by the stakeholders gives rise to the need for *correspondences*, which provide links between the elements of different viewpoints.

IEEE 42010 ISO/IEC/IEEE 42010, which is titled *Systems and Software Engineering—Architecture Description*, generalizes the notion of stakeholder-driven viewpoints [55]. Clements et al. explain that this concept is found in IEC 10746 as well as other existing approaches, "such as Kruchten's 4+1 approach, Zachman's Architecture Framework, and even the DoD Architecture Framework..." [11] At a high level, the standard requires a) the identification of stakeholders, b) identification of their "architecture-related concerns," c) a set of viewpoints that collectively address those concerns, d) a set of views that have a one-to-one mapping onto the set of viewpoints, e) a set of models that compose the views, and f) rationale justifying the architectural decisions made [11].

Techniques Used in this Work

Clements et al. explain that the approach detailed in [11] is compatible with IEEE 42010, and provide a mapping; a similar mapping has been proposed for describing this work in terms of IEC 10746. In this work, though, we primarily use the following styles, as identified by Clements et al.:

- *Decomposition:* This design-time, module-based style is used to relate software and hardware elements to their containing systems (e.g., MAP apps are composed of medical devices and software, the software is composed of processes which are themselves made up of threads, etc.).

- *Pipe and Filter:* This run-time, component-and-connector based style is used to show how information arrives in a component via some input port, is transformed by the component, and then leaves via some output port.

While these are some of the architectural documentation styles most well-suited to the MAP app domain, our choice was also guided to a large extent by our use of the architectural modeling language AADL.

2.3.3 Technological Approaches to Architecture Modeling

Clements et al. explain that there are three categories or styles of notation for architecture documentation [11]:

1. *Informal Notations:* These are formats that are typically hand-drawn on a whiteboard or in a document, with semantics described only in natural language.

2. *Semiformal Notations:* These are formats that have some well-defined rules, though there is not the full mathematical precision of a formal notation.

3. *Formal Notations:* These are formats that have fully and precisely defined semantics, which enables a number of automated tools, including those for analysis and for code generation.

In this work we did not consider informal notations since much of the MAP vision relies on the software-driven automation of tasks. We note, though, that in general these notations are quite useful, and a "sketch" of a system in some informal notation may precede its initial development in one of the more formal languages.

Unified Modeling Language and Systems Modeling Language

Add UML Diagram?

Unified Modeling Language (UML) UML is perhaps the best-known modeling language for software-based systems. It has a graphical notation, and Clements et al. note that it “has grown to become the de facto standard for representing software designs in systems of all kinds.” [11] Indeed, UML is even the de facto standard for ODP diagrams; a tailored version of the language is used throughout [54].

Though the language began as integration of object-oriented methods, it has grown to include architectural description capabilities [56]. UML provides a number of different types of diagrams, and Clements et al. explain that a developer should not expect or attempt to use all of them but should rather pick those most relevant to the system he’s building. While UML is useful for communicating much about a system, it is not suited to our needs because it is too broadly focused and can be ambiguous (in fact, Clements et al. offer a number of pieces of advice on avoiding “UML Ambiguity Traps”).

Add SysML Diagram?

Systems Modeling Language (SysML) SysML is a systems-modeling language that is developed as a profile of UML to support “the specification, analysis, design, verification, and validation of [systems which] may include hardware, software, information, processes, personnel, and facilities.” [57]. To that end, SysML solves one of the shortcomings identified in UML, in that it is much more tailored to the domain we work in. SysML adds a focus on the lifecycle of system development, and includes diagrams for things like requirements, while carrying over some UML diagrams e.g., sequence and package. As it is based on UML, though, it still has ambiguous semantics.

There are efforts, though, to formalize subsets or particular diagrams of UML and SysML: e.g., SysML’s Activity Diagram [58], UML’s Statechart Diagram [59], or UML’s sequence Diagram [60]. The tool support for these formalizations is disparate at best as most tools for UML and SysML are focused on flexible diagramming rather than enabling analysis.

Architecture Analysis & Design Language

The Architecture Analysis and Design Language (AADL) is a model-based architecture description language developed in 2004 by the Society of Automotive Engineers (SAE) [61]. It enables a developer to completely model a system's architecture: the software that provides the required functionality, the hardware that the software runs on, as well as the binding from the latter to the former. It was "designed to provide... modeling and analysis capabilities" that "enable the analysis and prediction of a product's critical aspects before it is operational." [62]

Tool and Industry Support AADL is supported by a number of tools, including RAMSES, Ocarina, and OSATE2 [63, 64, 65]. OSATE2 consists of an Eclipse-based IDE for AADL and a collection of plugins for architectural analysis. The language has previously been used successfully on a number of projects in a range of domains including Boeing's integrate-then-build approach in its System Architecture Virtual Integration (SAVI) effort, modeling of security and non-functional behavior, and code generation and compositional verification of medical devices [66, 67, 68, 69].

Basic Usage AADL has both a graphical and textual declarative syntax—which can be instantiated to an instance model—that describe a system's components and the connections between them. Figure 2.10 shows an example of the graphical notation, while the other figures in this section use the textual syntax. An AADL model consists of zero or more property sets and packages, which themselves are composed of any combination of component types, component implementations and annex libraries [62].

- *Property Sets:* These are collections of either pre-defined or custom property types and values that are used in the model. Property sets are attached to one or more

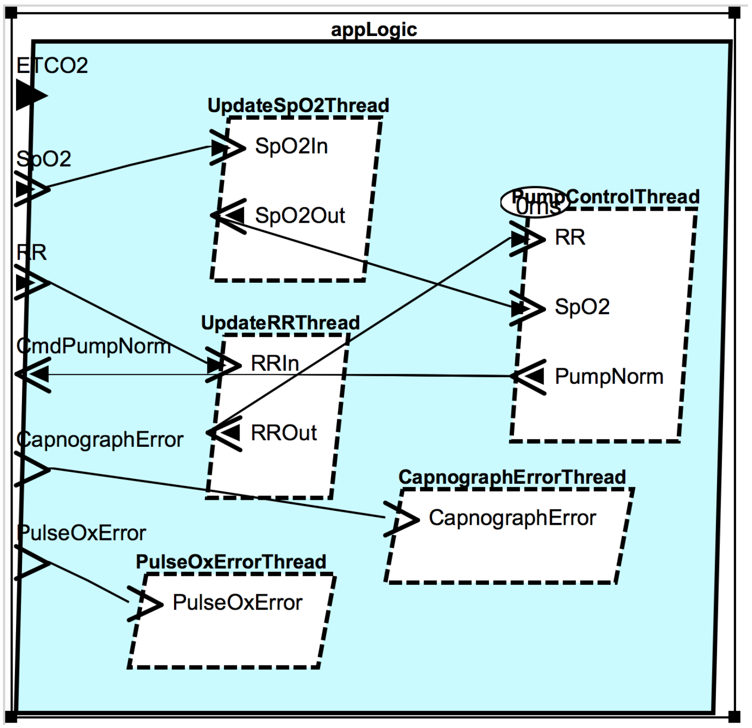


Figure 2.10: *A graphical view of an AADL component. The textual view of this component is shown in Figures 2.12 and 2.13.*

```
property set MAP_Properties is
    Process_Kind : type enumeration (logic, display);
    Process_Type : MAP_Properties::Process_Kind applies to (process);
    Component_Kind : type enumeration (actuator, sensor);
    Component_Type : MAP_Properties::Component_Kind applies to (device);
    Output_Rate : Time_Range applies to (port);
    Channel_Delay : Time applies to (port connection);
    Worst_Case_Execution_Time : Time applies to (thread);
    Exchange_Name : aadlstring applies to (port);
end MAP_Properties;
```

Figure 2.11: *A simple AADL property set*

components, and used by the various analyses performed by AADL tooling, e.g., timing properties would be used by latency analysis. Figure 2.11 shows an example of an AADL property set. Lines 2 and 4 show the creation of new property types while lines 3 and 5-9 show the creation of new properties. Note how the set of components that a property can be used on is restricted by its "applies to" clause, e.g., the

```
package PCA_Shutoff_Logic
public
with PCA_Shutoff_Types, PCA_Shutoff_Properties, MAP_Properties;

    process ICEpcaShutoffProcess
    features
        SpO2 : in event data port PCA_Shutoff_Types::SpO2;
        ETCO2 : in data port PCA_Shutoff_Types::ETCO2;
        RR : in event data port PCA_Shutoff_Types::RR;

        CapnographError : in event port;
        PulseOxError : in event port;

        CmdPumpNorm : out event data port PCA_Shutoff_Types::PumpCmd;
    flows
        spo2_flow: flow path SpO2 -> CmdPumpNorm;
    properties
        MAP_Properties::Process_Type => logic;
    annex EMV2 {**
        use types PCA_Shutoff_Errors;
        error propagations
            SpO2 : in propagation {SpO2ValueHigh};
            ETCO2 : in propagation {ETCO2ValueHigh};
            RespiratoryRate : in propagation {RRLow, RRHigh};
            CmdPumpNorm : out propagation {InadvertentPumpNormally};
            flows
                HighSpO2LeadsToOD : error path SpO2{SpO2ValueHigh} ->
  ↪ CmdPumpNorm{InadvertentPumpNormally};
                HighETCO2LeadsToOD : error path ETCO2{ETCO2ValueHigh} ->
  ↪ CmdPumpNorm{InadvertentPumpNormally};
                LowRRLeadsToOD : error path RR{RRLow, RRrigh} ->
  ↪ CmdPumpNorm{InadvertentPumpNormally};
        end propagations;
    **};
    end ICEpcaShutoffProcess;

    -- Package continues below...
```

Figure 2.12: *A simple AADL component type. A graphical view of this component is shown in Figure 2.10*

`Process_Type` property can only be used on `Process` components.

- *Packages:* These are collections of component types, implementations, and annex libraries. Figures 2.12 and 2.13 show an example package declaration.

  – *Component Types:* Component types are specifications of component interfaces,

```
    -- Package continues from above...

    process implementation ICEpcaShutoffProcess.imp
    subcomponents
        UpdateSpO2Thread : thread UpdateSpO2Thread.imp;
        UpdateRRThread : thread UpdateRRThread.imp;
        PumpControlThread : thread PumpControlThread.imp;

        CapnographErrorThread : thread CapnographErrorThread.imp;
        PulseOxErrorThread : thread PulseOxErrorThread.imp;
    connections
        incoming_spo2 : port SpO2 -> UpdateSpO2Thread.SpO2In;
        incoming_rr : port RR -> UpdateRRThread.RRIn;
        rr_to_pump_ctrl : port UpdateRRThread.RROut -> PumpControlThread.RR;
        spo2_to_pump_ctrl : port UpdateSpO2Thread.SpO2Out -> PumpControlThread.SpO2;
        pump_cmd_out : port PumpControlThread.PumpNorm -> CmdPumpNorm;

        capnog_err_in : port CapnographError -> CapnographErrorThread.CapnographError;
        pulseox_err_in : port PulseOxError -> PulseOxErrorThread.PulseOxError;
    end ICEpcaShutoffProcess.imp;
end PCA_Shutoff_Logic;
```

Figure 2.13: *A simple AADL component implementation. A graphical view of this component is shown in Figure 2.10*

and are categorized into one of: hardware (i.e., processor, memory, bus, etc.), software (i.e., data, thread, process, etc.), composite (system), or generic (abstract). Figure 2.12 shows an example AADL component type, in this case a process. All declarations and properties that are relevant to the type itself (as opposed to specific implementations) are aggregated in a component's type declaration, including:

∗ *Features:* Lines 7-14 are the external features of the component: incoming and outgoing communication points (ports in this case) which the component uses to communicate with other components.

∗ *Flows:* Line 16 is a flow specification, which documents how messages arriving on incoming communication points affect the component's output. They

```
package MAP_Errors
public
annex EMv2
{**
    error types

    -- ================================ --
    -- Timing Errors Applied to an App --
    -- ================================ --

    PhysioParamLate : type extends ErrorLibrary::LateDelivery;
    ControlActionLate : type extends ErrorLibrary::LateDelivery;
    PhysioParamFlood : type extends ErrorLibrary::HighRate;
    ControlActionFlood : type extends ErrorLibrary::HighRate;
    MissedPhysioParamDeadline : type extends ErrorLibrary::DelayedService;
    MissedControlActionDeadline : type extends ErrorLibrary::DelayedService;

    end types;

    -- ============================== --
    -- Error behavior of a Component --
    -- ============================== --

    error behavior MAP_Errors
    use types MAP_Errors;
    events
        badInput: error event;
        badAlgorithm: error event;
    states
        working : initial state;
        flaky : state;
        failed : state;
    transitions
        working -[badInput]-> failed;
        working -[badAlgorithm]-> flaky;
    end behavior;
**};
end MAP_Errors;
```

Figure 2.14: *Part of a simple AADL Annex Library*

can be high level, as in this example, or trace a more detailed path through the component's subcomponents. Additionally, this specification can be used by a flow specification in a higher level of abstraction (e.g., in the system component that contains this process).

∗ *Properties:* Line 18 is a property attachment, which sets the value of a property for all instances of this component type.

    * *Annexes:* Lines 19-31 are error modeling annotations for this component type. The details of this particular block are specific to the annex itself, and other annex annotations for things like behavior modeling would be attached to this component type in a similar fashion.

- *Component Implementations:* Component implementations specify the architectural implementation of component types. There may be zero or more implementations of a given type; determination of which implementation is used is part of the model instantiation process. Figure 2.13 continues the package that was started in Figure 2.12; it shows an example AADL component implementation. Implementations contain one possible architectural decomposition of the component type: i.e., its subcomponents (lines 39-44), their interconnections (lines 46-53), and any declarations specific to the particular implementation. Note that the connections in a component implementation are both between subcomponents (e.g., line 48) and between subcomponents and the component's external interface (e.g., line 46).

- *Annex Libraries:* AADL has a number of language annexes which extend its modeling scope beyond system architecture, to things like behavior or error modeling [70, 33]. Figure 2.14 shows an example package consisting only of an error modeling annex library.

Error Modeling Annex The AADL error modeling annex (in its second version, abbreviated EMV2) consists not only of special properties and declarations that can be attached directly to component types and implementations (e.g., lines 19-32 of Figure 2.12) but also high-level "libraries" which can contain error types, failure state machines, and other constructs (e.g., Figure 2.14) [33].

The annex has a number of features including a type system for describing error hierar-

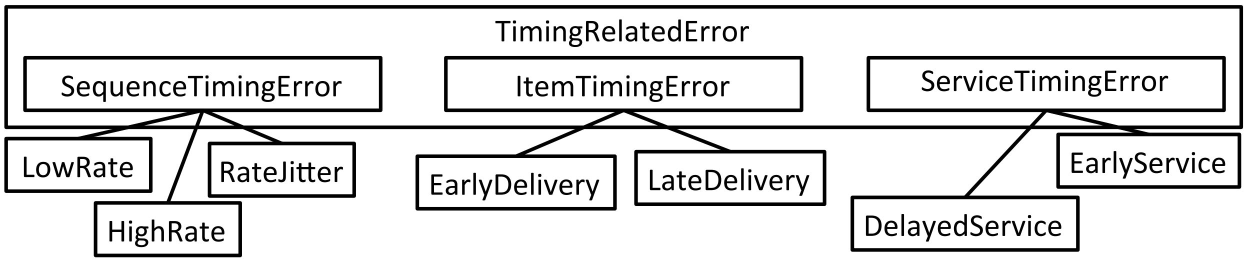


Figure 2.15: *Timing related errors in the EMV2 error type hierarchy, from [6]*

chies and a library of common error types, which are refined from five "root" error types. Figure 2.15 shows the timing related errors—other root error types are related to value, service, replication and concurrency problems—which can then be further refined by a developer to be more tailored to a system's specific architecture and implementation, as in Figure 2.16. The error type propagation aspects of the annex (e.g., lines 22-29 of Figure 2.12) are based on Wallace's fault propagation and transformation language and calculus; they enable a developer to model error types, sources, propagations, behavior, detections, etc. [33, 35]. Lines 11-16 of Figure 2.14 show the declaration of some fault types (those corresponding to the red errors in Figure 2.16), while lines 24-36 show a generic behavior state machine of a component that may get bad input (in which case it fails) or be built incorrectly (in which case it becomes unreliable). Larson et al. have previously examined the application of EMV2 to a safety-critical medical device, and found that it enabled the realization of a number of benefits, including (a) formal, machine-readable inputs; (b) tight integration with the architectural model; and (c) automated construction of FTA and FMEA-like reports [71].

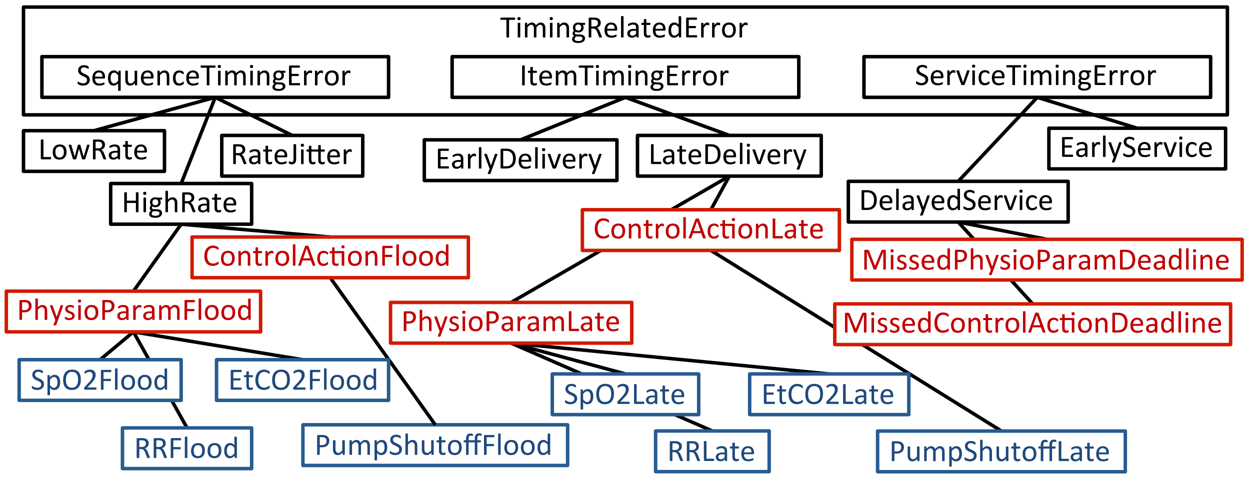


Figure 2.16: *Timing related errors extended first to the ICE Architecture (red) and subsequently the PCA interlock app (blue), from [6]*

Chapter 3

An AADL Subset for MAP Apps

3.1 Introduction

Although work has been done on MAP app requirements (e.g., [72]) and the architecture of MAPs (e.g., [73, 13]), little attention has been paid to the architectures of MAP apps themselves. As discussed in Section 2.1.1, these MAP apps are essentially new medical devices[1] that are: a) specified by an application developer, b) instantiated at the point of care, and c) coordinated by the MAP itself. Previously, most MAP apps were built to run on prototype platforms and were designed in an ad-hoc manner with the goal of demonstrating certain functionality concepts rather than as full-blown, industrial-strength implementations.

What is needed is to move from this ad-hoc approach to something that can enable systematic engineering and reuse. Such an approach would enable true component-based development, which could utilize network-aware devices as services on top of a real-time, publish-subscribe middleware. These components (and their supporting artifacts) could be composed by an application at runtime to define the medical system's behavior, though this would require careful reasoning about the architecture of the application. While this careful

[1]In fact, MAP apps are sometimes referred to as *virtual medical devices* or *platform-constituted medical devices*.

analysis of MAP apps has not been performed before, the architectures of other bus-based, safety-critical systems have been given a great deal of attention. Much of that attention has been focused on AADL (see Section 2.3.3).

Since MAP app development is in need of more engineering rigor and AADL was developed to provide architectural modeling and analysis for safety-critical systems, it seems natural to evaluate their combined use. However, it is not immediately clear whether AADL, a technology aimed at the integration of hardware and software in the automotive and aerospace industries, will be applicable to the domain of MAP apps. For example, since MAPs do not expose the raw hardware of their platform—rather programming abstractions above it—it is unclear how well certain AADL features (e.g., those designed to model hardware) will work when only these software abstractions are important.

What is needed, then, is: a) a subset of AADL that is useful when describing the architecture of MAP apps, and b) a supporting set of tools to facilitate app development. In this chapter, we describe a proposed subset and prototype toolset that target the Medical Device Coordination Framework (MDCF, see Section 2.1.3); while specific to MDCF, we believe our work generalizes to other similarly rigorously engineered MAPs.

Specifically, in this chapter we describe:

1. A proposed subset of the full AADL (selected components and port-based communication) that is useful for describing a MAP app's architecture.

2. A proposal for a set of properties necessary for describing the real-time (RT) and quality-of-service (QoS) properties of MAP apps. This set includes some of AADL's built-in properties, and it also utilizes AADL's property description mechanism to specify new properties.

3. An implementation of a translator that takes as input the relevant properties and app component structure (as identified in 1 and 2) and produces as output an application for the MDCF. Specifically, the translator produces code that automatically:

   (a) Handles non-developer-modifiable activities, e.g., task instantiation, port-to-channel binding, message marshalling/unmarshalling, etc.

   (b) Configures the component layout via the underlying publish/subscribe middleware as specified in the architectural model.

   (c) Enforces the RT and QoS parameters via properties described in 2.

4. A runnable app which demonstrates the expected translator output and implements the previously discussed PCA Interlock Scenario. The architecture of the app is specified in our proposed subset of AADL and the output is runnable on the MDCF.

3.1.1 App Development Environment Vision

At the center of the long-term MAP app development vision is an App Development Environment (ADE) that is built on an industrial grade Integrated Development Environment (IDE) in order to support model-driven development of apps. The IDE should provide access to traditional software development tools including editors, compilers, debuggers, and testing infrastructure. The ADE should then add on to that a pluggable architecture so that a variety of components for supporting app analysis, verification, and artifact construction can be easily connected to the environment. Additionally, the envisioned ADE should enable compilation and packaging of an app so that it can be directly deployed and executed on a MAP.

In addition to supporting traditional development, an important aspect of our vision is that the ADE should, in the long-term, support preparation of a variety of artifacts required for third-party certification and regulatory approval of the app. For example, the ADE might support the construction of requirements documents with capabilities enabling requirements to be traced to both individual features and formal specifications in the app model and implementation. The ADE should support preparation of hazard and risk analysis artifacts which should also be traceable to models/code. We envision a variety of possible forms of verification such as a) app to device interface compatibility checking, b) component implementation to component contract conformance verification, c) model checking of real-time aspects of concurrent interactions, d) error and hazard modeling using the EMV2 annex for AADL [71], and e) proof environments that support full-functional correctness proofs of an app's behavior. The ADE could also support construction of rigorous styles of argumentation, such as assurance cases (again, with traceability links to other artifacts). The ADE might also support preparation of third-party certification and regulatory submission documents (e.g., the FDA's 510(k) or specific documentation in the format of a relevant standard, see Section 2.2.5), as well as the packaging of artifacts into digitally-signed archives that would be shipped to relevant entities. These organizations would be able to use the same framework to browse the submitted artifacts and re-run tests/verification tools.

The work presented in this chapter describes the MDCF Architect, and it represents a necessary, enabling first step towards achieving this vision. The work presented in Chapter 4 is a second step, in that it achieves an architecturally integrated hazard analysis technique for MAPs. Installation information and user-targeted documentation for the MDCF Architect is available at santoslab.org/pub/mdcf-architect.

3.1.2 Mechanization and Regulatory Authorities

The enforcement of safety standards, e.g., those described in Section 2.2.5, is a primarily manual one that can take significant amounts of time. Since one of the key advantages of the MAP vision is that many apps will be considerably simpler to build than traditional devices, a corresponding change to the regulatory approval process will be necessary to avoid a backlog of unapproved MAP apps. One way to greatly reduce the amount of manual effort necessary for understanding the intended use of an app is to have much of the verification work done automatically. A logically separate, machine readable—and machine verifiable—architecture is one of several artifacts that could aid in this automated verification. Further, hazard analyses (i.e., FMEA, FTA, or STPA) that are currently not performed, or are developed in isolation from an app's implementation, could be built on top of an app's architectural description. AADL's error modeling annex, for example, already enables developers to generate FMEA and FTA-like reports from annotated models of a system's architecture, and Chapter 4 describes annotations and tooling to support an STPA-like process for MAP apps. We believe that a similar process could reduce the (currently substantial) difficulty involved in verifying claims about an app's behavior. This would enable the linking of claims about an app's functionality to the implementation of that functionality, and this traceability could further reduce the burden on regulatory authorities.

3.2 Why AADL for MAPs?

Before we examine the subset of AADL that we use, though, we provide an in-depth justification for why we believe AADL is particularly well-suited for MAP app architectures.

3.2.1 Medical Application Platforms

MAP applications, as distributed systems, are built using the traditional "components and connections" style of systems engineering. Any development environment for apps, then, should have a number of core features that are important to component-based development:

- *Support for well-defined interfaces:* The components of distributed systems should be self-contained, with the exception of the inputs and outputs they expose over well-defined interfaces. This enables a library of commonly used components to be made available to developers.
- *Support for common patterns of communication:* Not only are the components of such a system often reusable, but so are the styles of communication between the components (see, e.g., [74]). Adhering to common patterns will also result in a more

straightforward software verification process.

- *Support for real-time and quality-of-service configuration:* In a real-time, safety-critical distributed system, computational correctness requires not only correct information, but also getting it at the correct time. Safety arguments can be difficult to make without the ability to set expected timings in an app's configuration (e.g., a task's worst-case execution time) and have a guarantee of the enforcement of those timings from the underlying middleware.

The translator and example artifact portions of this work target the MDCF because it supports a rigorous notion of component specification. Since the MIDdleware Assurance Substrate (MIDAS) [19] is one of the middleware frameworks supported by the MDCF, our translator supports setting a range of timing properties attached to both connections (e.g., a port's latency) and components (e.g., a task's worst-case execution time). As previously noted, though, the work described here is not deeply tied to the MDCF but could be targeted to any MAP implementation that supports similarly rigorous notions of component definition, configuration, and communication.

We believe the core concepts common to definitions of app architectures are:

- *Layout:* A high-level schema that defines how various components connect to one another and collectively form the app.

- *Medical Device:* Individual medical devices which will either be controlled by software components, or produce physiological information that will be consumed by software components.

- *Software Component:* Software pieces that typically (though not exclusively) consume physiological information and produce clinically meaningful output: e.g., information for a display, smart alarms, or commands to medical devices.

- *Channel:* Routes through a network over which components (i.e., both medical devices and software) can communicate.

Taken together, the core features and concepts enable reusability by ensuring that components communicate over interfaces in common, pattern-based ways with strict timing constraints. A component is capable of interoperating in any other system where it “fits;” that is, its interface exposes the required ports, utilizes the same communication patterns, and has compatible timing requirements.

3.2.2 Architecture Analysis & Design Language

As explained in Section 2.3.3, AADL is a standardized, model-based engineering language that was developed for safety-critical system engineering, so it has a number of features that make it particularly well suited to our needs:

- *Hierarchical refinement:* AADL supports the notion of first defining an element and then further refining it into a decomposition of several sub-components. This will not only keep the modelling elements more clean and readable, but will also allow app creators to work in a top-down style of development. They will be able to first think about what components make up the system and how those components would be linked together, define those components, and finally reason about how those individual components would themselves be comprised.

- *Distinction between types and implementations:* AADL allows a developer to first define a component and then give it one or more implementations, similar to the object-oriented programming practice of first defining an interface and then creating one or more implementations of that interface. This keeps app models cleaner and enables code re-use.

- *Extensible property system:* AADL allows developers to create properties, specify their type, and restrict which constructs they can be attached to. We have used

AADL Construct	MAP Concept
Components	
System	Layout
Device	Device
Process	Software Component
Thread	Task
Connections	
System Level	Channel
Process Level	Task Trigger
Process Implementation Level	Task-Port Communication

Table 3.1: *AADL syntax elements and their MAP app mappings*

this feature to, for example, associate various physiological parameters with their IEEE11073 nomenclature "tag" [25].

- *Strong tool support:* AADL is supported by a wide range of both open source and commercial tools. We have used OSATE2 as the basis for our toolset, and have found a number of its features quite useful (e.g., element name auto-completion, type-checking, etc.).

3.2.3 Why subset AADL?

In general, AADL models are composed of components and their connections describing a complete "co-designed" system: software, hardware, and the bindings between the two. To support this, AADL includes a number of constructs for modeling software entities (e.g., thread, process, data, etc.), hardware entities (e.g., processor, memory, device, etc.), and composite entities (e.g., system and abstract). AADL also includes connections between components of various types such as ports, data accesses, bus accesses, etc. However, since MAPs are managed platforms, i.e., developers author only the software elements while the platform manages the allocation of (and bindings to) hardware resources. That is, since we use AADL in a restricted manner to focus—*exclusively*—on what an app developer would need to specify to enable construction of a MAP app, we have pared down the language considerably.

Therefore, we do not use the full AADL—because the hardware elements are fixed, along with the allocation of the software elements to them—but instead use only the subset specified in Table 3.1. Further, nearly every element of our subset[2] has had its semantics modified, at least slightly, from the original AADL specification. Two features that are representative of the problems that would be caused by the use of AADL without modification are the `process` construct, and `data access` connection:

- *`Process` Construct:* Feiler and Gluch write that "An AADL process represents a protected address space that prevents other components from accessing anything inside the process and prevents threads inside a process from causing damage to other processes." [62, pg. 135] Clearly, in a Java-based MAP like the MDCF, creation of protected address spaces is not easily achievable or (likely) desirable. Instead, as explained in Section 3.3.3, an AADL process in our subset translates to a Java class, which has many of the same desirable properties regarding encapsulation.

- *`Data Access` Connection:* The same AADL text explains that "... multiple threads may operate on a common data area (i.e., they share access to a data component)... In AADL, you model this through data access features and access connections." [62, pg. 210] Similar to the `process` construct, realizing the semantics of `data access` connections would not only be difficult or impossible in a distributed, publish-subscribe system like the MDCF, but doing so would not be particularly useful either.

Support for currently-unused constructs may be added in the future, as our modeling language expands to use more of the features of AADL; see the discussion in Section 7.1.1.

[2]The exception is the `device` construct.

3.3 Language Walkthrough

In this section, we describe the process of creating a MAP app with our prototype toolset using the motivating example of the PCA Interlock app initially described in Section 2.1.5. This app consumes various physiological parameters to determine the health of the patient and disables the flow of an analgesic when there are indications of respiratory failure. A high-level, ICE-configuration/logical view of the app is shown in Figure 2.4. The diagram shows that in addition to the PCA pump, there are four sensors: a) a blood-oxygen saturation (SpO_2) sensor, b) a pulse rate sensor, c) a respiratory rate sensor, and d) an end-tidal carbon dioxide ($ETCO_2$) sensor. In this application, the sensors may be on the same device: SpO_2 and pulse rate information are often produced by a pulse oximeter (e.g., the Ivy 450C [75]), and respiratory rate and $ETCO_2$ information can come from, e.g., a capnography machine (e.g., the Capnostream 20 [76]).

The PCA pump consumes information from the app (e.g., tickets) while the other devices produce information in the form of sensor data that is used by the app's logic. An important part of the app (and indeed the entire MAP vision) is that it will use suitable physiological parameters regardless of their source; that is, instead of building the app to work with a specific device or set of devices, it is built to work with a generic source of the required physiological parameters.

Figure 3.1 gives an overview of the app architecture development, code generation, and app instantiation process. Part (A) of the figure shows the various AADL artifacts that compose the app; note that they are labeled with the number of the subsection they are discussed in. Part (B) shows the execution and configuration artifacts that result from code generation, which are discussed in the next section. It also highlights the large number of components (signified by dashed lines) whose generation is completely automated. Part (C) shows the app's instantiation on a running MAP, which was first sketched in Figure 2.4, with both the computation hosting and communication aspects of the app having been realized in the ICE architecture.

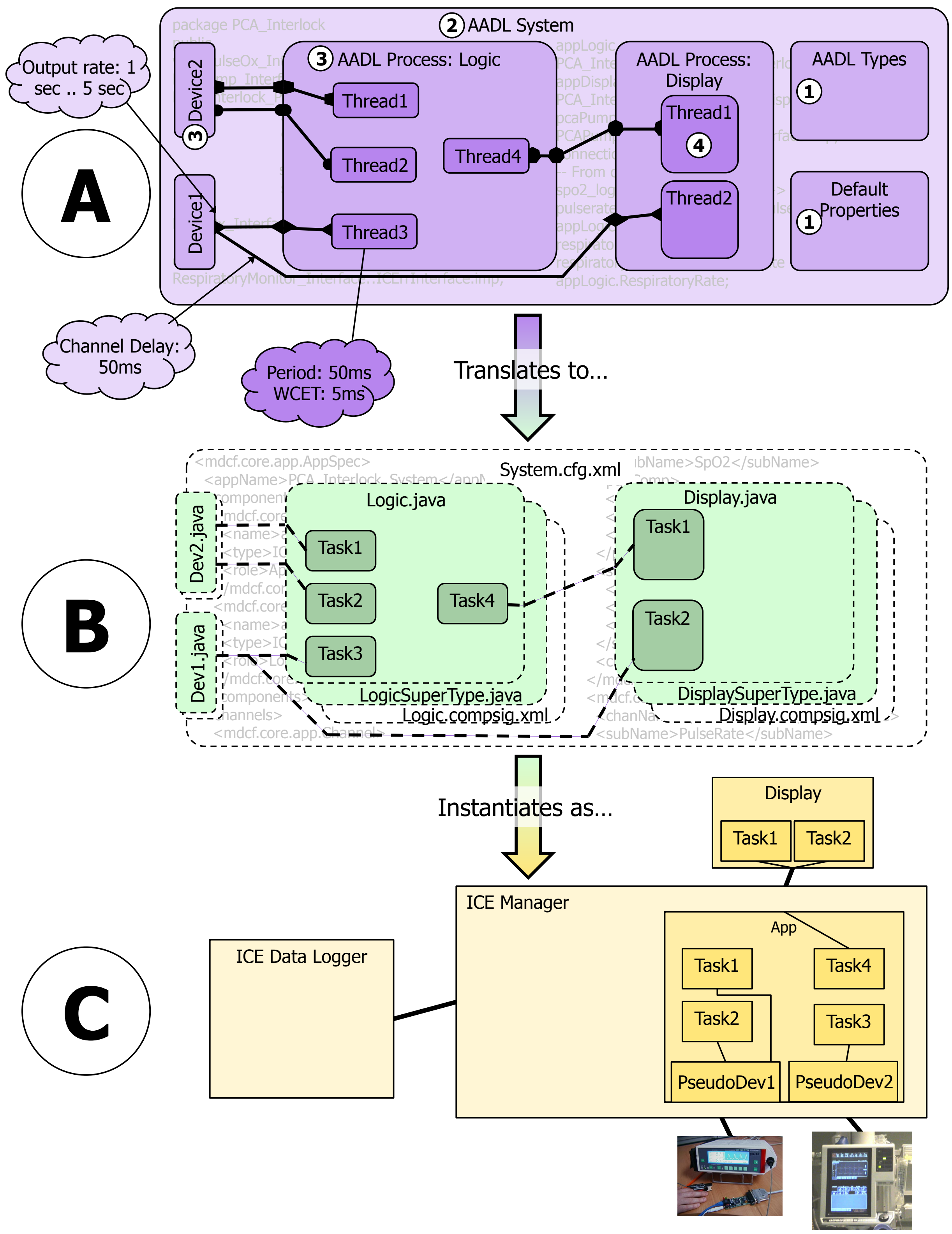


Figure 3.1: *(A) The AADL platform artifacts used by the code generation process, (B) the generated app configuration and executable files, and (C) the fully configured and executing platform.*

```
package PCA_Interlock_Types
public
with Data_Model, IEEE11073_Nomenclature;

  data SpO2
  properties
    Data_Model::Data_Representation => Float;
    IEEE11073_Nomenclature::OID =>
  ↪  IEEE11073_Nomenclature::MDC_PULS_OXIM_SAT_O2;
    Data_Model::Real_Range => 0.0 .. 100.0;
  end SpO2;

  data Ticket
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Integer_Range => 0 .. 600;
  end Ticket;

end PCA_Interlock_Types;
```

Figure 3.2: *The SpO_2 datatype used in the app excerpt*

This section presents excerpts of AADL models that specify the application architecture which, when used with our translator, results in application code runnable on the MDCF. For clarity, we only show one physiological parameter: SpO_2, though the full app would contain all four parameters (i.e., SpO_2, pulse rate, respiratory rate, and $ETCO_2$). In the next section, we discuss AADL types and default properties, followed by a top-down walkthrough of the hierarchy of components used by our toolset.

3.3.1 Preliminary tasks: Types and Default Properties

Before we can describe a MAP app's architecture, we should briefly examine AADL's type and property definition mechanisms (marked by a (1) in part (A) of Figure 3.1) and how they are used to specify various parameters in our app.

```
property set PCA_Interlock_Properties is

  -- Synchronize thread period and deadlines by using this default
  Default_Thread_Time : constant Time => 50 ms;

  -- A periodic task will be dispatched once per period
  Default_Thread_Period : Time
  ↪  => PCA_Interlock_Properties::Default_Thread_Time applies to (thread);

  -- A task will be scheduled so that it has time to run before its deadline
  Default_Thread_Deadline : Time
  ↪  => PCA_Interlock_Properties::Default_Thread_Time applies to (thread);

  -- The most time that a task will take to execute after it is dispatched
  Default_Thread_WCET : Time => 5 ms applies to (thread);

  -- Periodic tasks are run once per period, sporadic upon message arrival
  Default_Thread_Dispatch : Supported_Dispatch_Protocols => Sporadic applies
  ↪  to (thread);

  -- Ports must specify the most / least frequently they will broadcast
  Default_Output_Rate : Time_Range => 100 ms .. 300 ms applies to (port);

  -- The maximum time a message will spend on the network
  Default_Channel_Delay : Time
  ↪  => 100 ms applies to ({PCA_Interlock} ** port connection);

end PCA_Interlock_Properties;
```

Figure 3.3: *The default properties used in the app excerpt*

Data Types:

The data type for the SpO_2 parameter and pump-control tickets are shown in Figure 3.2. The SpO_2 parameter is stored as a floating point number between 0.0 and 100.0, while tickets are stored as integers between 0 and 600 seconds to allow tickets between zero and ten minutes in length. Note that these ranges are not enforced by the code generator since Java doesn't natively support restricting ranged types.

AADL's property description mechanism is easily extensible, allowing us to specify customer-specific metadata. In this example, we have leveraged this capability to attach an IEEE11073 nomenclature "tag" with our SpO_2 parameter [25]. Note that these datatypes could either be generated from or mapped down to a more standard interface definition language (e.g., CORBA IDL [77]).

Default property values:

While it is useful to be able to attach properties to individual AADL constructs (e.g., ports, connections, threads, etc.), sometimes a large number of constructs take the same values for certain properties. In this case, it is useful to set app-wide defaults, as shown in Figure 3.3. These properties apply to every applicable element, unless overridden. Override names, as well as types and example values, are specified in Table 3.2.

3.3.2 The AADL System

The top level of the app architecture is described by an AADL `system`, marked by a (2) in Figure 3.1, and shown textually in Figure 3.4. Systems have no external features (lines 5-7), though the `system implementation` lists their internals (lines 9-33). In our subset an AADL system implementation consists of, at its core, a declaration of sub-components (e.g., devices and processes) and the connections between them.

```
package PCA_Interlock
public
with SpO2Req_Interface, PCAPump_Interface, PCA_Interlock_Logic,
  ↪  PCA_Interlock_Display, MAP_Properties, PCA_Interlock_Properties;

  system PCA_Interlock_System
    -- No external features since we're at the top level of the app
  end PCA_Interlock_System;

  system implementation PCA_Interlock_System.imp
  subcomponents
    -- Some device, probably a pulse oximeter, that produces SpO2
    spo2Device : device SpO2Req_Interface::SpO2Interface.imp;

    -- A logic component to turn SpO2 readings into PCA pump tickets
    appLogic : process PCA_Interlock_Logic::ICEpcaInterlockProcess.imp;

    -- Logic to drive a display for a clinician
    appDisplay : process PCA_Interlock_Display::ICEpcaDisplayProcess.imp;

    -- The PCA pump attached to the patient
    pcaPump : device PCAPump_Interface::ICEpcaInterface.imp;
  connections
    -- From sensors to controller
    spo2_logic : port spo2Device.SpO2 -> appLogic.SpO2;

    -- From controller to PCA pump
    TicketChannel : port appLogic.Tickets -> pcaPump.Tickets
    {MAP_Properties::Channel_Delay => 50 ms;};

    -- From controller to display
    spo2_disp : port spo2Device.SpO2 -> appDisplay.SpO2;
    TicketDisplay : port appLogic.Tickets -> appDisplay.Tickets;
  end PCA_Interlock_System.imp;

end PCA_Interlock;
```

Figure 3.4: *The top-level app excerpt architecture via the AADL system component*

Name			
Override	Default	Type	Example
	Thread		
Default_Thread_Period	Period	Time	50 ms
Default_Thread_Deadline	Deadline	Time	50 ms
Default_Thread_WCET	Worst_Case_Execution_Time	Time	5 ms
Default_Thread_Dispatch	Dispatch_Protocol	Sporadic or Periodic	Periodic
	Port		
Default_Output_Rate	Output_Rate	Time Range	1 ms .. 3 ms
	Port Connection		
Default_Channel_Delay	Channel_Delay	Time	100 ms
	Process		
N/A	Process_Type	Logic or Display	Display

Table 3.2: *AADL properties used in our MAP-targeted subset*

3.3.3 The AADL Process and Device

Now that the software and hardware elements — the AADL processes and devices marked by a (3) in Figure 3.1 — have been referenced by the AADL system implementation, a developer must specify their type and implementations.

AADL Processes:

A process defines the boundaries of a software component, and is itself composed of a number of threads (which are described in Section 3.3.4). The type of a process in AADL is a listing of its interface points, i.e., the ports it uses to communicate with other components (see lines 7-12 of Figure 3.5). The `process implementation` is, like the system implementation that was discussed in Section 3.3.2, a listing of `subcomponents` and `connections`. Though the full AADL specification allows processes to have a number of different types of subcomponents and connections, our subset restricts these to threads and port connections. These connections are directional links between a thread and one of the process's ports,

```
package PCA_Interlock_Logic
public
with PCA_Interlock_Types, MAP_Properties;

  process ICEpcaInterlockProcess
  features
    -- Incoming SpO2 data arrives on this port
    SpO2 : in data port PCA_Interlock_Types::SpO2;

    -- Tickets sent to the pump leave on this port
    Tickets : out event data port PCA_Interlock_Types::Ticket
    {MAP_Properties::Output_Rate => 1 min .. 10 min;};
  properties
    -- If this process drove a display, this would be set to display
    MAP_Properties::Process_Type => logic;
  end ICEpcaInterlockProcess;

  process implementation ICEpcaInterlockProcess.imp
  subcomponents
    -- Note that we don't need a thread for the SpO2 port, since it's not an
  ↪  'event' or 'event data' port

    -- This thread calculates the length of ticket
    CalcTicketThread : thread CalcTicketThread.imp;
  connections
    -- Outgoing tickets from the thread are sent out of the process on the
  ↪  'Tickets' port
    outgoing_ticket : port CalcTicketThread.Tickets -> Tickets;
  end ICEpcaInterlockProcess.imp;

end PCA_Interlock_Logic;
```

Figure 3.5: *An AADL process specification used in the app excerpt*

```
package SpO2Req_Interface
public
with PCA_Interlock_Types, MAP_Properties;

  device SpO2Interface
  features
    -- SpO2 information leaves the device via this port
    SpO2 : out event data port PCA_Interlock_Types::SpO2;
  end SpO2Interface;

  device implementation SpO2Interface.imp
    -- This implementation is empty since our subset only describes device interfaces
  end SpO2Interface.imp;

end SpO2Req_Interface;
```

Figure 3.6: *An AADL device used in the app excerpt*

which allows messages received by the process to be routed to particular threads, and vice versa. Note that as both logic and display components are modeled as AADL processes they are distinguished from one another via the `MAP_Properties::Process_Type` property (line 15).

Ports must be labeled as `data`, `event`, or `event data`. The presence of `data` signifies a payload (and requires a type specification, i.e., `PCA_Interlock_Types::SpO2` on line 8 of Figure 3.5), while the presence of `event` signifies that a developer wants to be notified of (and handle via a **`thread`**) incoming messages. If `event` is not specified then the incoming data is simply stored (in the MDCF, this takes the form of a predictably-named **`protected`** field). `data` ports can be particularly useful in apps where there are a large number of physiological parameters: rather than specify behavior to be executed each time a message arrives, the most recent data can simply be used when needed.

AADL Devices:

Apps describe the interfaces of the devices they need to connect to using the AADL `device` construct (see Figure 3.6). Device components are placeholders for actual devices that will

```
thread CalcTicketThread
features
  -- Tickets leave the thread via this port
  Tickets : out event data port PCA_Interlock_Types::Ticket;
properties
  -- Our default properties don't fit for this thread, so we override them
  Thread_Properties::Dispatch_Protocol => Periodic;
  Timing_Properties::Period => 50 ms;
  Timing_Properties::Deadline => 10 ms;
  MAP_Properties::Worst_Case_Execution_Time => 5 ms;
end CalcTicketThread;

thread implementation CalcTicketThread.imp
  -- Thread implementations aren't specified in our subset of AADL
end CalcTicketThread.imp;
```

Figure 3.7: *Two AADL thread interfaces used in the app excerpt*

be connected to the app when it is launched. These actual devices will produce information and send it out over ports whose type matches the specification (see line 8 of Figure 3.6). Note that the `device implementation` is left empty, since the app's device needs can be met by any device that realizes the interface requirements.

3.3.4 The AADL Thread

AADL threads, marked by a (4) in Figure 3.1, represent semi-independent units of functionality, and are realized in the MDCF as MIDAS tasks (see Figure 3.7) [19]. They can be either `sporadic`, which signifies that they are executed when a port that they are "attached" to receives a message, or `periodic`, where they are executed after a specified period of time. Typically, threads which consume information operate sporadically (so they can act as soon as updated data arrive), and threads which produce information operate periodically.

Thread implementations are empty because this is the lowest level of abstraction supported by our subset; all work below this is implemented by a developer within code "skeletons" that are auto-generated by our translator. Figure 3.7 shows the interface of the `CalcTicketThread` which generates the tickets for the PCA pump. The skeleton and implemented form of this thread are shown in Figure 3.8 and Figure 3.9.

3.4 Code Generation and Instantiation

Once the previous AADL constructs have all been fully specified, an app's architecture description is complete. The next step, which is fully automatic, is to generate the MAP-compatible, executable code (part (B) of Figure 3.1). Our translator will interpret the AADL to create a model of the app, and then render it to a target MAP implementation; currently the only implementation supported is the MDCF.

For an app to run on the MDCF, a number of files must be generated. There are two types of files produced: .java files which contain the logic of the component, and .xml files[3] which specify the RT/QoS properties to the underlying middleware.

3.4.1 Executable Code Skeletons

Figure 3.8 shows a simplified user-modifiable executable "skeleton" for the PCA interlock logic module (the architecture of which was specified in Figure 3.5). Note that there is only one thread and an empty initialization function. These skeletons are ready to be edited, and can—leveraging the fact that OSATE2 is built on Eclipse, and this is also a Java development environment—be immediately loaded back into OSATE2 where the app architecture was specified [65].

At this point, an app developer can create as complex or simple of an app as she would like. A very simple example is shown in Figure 3.9. Note that a full-blown implementation would be larger and more complex, e.g., it might additionally use ETCO$_2$, respiratory rate, and pulse rate information (compare, for example, the complexity of Figures 2.12 and 2.13 to Figure 3.5, which was used to generate the examples in this section), and would likely have a far more sophisticated process for determining the length of tickets. A more complex

[3]These files are serializations of Java classes which are deserialized using XStream [78]

```java
package mdcf.app.pca_interlock;

import mdcf.channelservice.common.MdcfMessage;

public class ICEpcaInterlockProcess extends ICEpcaInterlockProcessSuperType {

  public ICEpcaInterlockProcess(String GUID, String host) {
    super(GUID, host);
  }

  @Override
  protected void initComponent() {
    // TODO Fill in custom initialization code here
  }

  @Override
  protected void CalcTicketThreadMethod(){
    // TODO: Fill in custom periodic code here
  }
}
```

Figure 3.8: *Executable "skeletons" produced by the translator*

algorithm that incorporates the findings of the hazard analysis portion of this dissertation is discussed in Section 6.1.2.

In the implementation shown in Figure 3.9, each time the `CalcTicketThreadMethod` is run (which is triggered periodically by the scheduler) it will check to see if the pump has a valid ticket (lines 24-27) and if so, it terminates. If not, though, the current SpO_2 value is used to calculate a ticket value (lines 29-41) which is sent out over the `TicketsSenderPort` port (line 44).

Of course, there's much more to making the app run than what's visible in Figure 3.9. Activities that are not user-modifiable, e.g., registering tasks with the scheduler and ports with the network are handled by an automatically generated abstract class; an example is shown in Figure 3.10. Though the class has been compressed somewhat for space, important regions of the code are the:

- *Constructor:* Shown in lines 7-13, this initializes the ports, creates the objects that provide the tasks' behavior and state, and puts the various objects into mappings used

```java
package mdcf.app.pca_interlock;

import java.time.Instant;

import mdcf.channelservice.common.MdcfMessage;

public class ICEpcaInterlockProcess extends ICEpcaInterlockProcessSuperType {

  private int previousTicketValue = -1;
  private Instant previousTicketExpires;
  private final int TICKET_DURATION_STEP = 60; // 1 minute increments

  public ICEpcaInterlockProcess(String GUID, String host) {
    super(GUID, host);
  }

  @Override
  protected void initComponent() {
    // No setup necessary...
  }

  @Override
  protected void CalcTicketThreadMethod(){
    Instant now = Instant.now();
    if(previousTicketExpires.isAfter(now)){
        return; // The pump still has a valid ticket
    }

    double spo2 = getSpO2Data();
    int ticketLength = 0;
    if (spo2 > 100.0){
      // Something's wrong, leave duration at 0
    } else if(spo2 > 99.0){
      ticketLength = TICKET_DURATION_STEP * 5;
    } else if (spo2 > 97.0){
      ticketLength = TICKET_DURATION_STEP * 3;
    } else if (spo2 > 95.0){
      ticketLength = TICKET_DURATION_STEP;
    } else {
      // The patient can't take more analgesic, so we leave the duration at 0
    }

    previousTicketExpires = now.plusSeconds(ticketLength);
    TicketsSenderPort.send(ticketLength);
  }
}
```

Figure 3.9: *The same "skeletons" complete with business logic*

```java
/* Imports and package declaration removed for space */

public abstract class ICEpcaInterlockProcessSuperType extends LogicComponent{
  /* Field and abstract method declarations removed */

  /* The constructor initializes the fields and registers with the MDCF */
  public ICEpcaInterlockProcessSuperType(String GUID, String host) {
    super(GUID, "ICEpcaInterlockProcess", host);
    SpO2ReceiverPort = new MdcfReceiverPort<Double>("SpO2", Double.class,
↪  host);
    TicketsSenderPort = new MdcfSenderPort<Integer>("Tickets",
↪  Integer.class, host);
    taskInstanceMap.put(SpO2TaskTask.class.getSimpleName(), new
↪  SpO2TaskTask());
    taskInstanceMap.put(CalcTicketThreadTask.class.getSimpleName(), new
↪  CalcTicketThreadTask());
  }

  @Override /* Call the user-specified initialization method */
  public void init(){ initComponent(); }

  /* The auto-generated getter for our incoming data port named "SpO2" */
  protected Double getSpO2Data(){ return SpO2Data; }

  /* Pub / Sub handler registration removed for space */

  /* An autogenerated "set" for the SpO2 port */
  private void SpO2ListenerOnMessage(MdcfMessage msg, Double SpO2Data){
      this.SpO2Data = SpO2Data; }

  /* Tasks are the MDCF equiv. of AADL threads. Even though there is no SpO2
      thread, there is an implicit one created to handle incoming messages */
  public class SpO2TaskTask implements Task {
    @Override public void run() {
      MdcfMessage message = SpO2ReceiverPort.getReceiver().getLastMsg();
        try { Double SpO2Data = SpO2ReceiverPort.getLastMsgContent();
          SpO2ListenerOnMessage(message, SpO2Data); /* Call our setter */
        } catch (MdcfDecodeMessageException e) {
          System.err.println(getComponentTypeName() + ".SpO2TaskTask task:
↪  invalid message:" + message.getTextMsg());
          e.printStackTrace(); }
    }
  }

  /* This task will be run by the MDCF / MIDAS scheduler once per period */
  public class CalcTicketThreadTask implements Task {
    @Override public void run() { CalcTicketThreadMethod(); }
  }
}
```

Figure 3.10: *A partial logic-module "supertype" which hides most of the autogenerated code*

by the MDCF.

- *Message Handlers:* Handler methods for SpO_2 data are shown on lines 18-38. Though the exact configuration of the handler depends on the port's specification (i.e., whether it is a `event`, `event data`, or `data` port) there are some common pieces. These include the task itself (lines 29-38) which handles unmarshalling the data and calls the listener, which may be abstract or fully specified, as in lines 24-25.

- *Tasks:* Specified as classes that implement `mdcf.core.ctypes.Task`, sporadic tasks are instantiated with whatever message triggered their launch while periodic tasks are instantiated without parameters, as in lines 41-43.

3.4.2 App Configuration

In addition to the business logic skeletons, the MDCF Architect also produces both a layout schematic that describes both how the components of the app fit together and a collection of files specifying each component's real-time and quality-of-service settings.

App Topology Specification

Figure 3.11 shows an excerpt of an app configuration XML file; at app launch the runtime system deserializes this file and instantiates the software components. An app's configuration consists of three elements:

1. *App Name:* The application's name (line 2).

2. *Components:* A list of components that must be instantiated for the app. Each component consists of a name, the Java type to be instantiated, and the component's role: "AppPanel" for components that will need a user interface or "Logic" for logic components and device interfaces (lines 3-10). Note that as device components themselves are not instantiated (since they are physical objects) they are not listed here.

```xml
<mdcf.core.app.AppSpec>
  <appName>pca_shutoff</appName>
  <components>
    <mdcf.core.app.VirtualComponent>
      <name>appLogic</name>
      <type>ICEpcaInterlockProcess</type>
      <role>Logic</role>
    </mdcf.core.app.VirtualComponent>
    <!-- Other virtual components removed for space -->
  </components>
  <channels>
    <mdcf.core.app.Channel>
      <chanName>$PLACEHOLDER$</chanName>
      <pubName>Tickets</pubName>
      <subName>Tickets</subName>
      <pubComp>
        <name>appLogic</name>
        <type>ICEpcaInterlockProcess</type>
        <role>Logic</role>
      </pubComp>
      <subComp>
        <name>appDisplay</name>
        <type>ICEpcaDisplayProcess</type>
        <role>AppPanel</role>
      </subComp>
      <channelDelay>100</channelDelay>
    </mdcf.core.app.Channel>
    <!-- Other channels removed for space -->
  </channels>
</mdcf.core.app.AppSpec>
```

Figure 3.11: *An excerpt of the app's overall layout configuration*

3. *Channels:* A list of the connections between the components, consisting of the publishing and subscribing port names and component descriptions (lines 11-29). Information that is unknowable at compile-time, such as channel names and the names of specific devices, is represented by the string `$PLACEHOLDER$` and is replaced as the app is launched.

Component Timing Specification

A software component's RT/QoS specification is shown in Figure 3.12. Like the app topology, this class is deserialized at app launch, and used by the MDCF for scheduling. There are three parts of a component's timing specification:

1. *Type:* The Java type name of the component (line 2).

2. *Module Tasks:* A listing of the component's tasks and their properties (lines 3-20). In addition to timing properties (i.e., period, deadline, and worst-case execution time or "wcet") tasks also have a name and a "trigger port." Message arrival on a sporadic task's trigger port will cause the task to be executed with the value of the message as a parameter. Note that some values are unused: periodic tasks have an unused placeholder value for their trigger port, and sporadic tasks have an unused placeholder value for their period.

3. *Port Signatures:* A mapping of a component's port's names to their properties (lines 21-42). In addition to periodicity information, a port's name, direction, and the type of its payload is specified in each port signature.

3.4.3 Launching the App

Once the business logic has been specified, the app is ready to be launched on a compatible MAP. When the app is launched, the specifications will be deserialized, the executable artifacts will be instantiated, and if compatible devices and the necessary computational

```xml
<mdcf.core.ctypes.AppModuleSignature>
  <type>ICEpcaInterlockProcess</type>
  <moduleTasks>
    <mdcf.core.ctypes.TaskSignature>
      <type>PORT_SPORADIC</type>
      <trigPortName>SpO2</trigPortName>
      <periodMs><!-- Placeholder value: Sporadic task’s periods are derived
    ↪  from their triggering port -->-1</periodMs>
      <taskName>SpO2TaskTask</taskName>
      <deadlineMs>50</deadlineMs>
      <wcetMs>5</wcetMs>
    </mdcf.core.ctypes.TaskSignature>
    <mdcf.core.ctypes.TaskSignature>
      <type>PERIODIC</type>
      <trigPortName><!-- Placeholder value: Periodic tasks are triggered by
    ↪  time, not message arrival -->Placeholder</trigPortName>
      <periodMs>50</periodMs>
      <taskName>CalcTicketThreadTask</taskName>
      <deadlineMs>10</deadlineMs>
      <wcetMs>5</wcetMs>
    </mdcf.core.ctypes.TaskSignature>
  </moduleTasks>
  <portSignatures>
    <entry>
      <string>SpO2</string>
      <mdcf.core.ctypes.PortSignature>
        <portName>SpO2</portName>
        <portDirection>SUBSCRIBE</portDirection>
        <minPeriod>100</minPeriod>
        <maxPeriod>300</maxPeriod>
        <portType>Double</portType>
      </mdcf.core.ctypes.PortSignature>
    </entry>
    <entry>
      <string>Tickets</string>
      <mdcf.core.ctypes.PortSignature>
        <portName>Tickets</portName>
        <portDirection>PUBLISH</portDirection>
        <minPeriod>60000</minPeriod>
        <maxPeriod>600000</maxPeriod>
        <portType>Integer</portType>
      </mdcf.core.ctypes.PortSignature>
    </entry>
  </portSignatures>
</mdcf.core.ctypes.AppModuleSignature>
```

Figure 3.12: *The logic module’s configuration*

resources (cpu time, network capacity, etc.) are available the app will come online. Part (C) of Figure 3.1 shows how primary elements of the app excerpt would look on the MDCF at runtime.

3.5 Tailoring AADL to a Domain

In the design of our language subset, we encountered some challenges that could potentially be addressed by future revisions to AADL. Specifically, in Section 3.2.3, we discussed the need for our identification and use of a subset of AADL rather than the full language. We believe that other users of AADL who target managed platforms rather than a system's "raw hardware" would similarly want to exclude parts of the full language and / or slightly modify the semantics as we have.

Unfortunately, there is no straightforward way to make these modifications, especially if industrial-quality tooling is desired. That is, there is no automated enforcement of our subset's modified syntactical rules so syntax errors are only discovered and marked by the translator when translation is attempted, rather than as a developer uses a disallowed construct or writes erroneous code. We note that OSATE has a number of standard development-environment features, e.g., syntax highlighting, code completion, etc., but these features are not easily adaptable to subsets of AADL like ours.

What is needed, then, is some way for language users to identify the subset of the language they would like to work with, and then to enforce the boundaries of their modified language. In effect, this would allow users to create a domain-specific language (ideally through some sort of meta-modelling functionality[4]) that would be customized to their particular needs. The core of AADL—a collection of constructs, connections, and property specifications—would be present in any domain-specific incarnation, but the overhead in-

[4]To be explicit, we advocate a user-editable metamodel for AADL, which would be a type-model (in the language of Kühne) that describes the permitted connections, features, and properties for each language element. [79]

volved in creating and using a tailored subset, like the one identified in Section 3.3, would be greatly reduced.

Chapter 4

The SAFE Process

When initially examining how best to perform hazard analysis on the applications that run on Medical Application Platforms, we considered both Fault Tree Analysis (FTA) and Failure Modes and Effects Analysis (FMEA). They are both well-established, partially parallelizable/compositional, and had tool support in the OSATE, the AADL editing environment we had previously used (see Section 3.4). Additionally, they had a detailed process and collection of worksheets available, leaving little ambiguity to their implementation [7].

However, the research done by Leveson and her students includes compelling arguments against the use of non-systems-theory based analyses like FMEA and FTA [30, 80, 81, 82]. Following this realization, we attempted to use System Theoretic Process Analysis (STPA), but were unable to use it without significant modifications for four main reasons:

1. *STPA lacks a precisely defined process:* Though *Engineering A Safer World* explains the need for and motivation behind STPA in considerable detail, the process itself is only described at a high-level. The descriptions of STPA's steps are described in somewhat general terms rather than in an explicit, mechanizable level of precision. This is almost certainly a conscious choice on the part of Leveson—her process derives a great deal of power from its flexibility, particularly when analyzing sociotechnical elements like the behavior of humans or organizations.

2. *STPA is informal:* System safety is a large field, and there are formalizations of many concepts. Leveson, however, did not use or connect to mathematical treatments of many existing topics. These include formalisms for everything from causality (e.g., [83, 35]), to decomposition/refinement (e.g., [84]), to system architecture (e.g., [61]). We believe that many of these formalisms can be used to improve STPA's rigor and repeatability.

3. *STPA lacks a specific output format:* While there are STPA analyses of a number of example systems, there seems to be little convergence on output formats: unsafe control actions are sometimes presented in a tabular format (e.g., Table 7 in the Balgos Thesis, Table 11 of the Thornberry Thesis, or Table 9 of the Placke Thesis [80, 85, 86]), other times in graphical notation (e.g., Figure 8.7 of Leveson's text [30]), and sometimes in large, narrative or list-based documents (e.g., Section 4.4.3 of the Sotomayor Thesis [81]). Like the imprecision in the process specification, this is very likely by design: a single, rigid documentation format would likely be a poor fit for the full range of possible sociotechnical system elements.

4. *STPA is completely non-compositional:* Leveson explains that "[t]he systems approach focuses on systems taken as a whole, not on the parts taken separately" and her reasoning is well-documented in [30, pg. 63]. She writes that, in contrast to small systems (which can be completely examined) and huge populations (which can be analyzed statistically), some systems exhibit "organized complexity." She explains: "These systems are too complex for complete analysis and too organized for statistics." [ibid.]

We believe that once a system has been decomposed into subsystems that are small enough to be completely examined, a by-the-book STPA is—in some cases—no longer the most appropriate analysis. Specifically, the small software- and hardware-based systems (like MAP apps) that are small enough to be completely examined should be. We recognize, though, that MAP apps are used as parts of full sociotechnical systems, and so we produced a heavily modified form of STPA for these small software- and hardware-based subsystems. Our process is designed to “take over” from a standard STPA once analysis has reached a low level of a system’s abstraction hierarchy. That is, our process can use as input the outputs of an STPA that has been performed on the full system that encloses a low-level subsystem.

We call this process the *Systematic Analysis of Faults and Errors*, or SAFE. SAFE has two documentation formats: one that is spreadsheet-based and can be performed manually and one that consists of AADL constructs/properties and is tool-assisted. We use SAFE to refer to the generic process, but where differentiation is required we denote the manual version as M-SAFE and the tool-assisted version as T-SAFE.

In the next section, we introduce concepts and terminology that are unique to SAFE. The three sections following that describe the activities that collectively compose our process, and we conclude with an assessment of SAFE in Section 4.5. Note that the full, analyst-targeted process for M-SAFE is given in Appendix A, blank M-SAFE worksheets are given in Appendix B, and the complete PCA Interlock scenario analysis is given in Appendix C.

4.1 Core Concepts

SAFE is, at its core, a backwards-chaining analysis that moves through a system’s architecture in an iterative manner. While not fully compositional, it is much more component-oriented than STPA. Though STPA’s “whole system” approach is valid for analysis of the socio-technical aspects of systems, it is not mechanizable or parallelizable; both significant barriers to integration in a modern, component-based software-engineering process like we envision for MAP app development.

Component-based engineering is part of the bedrock of modern software engineering practice. One central tenet of component-based engineering is the notion of a component’s *interface*: when elements are designed to work in any conformant component-based system rather than bespoke designs, they must communicate with other elements exclusively via declared inputs and outputs. In this style, elements declare the types of messages they can receive and produce as well as other metadata required to ensure smooth interconnection with other elements, e.g., RT/QoS specifications. Modern hardware- and software-architecture specification languages like AADL include capabilities for specifying interfaces for failure-related behavior as well; see the EMv2 discussion in Section 2.3.3.

Indeed, Hatcliff et al. explain that the goals of the MAP vision include specific "Architecture and Interfacing" needs. These include, a) interoperability points (interfaces), b) interface compliance and compatibility[1] checking, and c) a "rich device interface language" that would be able to fully specify such interfaces [13]. Compositional certification is also raised as a challenge to certifiably-safe software-dependent systems in [87], which cites as one of the challenges "Limited Engineering Approaches and Ready-to-Use Solutions for Partitioning and Delimiting Emergent Behaviors in Dynamic Contexts."

Clearly, then, even a semi-compositional approach to hazard analysis needs a notion of component interface that supports descriptions of failure-related behavior. In this section we first closely examine two concepts that are central to these component interfaces: successor dangers and manifestations. We then describe areas where STPA's informality can be improved upon: fault identification, causality models, and decomposition. Next, we discuss terminology that is used throughout SAFE. Finally, we are able to specify the exact dependencies between the various activities in SAFE, which enables a precise discussion of what is, and is not, compositional and/or parallelizable in our process.

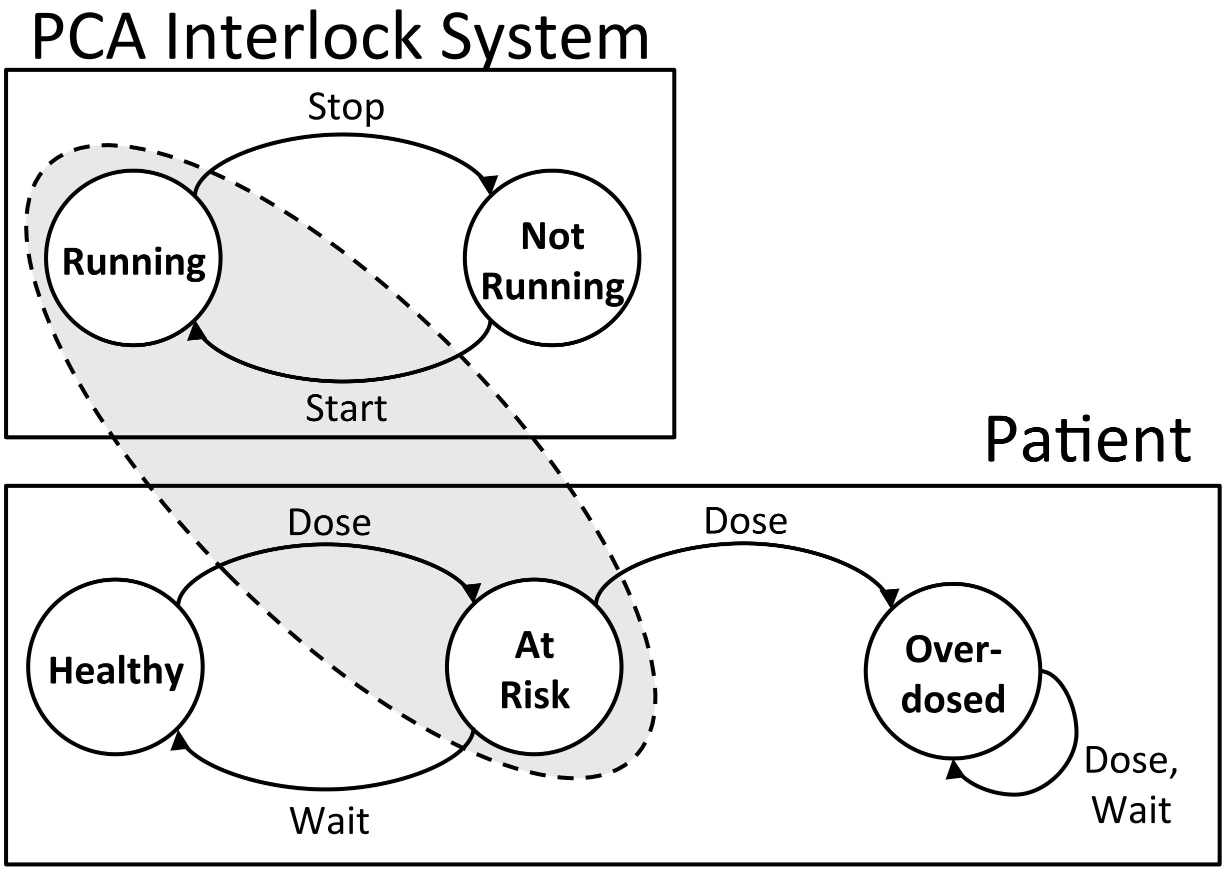


Figure 4.1: *A graphical representation of Leveson's definition of a hazard, which requires the system to be in some state and its environment to be in a worst-case state. Note that "worst-case" is specific to the system state, i.e., the Not Running state might have its own worst-case environment state(s).*

4.1.1 Successor Dangers

One of the key parts of Leveson's work is its two-part definition of hazard: not only must the system be in some state, but the environment must also be in a worst-case state for losses to necessarily occur. Her definition is "[a] system state. . . that, together with a particular set of worst-case environmental conditions, will lead to an accident." [30, pg. 184] Consider the PCA Interlock scenario: it is inaccurate to say that it is hazardous for the PCA pump to run, since this considers only the state of the system. Similarly, it is not accurate to say that

[1]Compliance essentially asks "Does the component implement its interface?" while compatibility asks "Does component A's interface allow it to work with component B?"

it is hazardous for the patient to be at risk of respiratory depression—if an at-risk patient is not given any more opioid, no damage to his health will occur—since this considers only the state of the environment. Rather, the hazard comes about when the PCA pump is administering more narcotic when the patient is at risk of respiratory depression.

Consider Figure 4.1, which shows the states of the app and the patient modeled as deterministic automata. The highlighted pairing—where the PCA pump is running and the patient is at risk of an overdose—is clearly a hazard. But what would happen if we decompose the system into its requisite elements, i.e., pump, app logic, sensor, and connections? Now only the pump can directly cause the hazard, since no state of the app logic or sensor will independently move the patient from "At Risk" to "Overdosed."

In order to solve this problem, we introduce the concept of *undesirability*. It is the analogue of Leveson's hazard in that it is a pairing of one system state and one environment state that will necessarily lead to a hazard, but it applies to components that are not at the system boundary. The critical difference between the two terms is that while a hazard requires an *intransitive* causal link between the state pairing and the notion of loss, undesirability requires a *transitive* causal link.

Consider Figure 4.2, which shows the PCA Interlock system from Figure 4.1 broken down into its requisite parts of pump, app logic, and a generic "Sensors" component. Undesirable app-state and patient-state pairings are highlighted by grey ovals. Observe that incorrect ticket values from the app are not hazardous, since they require a transitive step—through the PCA pump—in order to cause an overdose. They are clearly undesirable, though, both intuitively and under our definition.

While we formalize the concept of undesirability and explore its theoretical ramifications in the next chapter, for now we focus on using the notion of undesirability to move from Leveson's whole-system approach to SAFE's per-component approach. This leads to a slight reformulation of the question analysts must pose to themselves, since they must now avoid not system-level notions of loss but rather individual elements providing outputs that

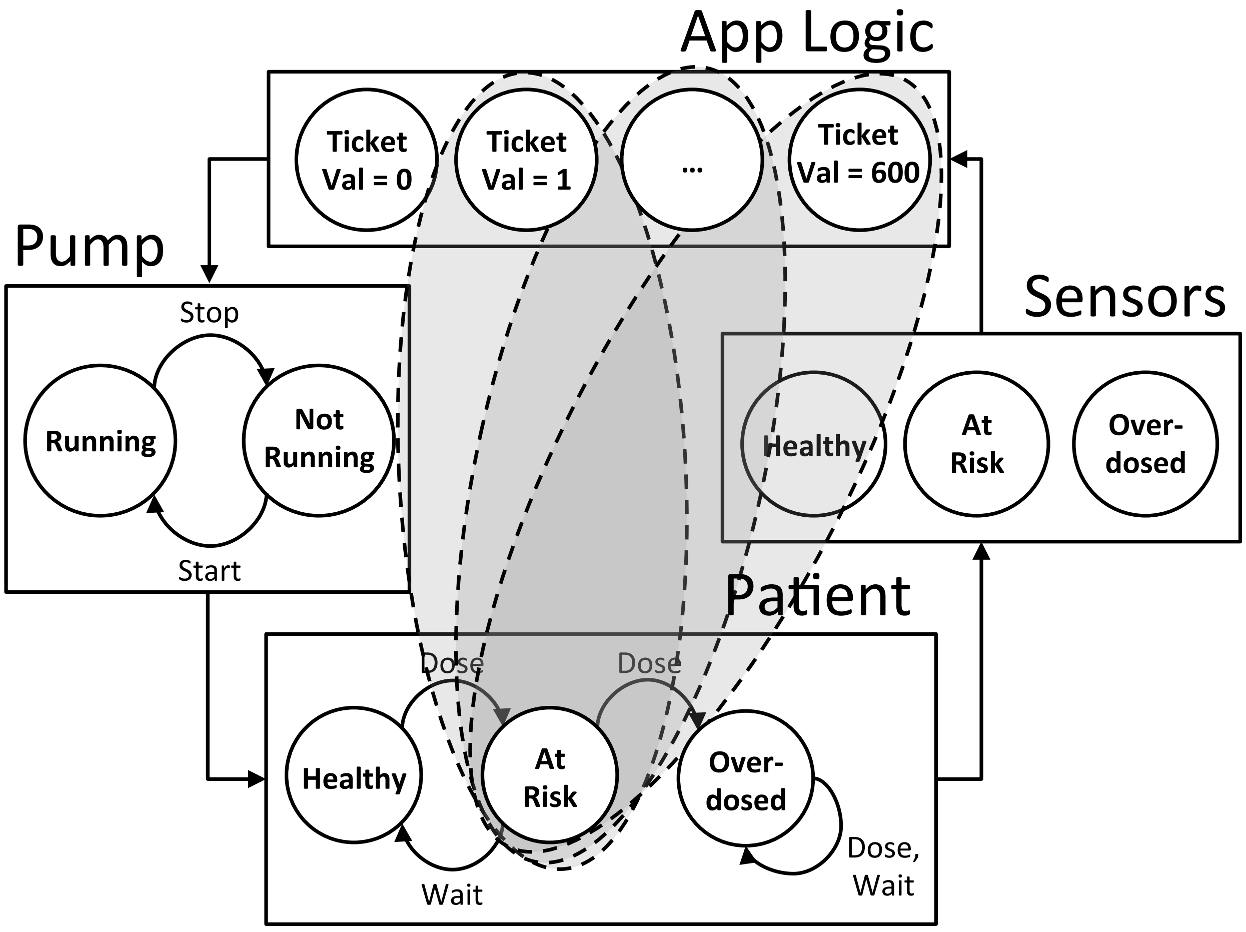


Figure 4.2: *A graphical representation of undesirable state pairings.*

are dangerous to their direct successors. Thus, while hazard avoidance asks "How can the system avoid causing loss?" undesirability avoidance asks "How can I avoid giving providing dangerous output to my successor?" The identification of this dangerous output is one of the key parts of SAFE. Undesirable outputs that are provided to a component's successor are termed *successor dangers*.

4.1.2 Manifestations

Bottom-up analyses like FMEA can be quite verbose since all failure modes of a component must be considered. Though this is a tractable strategy for simple, mechanical elements, it is less tenable for software-based or sociotechnical systems that can fail in myriad, unpredicatble ways. Other analyses, like FTA or STPA, are top-down. These approaches are by design much more limited in scope than bottom-up styled analyses, since faults and errors that do not lead to top-level hazards are never considered. A full discussion, and vocabulary of terms, on this topic is provided in Section 5.5.3.

In creating SAFE, we have attempted to merge the two approaches. Since we are aligning with STPA, full-system analyses should be top-down and thus driven by system-level notions of loss. However, since we also aim to be component-oriented—and to some degree compositional—we have to consider component-level failures in the absence of a full system design. Trying to consider every way that a software- or hardware-based element could fail can be very difficult. Further, the difficulty only compounds when a component must be connectable to other components, some of which may be created by external organizations or at a later date. There are so many ways that an error produced by a component could negatively impact any other component to which it might be connected that enumerating all of them can be difficult or impossible. This problem, which is common to other areas of software verification like model checking, is often referred to as a *combinatorial explosion* of possibilities.

What is needed is some way to compress the space of possible failure modes: a way to classify the errors that propagate from one component to another. One popular way to classify errors is by their appearance to the component that receives them. That is, regardless of what causes an error, we ask "How does the error *manifest* at its recipient?" Such an approach has been advocated by, e.g., Walter and Suri who wrote that a model they proposed "organizes diverse fault categories into a cohesive framework by classifying faults according to the effect they have on the required system services rather than by targeting the source of the fault condition." [88]

Though we do not use Walter and Suri's model, we note that the service failure domains from Avižienis et al. align well with their approach. Our adapted form of these domains—

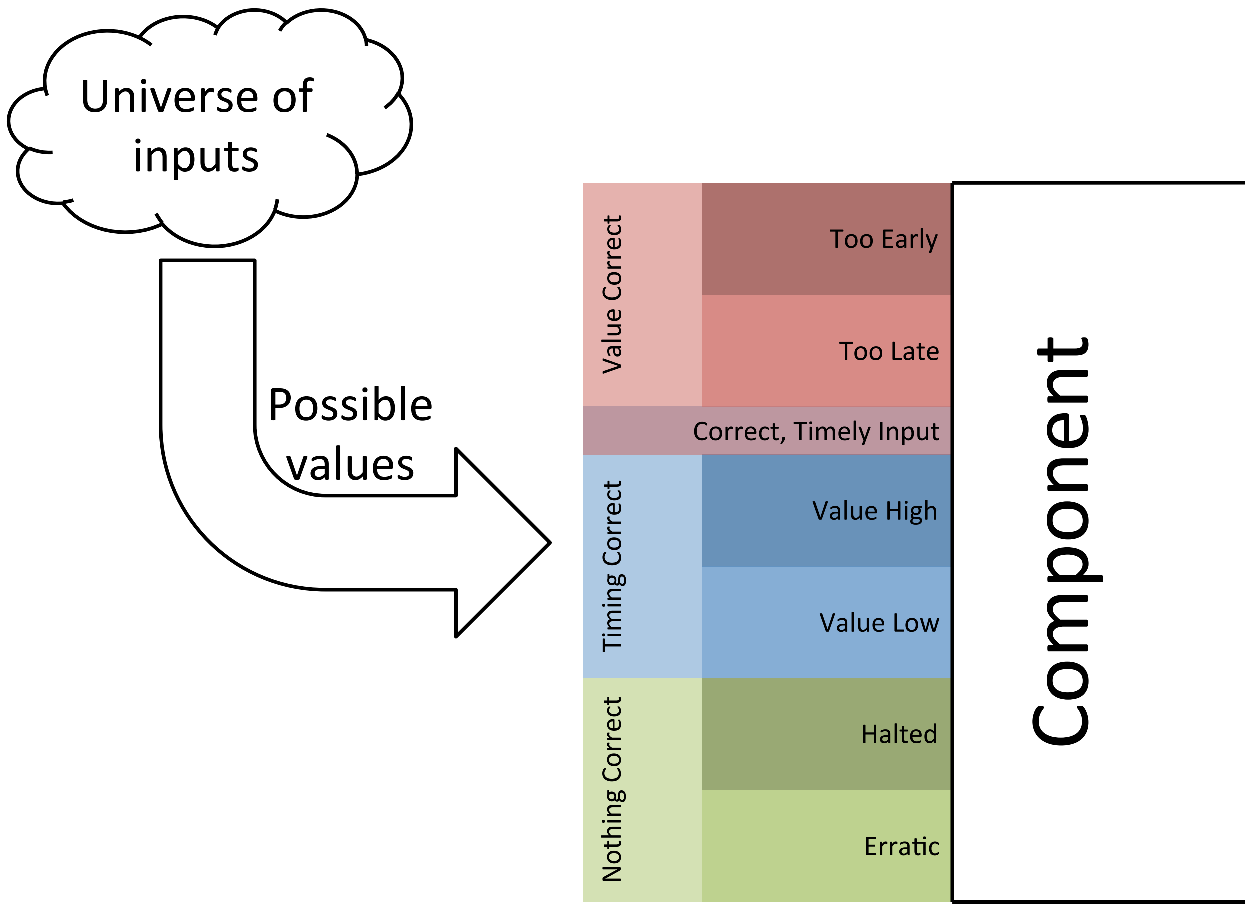


Figure 4.3: *The ways that input can fail (adapted from Avižienis et al.'s failure domains) [3]*

which we term *manifestations*—is shown in Figure 4.3, which is a graphical representation of the six ways that input errors can manifest to a receiving component[2]. That is, input can have a correct value but a) arrive too early or b) too late, it can arrive at the correct time but (assuming it is numeric) be c) incorrectly high or d) incorrectly low, it can e) not arrive at all, or f) arrive out of the blue. Non-numeric input can also have value errors, though further delineations will depend on the input's type.

[2]The seventh option is correct, timely input.

4.1.3 Fault Classification

Ideally, a similarly clean distinction would exist when dealing with faults, which are the "internally caused" analog to errors, which can be thought of as externally caused problems (a full discussion of these terms was provided in Section 2.2.1). Unfortunately, there are a number of ways to classify faults: e.g., by phase of creation, or phenomenonological cause, etc., and no single classification comfortably divides the large space of possible faults. Instead, Avižienis et al. describe eight fault classes, and argue that a given fault can be classified according to all eight. The classes they propose are: [3]

1. *Phase of Creation:* When the fault is created, i.e., design-time or runtime.

2. *System Boundaries:* Where the fault occurs, i.e., inside the system/element or outside of it. Note that we consider environmentally caused problems to be faults, rather than errors. Even though the fault source (e.g., water, heat, etc.) is external to the physical boundaries of the element, if it does not affect the element through an established link/port/interaction point, it is considered to be a fault[3].

3. *Phenomenonological Cause:* Whether the fault is caused by a human action or a "natural phenomena without human participation."

4. *Dimension:* Whether the fault affects the element's hardware or software.

5. *Objective:* Whether the fault is the result of a malicious act.

6. *Intent:* Whether or not the person who introduced the fault was aware of the impact of their choice. Note that this is separate from objective; this distinction allows the analyst to distinguish good-faith but incorrect choices from actively malicious ones.

[3]Note that problems with measured physical values would be errors rather than faults according to this definition. That is, if a thermometer's sensor can be damaged by too much heat, that would be considered an error because it occurs at an established interaction point.

7. *Capability:* Whether or not the fault was an accident or the result of incompetence. This allows the analyst to consider whether blame is appropriate for the action that caused the fault.

8. *Persistence:* Whether the fault's occurrence is permanent or transient.

These classes are combined into a 256-element possible-fault matrix, which is then reduced to 31 base fault classes [3]. As part of the creation of SAFE, we further reduced the fault classes by eliminating fault classes 7 and 8. The Capability class was eliminated because, as Leveson argues, blame should not be part of hazard analysis but is rather a legal question. She writes that "Blame is the enemy of safety. Focus should be on understanding how the system behavior as a whole contributed to the loss and not on who or what to blame for it." [30] The persistence fault class was eliminated because it does not address the source of the fault: it seems unreasonable to think that, given two fault descriptions which vary only in their persistence, an analyst will specify different detection or compensatory mechanisms.

The elimination of these classes left us with 15 base fault classes. To this list we added three more to address interaction problems, which address situations where two elements function correctly but work together in such a way that causes a successor danger. Leveson notes that these problems are increasingly common, writing that "In complex systems, accidents often result from interactions among components that are all satisfying their individual requirements, that is, they have not failed." The 18 fault classes used in SAFE are listed and described in Table 4.1. We note that this list can be extended, shortened, or modified as necessary by domain experts in order to tailor the set of faults to a particular area. This can be done in order to align the guidewords with standardization efforts, a concept introduced by Procter et al. [6].

We note that consideration of the final three fault classifications in Table 4.1 involves elements other than the one under analysis, which seemingly violates our claims of enabling local reasoning. These classifications, though, are crucial for detecting barriers to safe

No.	Guideword	Description
1	SW Bug	Mistakes made in software creation
2	Bad SW Design	Poor choices made in software creation
3	Compromised SW	Adversary tampers with software in development
4	Compromised HW	Adversary tampers with hardware in development
5	HW Bug	Mistakes made in hardware development
6	Bad HW Design	Poor choices made in hardware development
7	Production Defect	Production defects due to natural phenomena
8	Deterioration	Hardware fault at runtime due to degradation over time
9	Environment Damages HW	Hardware fault at runtime due to environment
10	Operator HW Mistake	Operator makes a mistake while interacting with hardware
11	Operator HW Wrong Choice	Operator makes a poor choice while using hardware
12	Adversary Accesses HW	Adversary tampers with hardware at runtime
13	Adversary Accesses SW	Adversary tampers with software at runtime
14	Operator SW Mistake	Operator makes a mistake while interacting with software
15	Operator SW Wrong Choice	Operator makes a poor choice while using software
16	Syntax Mismatch	Sender uses a different syntax than receiver
17	Rate Mismatch	Sender transmits at a different rate than receiver expected
18	Semantic Mismatch	Sender and receiver interpret same value differently

Table 4.1: *The 18 combined fault classes used in SAFE*

composability, i.e., both syntactic and semantic interoperability (see Section 2.1.4). And—except in strongly connected systems—analysis of these classifications will not require fully global reasoning. Rather, since they involve consideration only of an element's immediate neighbors, reasoning about them is bounded by a small constant in all but pathological systems.

4.1.4 Formality in Causation and Decomposition

As discussed previously, we have been careful to—where possible—align SAFE with STPA so that the two can be used together on different parts of the same system. When, though, should an analyst use SAFE instead of STPA? This is to some degree a judgement for the

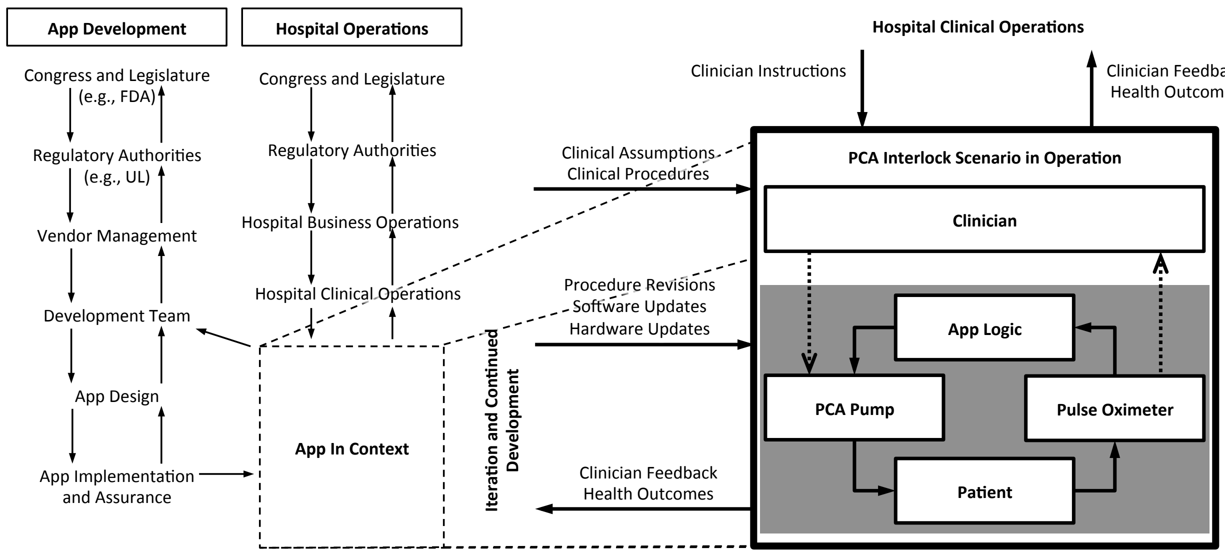


Figure 4.4: *The PCA interlock loop in the clinical context, adapted from figure 4.4 of Leveson [30]. SAFE would apply only to the shaded region.*

analyst herself to make, though we believe SAFE should be used when both a) a systems theoretic model of causality begins to hinder analysis more than it helps, and b) components can accurately decomposed/refined into a collection of cooperating automata. A view of the PCA Interlock scenario in the full clinical context is shown in Figure 4.4; we imagine that SAFE would be most useful in the shaded region. For absolute clarity, this section discusses research in the areas of causality and decomposition, but we believe that the direct use of this research will be necessary only in very rare circumstances.

Causality

Leveson writes that "The definition of accident causation needs to be expanded beyond failure events so that it includes component interaction accidents and indirect or systemic causal mechanisms." [30] We agree that both component interaction and indirect causes are important to consider, but STPA's literature provides no clear definition of causality. In STPA's hierarchical control structures (like Figure 4.4), what do the arrows between components mean, precisely? Do arrows between two software- or hardware-based elements signify something different than those between social constructs like organizations?

At all levels of a system's sociotechnical hierarchy, arrows between components intuitively read as communication, or more precisely a causal relationship between observable actions of the sender and the behavior of the receiver. Can we be more precise about the semantics? Similarly, what is meant by the lack of arrows between two elements? It would be incorrect to state that the app logic does *not* affect the patient, but there is no arrow connecting the two. We believe that the semantics of arrows between software- and hardware-based elements of a system's control structure diagrams should be those of *actual, intransitive* causality.

Actual Causality We agree with Leitner-Fischer, who writes that actual cause, as proposed by Halpern and Pearl, is the correct model of causality for reasoning about safety-critical systems [89, 83]. Leitner-Fischer explains that actual cause theory is a refinement of, and addresses problems with, the popular *counterfactual* (or *alternative world*) style of reasoning, which he explains as:

> The "naive" counterfactual causality criterion according to Lewis is as follows: event A is causal for the occurrence of event B if and only if, were A not to happen, B would not occur. The testing of this condition hinges upon the availability of alternative worlds. A causality can be inferred if there is a world in which A and B occur, whereas in an alternative world neither A nor B occurs.

Unfortunately, Leitner-Fischer explains that there are a number of problems with this naive approach, including: a) the failure to identify conjunctions or disjunctions of events as causal, b) preemption of one event by another (and event ordering in general, a major focus of [89]), as well as c) event non-occurrence and irrelevance[4]. In order to address these problems, the counterfactual model was extended by Halpern and Pearl to what they term

[4]We note that these are similar to many of the problems with existing hazard analyses that drove Leveson to adopt systems theory.

the *structural equation model* (SEM)[5]. The SEM presents situations as "logical combinations of events as well as a distinction [between] relevant and irrelevant causes." [89]

Halpern and Pearl specify a formalized, three-part test to determine if some event is an actual cause of another event. This test is quite detailed, and not fully germane to this work, so we do not reproduce it here. We recommend, however, the interested reader to both Halpern and Pearl and Leitner-Fischer's works for a full description and formalization of actual causality [83, 89].

Intransitivity At first, using the alternate-world semantics of actual causality seems to lead to a strongly-connected control structure, since each element in the control structure clearly affects each other element. That is, in an alternate world where there is no app logic, the patient may experience a PCA overdose, so why is there no arrow directly connecting the app and the patient? Indeed, every square in the shaded region Figure 4.4 would have to have an arrow pointing to every other square, which is unsatisfactory and confusing.

We believe a reasonable solution to this problem is *intransitive noninterference*, which comes from the information flow community [91]. Rushby explains that "The idea of noninterference is really rather simple: a security domain u is noninterfering with domain v if no action performed by u can influence subsequent outputs seen by v." [92]. Rephrased into the domain of elements, communication, and observability, then, we might say that "an element e is noninterfering with another element f if no action performed by e can influence subsequent outputs seen by f."

We believe that the existence of an arrow in a system's control structure should signify intransitive observability. If one element is reachable from another (i.e., some path exists between them) that would signify (exclusively) transitive observability. That is, interaction between two components that are not directly connected would necessarily be mediated by all intermediary components. This aligns with our intuition: the only information the pump

[5]We note that the SEM bears some resemblance to the formulae created by Thomas in Section 4.3 of [90].

receives from the sensors in Figure 4.4 would be via commands generated by the app based on the sensors output. This notion of intransitivity of interference/observability becomes especially important when we consider the propagation of errors between components, a topic addressed in some depth in Section 5.5. We note that the arrows in STPA's diagrams model *interactions*—primarily information or command flow, but also occasionally physical phenomena. Causality can often be derived from these interactions.

Decomposition and Refinement

The entire PCA Interlock scenario will likely be thought of as one element by the hospital management, and it is possible that another implementation of the same scenario could satisfy the same goals. That is, a number of systems could suffice for the objectives of the shaded region of Figure 4.4. They would share the same inputs and outputs but could be composed of different devices and application logic. Indeed, this interchangeability is at the core of the component-based MAP vision. How, though, can we know the refinement from one element to a set of subelements is sound? We aim to answer this question with the firm notion of decomposition and refinement from Shankar's *Lazy Compositional Verification* [84].

The hierarchical safety structures used in both STPA and SAFE (i.e., Figure 4.4) can be thought of as models of decomposition and refinement. Decomposition is a term for breaking a system down into its component parts, while refinement is a related term that signifies increasing the level of specificity of a model into a more detailed specification, sometimes down to the implementation level. A good example of decomposition is moving down an abstraction hierarchy, i.e., in the language used to describe MAP apps from Section 3.3, from the AADL `System` level to the AADL `Process` level. Similarly, the code generation aspects of the MDCF Architect (from Section 3.4) is a good example of refining a component's architecture down to the implementation level.

Careful thought has been given to these notions in the formal methods community and

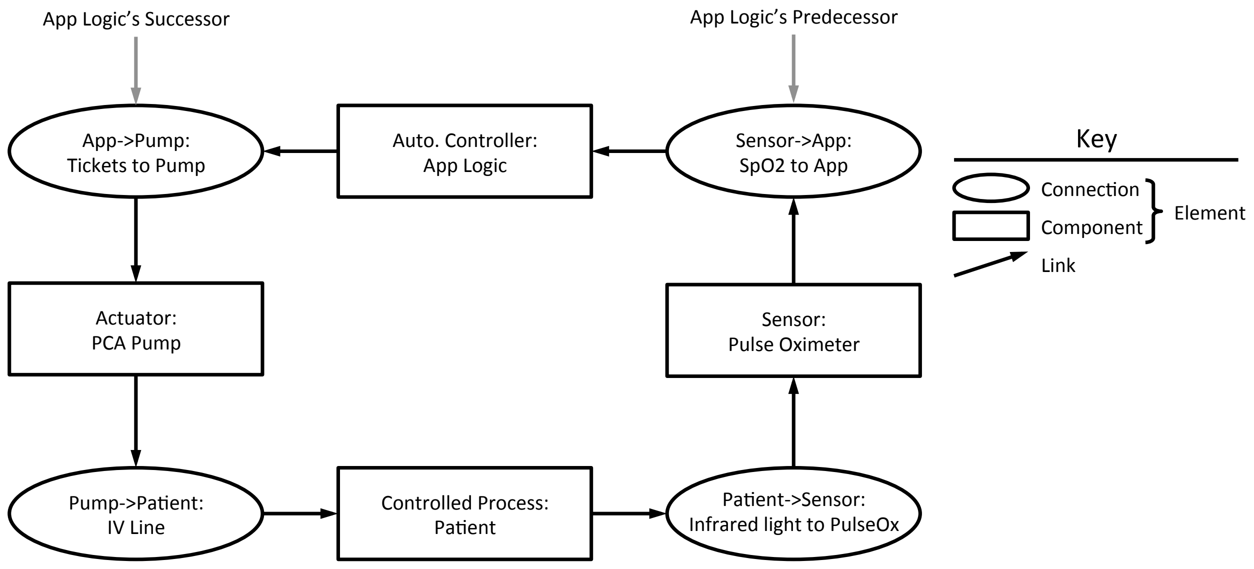


Figure 4.5: *An expanded view of the shaded region from Figure 4.4, showing components, connections, and the links between the two*

we believe that similarly rigorous notions should be used in SAFE as well. Using such formalisms, we could know if the shaded region in Figure 4.4 could be safely replaced by a different process that claims to achieve the same goals, i.e., could uphold the same invariants. This is a topic we explore in some depth in Section 5.4.

4.1.5 Terminology

In order to discuss subtle concepts clearly, we have assigned specific meanings to several terms that are commonly used in component-based engineering.

Components, Connections, and Links

One of the key realizations from Wallace's FPTC (described in Section 2.2.3) is that connections between elements can modify the failure behavior of a system in a similar way to components [35]. Dolev and Yao also explain the same realization, writing that connections—or adversaries with access to connections—can drop, duplicate or resend, or modify the payloads of messages [93]. Thus, safety analyses should treat them as first-class citizens, and not focus entirely on component analysis. The terminology of SAFE reflects this importance.

Figure 4.5 shows the shaded region from Figure 4.4 after it has been expanded to explicitly show both components and connections. Note that we use the term *element* to refer to a generic component or connection. Additionally, note that the arrows between elements are referred to as "Links;" links are infallible, directed bonds between elements of different types. That is, links cannot transform, produce, or consume errors; they can only propagate them. Additionally, SAFE does not allow one component to be linked directly to another component (or a connection linked directly to another connection): there must be a mediating element of the opposite type.

Predecessors and Successors

We often consider an element in the context of its immediate neighbors, i.e., those elements it is connected to via link. We phrase the relationship between an element and such neighbors in terms of "predecessors" and "successors." That is, in Figure 4.5 the connections from the sensors are the app logic's predecessors because they carry the messages it uses as input, while the connection to the PCA pump is the successor to the app logic since the logic generates the messages it carries.

Activities, Steps, and Tasks

We term the basic unit of the SAFE process a *task*, which is a single part of the process that cannot be reasonably divided further. Tasks combine to make up *step*s, which are ordered collections of tasks that accomplish some specific goal. Tasks are then ordered and collected into three top-level *activities*, which are intended to completely address either the system itself (Activity 0), all of an element's interactions with its predecessors (Activity 1), or all of an element's possible internal faults (Activity 2).

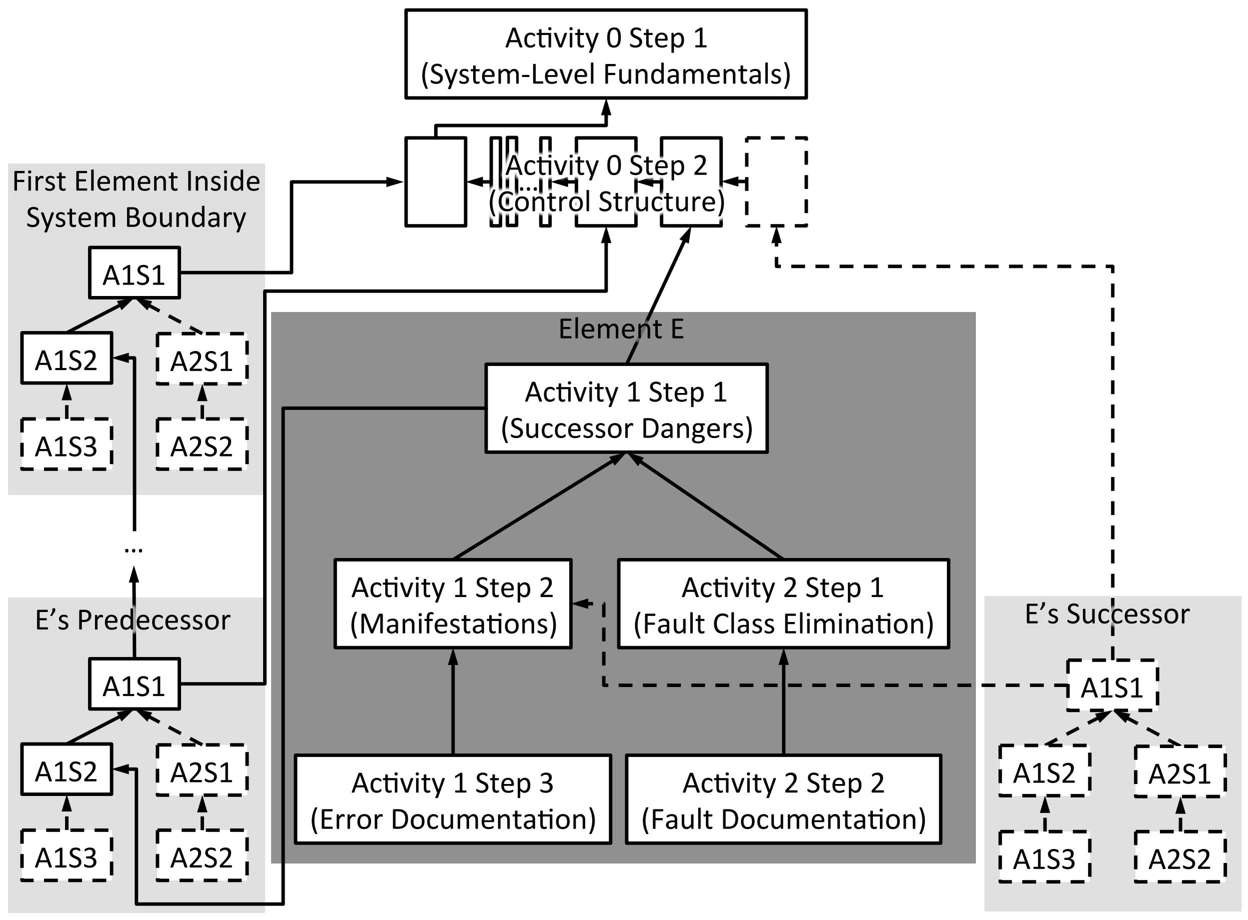


Figure 4.6: *The dependencies between the steps in SAFE. To completely analyze some element E (darkly shaded, center), all steps (white rectangles) with solid borders must be completed in the topological ordering implied by the dependencies (arrows). Steps or dependencies represented as dashed lines can be done at any time, but are not necessary for the analysis of E.*

4.1.6 Parallel and Compositional Aspects of SAFE

The SAFE process begins either by analyzing a system's fundamental aspects (notions of loss, hazards, etc.) or by importing those aspects from an STPA of the system's sociotechnical elements. After that step (Activity 0, Step 1) is completed, the next step is creating the system's control structure (Activity 0, Step 2). Defining this control structure consists of specifying element interactions and element-level fundamentals in a backwards chain from

Step	Enables
A0S1	A0S2 on first element inside system boundary
A0S2	A1S1 (given A1S2 on E's predecessor)
A1S1	A1S2, A2S1
A1S2	A1S2, A1S1 on E's successor
A1S3	—
A2S1	A2S2
A2S2	—

Table 4.2: *Dependencies between the steps of SAFE. ANSM signifies "Activity N, Step M," and all steps (except Activity 0's first step) are assumed to be performed on some element E.*

the element closest[6] to the controlled process. Each element's successors must all be defined before the element itself is specified.

Unlike Activity 0, Activities 1 and 2 focus on individual system elements. In order to begin analysis of a particular element E, an analyst will need to have a) performed Activity 0's second step on E (i.e., created a control structure that includes E), and b) performed Activity 1's first and second steps (identifying successor dangers and manifestations) on E's predecessor. Once Activity 1's first step is complete for E, the analyst can perform Activity 1's second step or Activity 2's first step on E. Performing Activity 1's second step enables Activity 1's first step on E's successor as well as Activity 1's third step on E. Activity 2 is largely standalone: its first step enables only its second step. Note that as the first element inside the system boundary has no predecessor, it instead relies on system-level fundamentals, which were established in Activity 0's first step.

Figure 4.6 shows a graphical representation of this process, and Table 4.2 shows a tabular representation. In the figure, the required steps to fully analyze element E are depicted as solid-bordered squares, and the solid arrows signify a "depends-on" relationship that must be met. Optional steps and dependencies, which can be performed/met at any time or in

[6]Closest in a topologically sorted version of the system's architecture.

parallel to the required ones, are depicted as dashed squares and arrows. This flexibility in ordering allows an analyst to focus on whatever aspects of the system she prefers. Additionally, though the first two steps of each component's analysis are sequential, the rest are fully parallelizable, and so additional analysts can be cooperate to decrease the time required for a full system analysis.

4.2 Activity 0: Fundamentals

The first activity in SAFE is centered around defining certain "fundamental" characteristics of a system. These include the both the notions and criticality levels of possible losses, known as *accidents* and *accident levels*, respectively. These losses are typically safety related (i.e., death or injury to humans, damage to the environment, etc.) but could, in theory, be any undesirable occurrence. Leveson terms the events that lead to accidents *hazards*, and rather than define them as a state of the system, she explicitly acknowledges that the environment must also be in a particular state for the loss to occur. Thus, hazards in SAFE are pairs of system and environment states. Lastly, the analyst considers *system-level safety constraints*, which are properties that—if maintained—would prevent the hazards from occurring.

As the analyst works through the system's accidents, hazards, and constraints she is likely considering, and should document, a candidate *control structure* for the system. Creating such a structure involves two related but subtly different tasks:

- *Determining the top-level system decomposition:* This requires assigning tasks to the elements that collectively comprise a system. Typically, an analyst/designer first allocates sensory functions, control algorithms, and actuation to their respective components and then connects the components to one another via communication channels. This decomposition can be formally shown to uphold the same invariants as the top-level system, if necessary (see Section 5.4).

- *Determining the top-level system boundary:* While the analyst allocates system tasks

Fundamental	Narrative Explanation
Accident Level	Death or serious injury
Accident	Patient harmed by too much analgesic from the PCA pump
Hazard	System Element: PCA Pump
	System State: Administering analgesic
	Environmental Element: Patient
	Environment State: Cannot tolerate more analgesic
	Hazardous Factor: Opioid analgesic
Safety Constraint	Disable pump when patient is at risk of overdose

Table 4.3: *Example safety* fundamentals *for the PCA Interlock scenario. Note that these are specific to the notion of opioid overdose; other undesirable events would have other fundamentals.*

to components, she also must consider what tasks are handled by elements in the environment. For the purposes of SAFE, environmental elements are those outside of the analyst's direct control. Environmental elements can either provide information to the system (monitored variables in [94]) or be controlled by the system (controlled variables, ibid.).

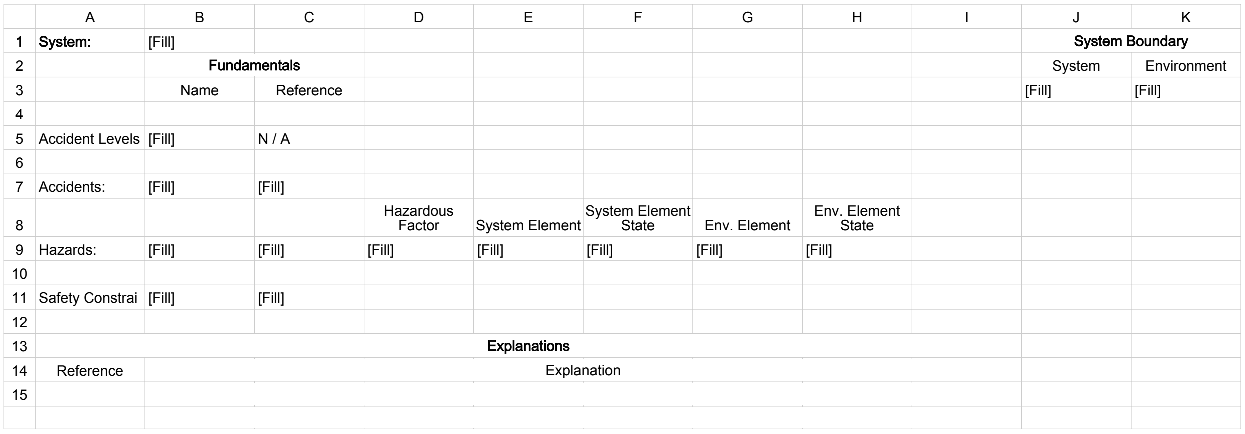
We use the term *fundamentals* to collectively refer to the system-level accident levels, accidents, hazards, and safety constraints as well as a specification of the control structure. An example of these textual fundamentals for the PCA Interlock Scenario is given in Table 4.3, while the shaded portion of Figure 4.4 is a partial example of the PCA interlock's control structure. A more complete example is given in Figure 4.12 and discussed in Section 4.2.2.

4.2.1 System-Level Fundamentals

SAFE's first step consists of documenting system-level fundamentals. These properties apply to the entire system, and are necessarily considered and documented before individual components are analyzed. This step aligns very closely with STPA: if an analyst has already

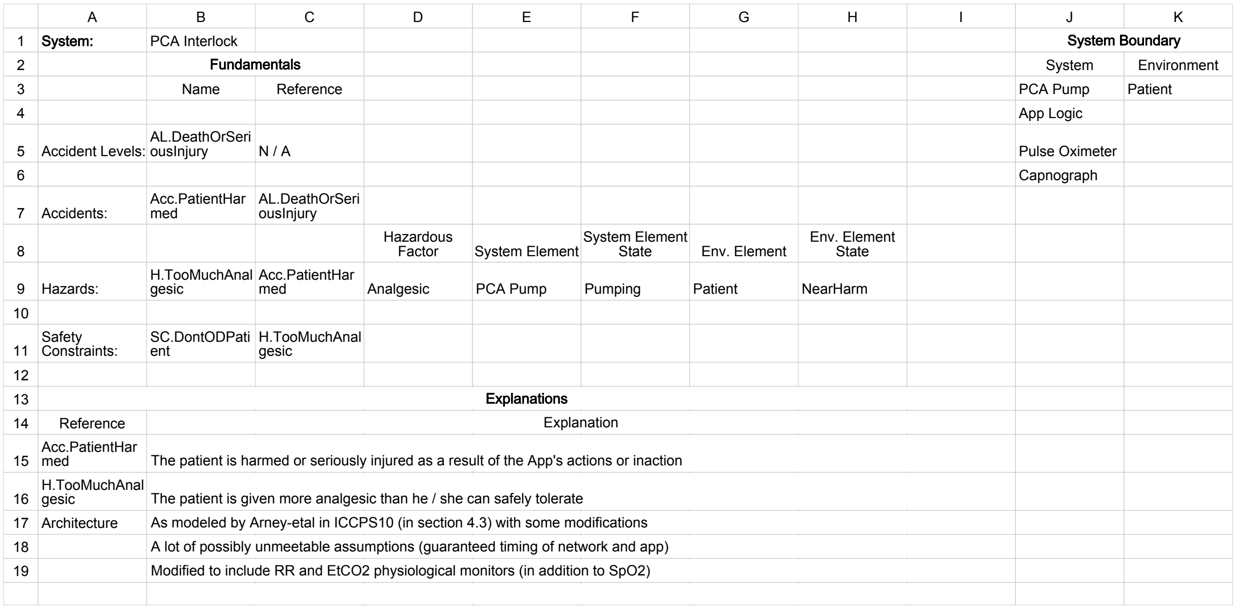
```
property set MAP_Error_Properties is

-- Other properties removed for space

Accident_Level : type record (
  Name : aadlstring;
  Description: aadlstring;
  Explanations : list of aadlstring;
  Accidents : list of MAP_Error_Properties::Accident;
);

Accident : type record (
  Name : aadlstring;
  Description : aadlstring;
  Explanations : list of aadlstring;
  Hazards : list of MAP_Error_Properties::Hazard;
);

Hazard: type record (
  Name : aadlstring;
  Description : aadlstring;
  HazardousFactor : aadlstring;
  SystemElement : reference (device, process);
  EnvironmentElement : reference (abstract);
  Explanations : list of aadlstring;
  Constraints : list of MAP_Error_Properties::Constraint;
);

Constraint: type record (
  Name : aadlstring;
  Description : aadlstring;
  ErrorType : reference({emv2}** error type);
  Explanations : list of aadlstring;
);

Fundamentals : record (
  Fundamentals : list of MAP_Error_Properties::Accident_Level;
  Explanations : list of aadlstring;
) applies to (abstract);

end MAP_Error_Properties;
```

Figure 4.7: *The `fundamentals` property type definition*



	A	B	C	D	E	F	G	H	I	J	K
1	**System:**	[Fill]								**System Boundary**	
2		**Fundamentals**								System	Environment
3		Name	Reference							[Fill]	[Fill]
4											
5	Accident Levels	[Fill]	N / A								
6											
7	Accidents:	[Fill]	[Fill]								
8				Hazardous Factor	System Element	System Element State	Env. Element	Env. Element State			
9	Hazards:	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]			
10											
11	Safety Constrai	[Fill]	[Fill]								
12											
13	**Explanations**										
14	Reference	Explanation									
15											

Figure 4.8: *The system-level fundamentals worksheet used in M-SAFE*



	A	B	C	D	E	F	G	H	I	J	K
1	**System:**	PCA Interlock								**System Boundary**	
2		**Fundamentals**								System	Environment
3		Name	Reference							PCA Pump	Patient
4										App Logic	
5	Accident Levels:	AL.DeathOrSeriousInjury	N / A							Pulse Oximeter	
6										Capnograph	
7	Accidents:	Acc.PatientHarmed	AL.DeathOrSeriousInjury								
8				Hazardous Factor	System Element	System Element State	Env. Element	Env. Element State			
9	Hazards:	H.TooMuchAnalgesic	Acc.PatientHarmed	Analgesic	PCA Pump	Pumping	Patient	NearHarm			
10											
11	Safety Constraints:	SC.DontODPatient	H.TooMuchAnalgesic								
12											
13	**Explanations**										
14	Reference	Explanation									
15	Acc.PatientHarmed	The patient is harmed or seriously injured as a result of the App's actions or inaction									
16	H.TooMuchAnalgesic	The patient is given more analgesic than he / she can safely tolerate									
17	Architecture	As modeled by Arney-etal in ICCPS10 (in section 4.3) with some modifications									
18		A lot of possibly unmeetable assumptions (guaranteed timing of network and app)									
19		Modified to include RR and EtCO2 physiological monitors (in addition to SpO2)									

Figure 4.9: *The system-level fundamentals of the PCA Interlock scenario from Table 4.3, specified using the worksheet from Figure 4.8*

```
package PCA_Shutoff
public

  system PCA_Shutoff_System
    -- Type specification removed for space
  end PCA_Shutoff_System;

  system implementation PCA_Shutoff_System.imp
  -- Implementation removed for space
  annex EMV2 {**
    use types PCA_Shutoff_Errors;
    properties
      MAP_Error_Properties::Fundamentals => [
        AccidentLevels => ([
          Name => "DeathOrSeriousInjury";
          Description => "Death or serious injury to a human";
          Accidents => ([
            Name => "PatientHarmed";
            Description => "Patient dies";
            Explanations => ([ "The patient is harmed..." ]);
            Hazards => ([
              Name => "TooMuchAnalgesic";
              Description => "The patient receives more analgesic than they
  ↪  can tolerate";
              HazardousFactor => "Analgesic";
              SystemElement => reference (pcaPump);
              EnvironmentElement => reference (patient);
              Explanations => ([ "The patient is given more analgesic
  ↪  than..." ]);
              Constraints => ([
                Name => "DontODPatient";
                Description => "The pump shouldn't run if the patient shows
  ↪  signs of an overdose";
                ErrorType => reference(TooMuchAnalgesic);
              ]);
            ]);
          ]);
        ]);
      ];
          -- Top-level explanations removed for space
  **};
  end PCA_Shutoff_System.imp;

end PCA_Shutoff;
```

Figure 4.10: *The fundamentals of the PCA Interlock scenario from Table 4.3, specified using the* `fundamentals` *property type from Figure 4.7*

```
package PCA_Shutoff_Patient
public
    with MAP_Properties, PCA_Shutoff_Types;
    abstract PCA_Shutoff_Patient
        features
            vein: in feature;
            fingerclip: out feature;
            exhalation: out feature;
        properties
            MAP_Properties::Component_Type => controlled_process;
    end PCA_Shutoff_Patient;

    abstract implementation PCA_Shutoff_Patient.imp
        subcomponents
            health_status: data PCA_Shutoff_Types::PumpCmd;
    end PCA_Shutoff_Patient.imp;

end PCA_Shutoff_Patient;
```

Figure 4.11: *The PCA Interlock Scenario's patient in AADL*

begun the STPA process she can likely carry much of this information over to SAFE without modification. The exact format of this documentation depends on whether the analyst is using M-SAFE or T-SAFE: M-SAFE's system-level worksheet is shown in Figure 4.8, T-SAFE's `fundamentals` property type definition is shown in Figure 4.10.

Identifying System Components

The first tasks in SAFE are straightforward: they consist of identifying the system and its top-level decomposition. If the analyst is examining an extant system, then she can simply write down the human-readable names of the system itself and its components. If, however, the system is still being designed then this task can be slightly more complex due to the two-task process of determining the system composition and boundaries (see the introduction to this section).

In M-SAFE, these names are filled in the provided spreadsheet cells: see cells B1 and J3-K3 of Figures 4.8 and 4.9. In T-SAFE, this step consists of creating and naming the various components that make up the system as discussed in Section 3.3. Note that we have extended the language slightly, however, to include the use of `abstract` components and `feature` connections, which are used to model components and connections for which no code-generation is required. An example, which models the patient in the PCA Interlock scenario, is shown in Figure 4.11. Use of these language constructs enables hazard analysis annotations required by T-SAFE to be used without affecting the code generation aspects of the MDCF Architect.

Accident Levels

Next, the analyst should consider what the worst-case result of the scenario going wrong would be, i.e., the application's *accident level*. This can be an application specific notion, or an analyst can choose to align with standardized levels of criticality. Example standards-based notions include IEC 62304's "Software Safety Classifications" and ISO 14971's qualitative severity levels (discussed in Section 2.2.5). For the analysis of the PCA Interlock scenario, we have—somewhat arbitrarily—chosen to use the safety classifications from ISO 62304, and so we note that this application would be "Class C" as "Death or SERIOUS INJURY is possible."

The worst-case result of a failure in the PCA Interlock scenario is the death of, or serious harm to, the patient. Other MAP apps may involve other accident levels, such as damage to equipment or a waste of valuable resources. Multiple accident levels should be ordered in descending order of undesirability, e.g., human death would rank first, followed by human injury, followed by equipment damage. In M-SAFE, accident levels are written in a spreadsheet cell and prefixed with "AL" for identification purposes, as in cell B5 of Figure 4.9. In T-SAFE the analyst would document the accident level using the `Accident_Level` property from the `MAP_Error_Properties` property set. The property is defined in lines 5-10 of Figure 4.7, and the accident level associated with the PCA Interlock example is shown see lines 15-16 of Figure 4.10.

Accidents

The next action required by SAFE is to document the actual losses that can occur, these are termed *accidents*. Each accident should cause harm at a level specified by a given accident level, and be traceable to that accident level. In the PCA Interlock scenario, we have primarily focused on the accident of opioid overdose: i.e., if our application fails to disable the pump when it should, the patient could be seriously injured or killed by an overdose of analgesic. Alternatively, an analyst may want to also avoid the underinfusion of a patient: this would be a separate accident and would trace to a second, lower-ranked accident level.

In M-SAFE, accidents are documented by name in the a spreadsheet cell and prefixed with "Acc." They should also, in a reference cell, have a link to the accident level that the accident would cause. Cells B7 and C7 of Figure 4.9 show the accident name and accident level reference, respectively. T-SAFE allows multiple accidents to be embedded as sub-properties of the accident level they are linked to. The `Accident` property type is specified in lines 12-17 of Figure 4.7, and the PCA Interlock example is shown in line 17-20 of Figure 4.10. This structure allows a developer/analyst to directly connect a collection of accidents to the same accident level.

Hazards

After identifying system-level accidents, the analyst should next consider the system-level hazards that would cause them. Recall that hazards are pairings of one system state and one worst-case environment state: we ask the analyst to consider both system and environmental elements as well as their states. Additionally, if possible, analysts should identify the *hazardous factor*, which is a concept advocated by Ericson[7]. Ericson defines the term as: "...the basic hazardous resource creating the impetus for the hazard, such as a hazardous energy source...being used in the system."

[7]Ericson uses the term *hazardous element*, but since element has a specific meaning in SAFE (component or connection) we have renamed the term.

In the PCA Interlock scenario, our primary hazard then consists of the elements identified in Table 4.3. That is: the hazard name, a reference to the caused accident, the hazardous factor, system element, system element's state, environment element, and environment element's worst-case state are all recorded together. In M-SAFE, the information is entered in row 9, columns B-I in Figure 4.8. The worksheet complete with information regarding the PCA Interlock is shown in the same cells in Figure 4.9. T-SAFE follows a similar pattern as with accidents and accident levels: the hazard property is specified in lines 19-27 of Figure 4.7, and an example of the hazard embedded as a subproperty of the accident it would cause is shown in lines 21-27 of Figure 4.10. The current version of T-SAFE does not allow identification of system or environment element states, but only the elements themselves. The relevant states of the elements are derivable, though, so analyst-specified state information is not required.

Safety Constraints

The specification of safety constraints is central to STPA, and is similarly important in SAFE. Leveson writes that "The most basic concept in [the accident model underlying STPA] is not an event, but a constraint. Events leading to losses occur only because safety constraints were not successfully enforced." [30] Safety constraints, like the one specified in Table 4.3, are associated with a hazard and—if enforced—guarantee the avoidance of the associated hazard. Often they take the form of requiring the system to be in a specific state when the environment/controlled process is in a particular state.

In M-SAFE, safety constraints are documented much like the other fundamentals: with a name, prefixed by "SC.," and a reference to the associated hazard. See cells B11 and C11 of Figures 4.8 and 4.9. In T-SAFE, safety constraints are embedded as subproperties of the hazard they are associated with; see lines 29-34 of Figure 4.7 and 28-31 of Figure 4.10.

Explanations

While we recognize that one of the strengths of STPA is its flexibility, we also believe that too much flexibility in a documentation format can lead to variance in analysis quality. We have attempted to strike a balance between an overly-rigid format and one that would allow too much variability, and that balance comes in free-form, narrative explanations that can be attached to any fundamental construct. These are designed to be short (sentence-length) justifications or elaborations that clarify unintuitive accidents, hazards, constraints, or architectural decisions. That is, these explanations are not intended to be used for paragraph (or longer) narratives.

In M-SAFE, these explanations are written in rows 15+ of columns B-I; column A is used for a reference to the accident level, accident, hazard, or safety constraint. Figure 4.9 shows some sample explanations for the PCA Interlock scenario, including fuller explanations of the accident and hazard as well as some notes about the architecture of the system itself. In T-SAFE these explanations are added using the optional `Explanations` subproperty of each fundamental property type or the top-level `Fundamentals` property for explanations not bound to any particular accident level, accident, hazard, or safety constraint. For example, see lines 20 and 27 of Figure 4.10.

Candidate Graphical Control Structure

The final task in Activity 0's first step is to determine a candidate control structure. We note that this task is both optional (but recommended) and specific to M-SAFE. Additionally, if the analysis is being performed on a system that has already been built then this step is straightforward, and the structure is unlikely to change much. If, however, the analysis is being performed as the system is built, then the control structure will likely be modified as the design and safety analysis proceed—thus, the output of this step should be considered more of a sketch than a final version.

Control structures in M-SAFE are required to be specified textually (see Step 2, below)

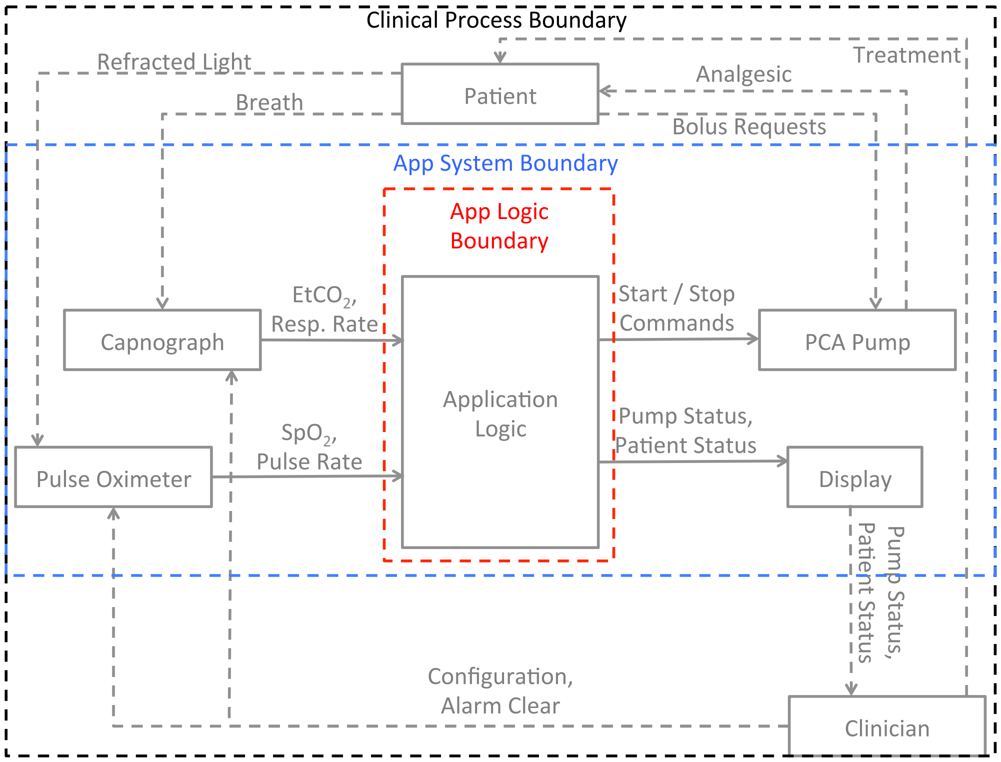


Figure 4.12: *Possible system boundaries for different levels of abstraction in the PCA Interlock scenario*

and optionally graphically as well. Graphical representations are typically much easier to use, but play no role in the rest of the analysis; the textual representation is used to guide the order in which all remaining tasks are performed. Control structures are determined programmatically in T-SAFE, so this step has no equivalent in the tool-supported process.

4.2.2 Specifying a Control Structure

Activity 0, Step 2 involves specifying the candidate control structure in an actionable format. This sets up the actual analysis to be performed in activities 1 and 2.

Drawing System Boundaries

An analyst begins by selecting the element closest to the controlled process that is inside the system boundary. Which elements, though, should be considered "inside" the system boundary? Recall from the introduction to Activity 0, where we explained that elements which are directly under the analyst/developer's control should be considered inside the system, while those that are not should be considered to be in the environment. Note, however, that the boundary will, expand/shrink as analysts move up/down a system's abstraction hierarchy.

Consider Figure 4.12, which overlays three boundaries onto the app-developer's view of the PCA Interlock scenario from Figure 2.3. The innermost boundary, in red, is that of the app logic developer: he gets input from, and sends output to, the devices, but he has no control over what those devices are. The middle boundary, in blue, is that of the app designer herself: by specifying the properties required of the devices, she exercises direct control over which ones get used and which do not. She does not, though, control the patient or the clinician directly, so they are still considered environmental. Finally, the outermost boundary, in black, includes all elements of the clinical process. Other boundaries are possible, including those at both lower and higher levels of abstraction. Additionally, boundaries are not necessarily concentric: the PCA Pump creator's boundaries exist at the same level of abstraction as the app logic developer's, but they do not overlap.

Identifying Elements

The next task is to uniquely identify the elements by their place in the system's control structure. When specifying a control structure, analysts should typically start at the controlled process and work backwards, at each element asking "what element(s) affects this current one?" For example, in the PCA Interlock scenario, an analyst would begin by considering the patient: what component or connection affects the patient? The IV line. What affects the IV line? The PCA pump. This process would continue iterating backwards through the control structure until all the elements identified previously have been allocated. When in doubt, analysts should recall the two-part notion of causality discussed in Section 4.1.4: for some component A to control some component B, A should have an *intransitive, actual* effect on B. In fact, it as at this level that we believe system causality becomes more of a burden than a benefit: when hardware- and software-based components are directly interacting, non-traditional models of causality can become quite unintuitive.

In Figure 4.12, the first element inside the app system boundary—when working backwards from the patient—is the PCA Pump, so the analyst should start here. For M-SAFE this requires creating a copy of the element worksheet shown in Figure 4.15 for the pump. Then, the element is named and the names of predecessor and successor links are specified using row 4 of the worksheet.

In T-SAFE this step is automatically performed by specifying the app layout in the language subset from Chapter 3. Both the `System` and the `Process` constructs can be thought of as defining boundaries and decompositions, since they include an external interface (the type) and the internal decomposition (the implementation). `Systems` are used to specify the app's overall component and connection topology, equivalent to the blue boundary in Figure 4.12. The `Process` construct is one level lower in the abstraction hierarchy, and is equivalent to the red boundary in Figure 4.12. See Sections 3.3.2 and 3.3.3 for details.

Specifying Classifications

In the hierarchical control structures used by STPA, components are typically assigned roles, e.g., controller, actuator, sensor or controlled process. We believe that this notion is useful, but SAFE generalizes the concept somewhat. We name these roles *architectural classifications*, but we expect that future work may come up with other, possibly domain-specific, classifications as well. These classifications may be used to guide tasks in activities 1 and 2, and we believe that they may be tailored by domain experts to if domain specificity would be useful. The final task in Activity 0 is to document the role played by a particular component, or in the case of a connection, the roles played by its sender and receiver. In M-SAFE a component's architectural classification is specified in cell 4H, while T-SAFE uses the `Component_Type` property to set the value (see line 10 of Figure 4.13).

Note that if a single component has multiple successor links and could reasonably be classified under multiple architectural classification, the analyst should "split" the component for the purpose of the SAFE analysis. That is, since the system is being modeled based on its control structure—rather than its physical deployment—what we consider to be a component or connection are technically logical components and connections. Typically these logical structures (e.g., app logic, PCA pump) align with physical realizations (e.g., executable code, a specific PCA pump) when a system is deployed, but in the cases where they do not, the logical structures should be modeled.

Backwards Iteration

Once a component's classification(s) have been specified, the analyst can either start identifying a new component by moving backwards one step in the control structure or proceed with analysis of a previously-identified element by beginning Activity 1. If the analyst moves to a new element, a new copy of the worksheet from Figure 4.15 should be created for each predecessor that has a link to the current element, and the analyst should return to the "Identifying Elements" task. This backwards iteration should continue until all elements inside the system boundary (as specified at the beginning of this step) have been identified. Note that any time after a component's predecessors have been fully identified and documented as part of Activity 0's second step, Activity 1's tasks can be performed on it. If multiple analysts are available, the two activities can be performed in parallel, as long as each component's predecessors are fully identified before its external interactions are analyzed.

4.3 Activity 1: Externally Caused Dangers

The first activity in SAFE is, like the second step of Activity 0, performed on each element individually. Activity 1 involves first discovering, and then documenting, a) a component's individual, local notion of danger (i.e., its successor dangers, see Section 4.1.1); b) whether or not external errors arriving via the element's predecessor links would manifest as those successor dangers (i.e., manifestations, see Section 4.1.2); and finally c) the relationship between the element's successor dangers and its manifestations. Documenting that relationship involves not only linking a manifestation to a successor danger, but also explaining the link and any possible compensatory actions.

These explanations are written primarily in a narrative format, but may also involve quasi-formal statements involving *process models*. Process models are a central part of STPA, and Leveson describes them as "a model of the process being controlled." [30] Process models are most relevant for components that calculate control actions based on sensor input though they can be useful for sensing and actuation components as well.

4.3.1 Successor Dangers and Process Models

The first step in Activity 1 is to determine an element's *successor dangers* and, if it is a component, document its process model.

Import the Element's Successor Dangers

Recall from Section 4.1.1 that these are outputs from our current component which would transitively cause system-level safety constraint violations. Typically this task is straightforward, as the manifestations of a component's successor are its successor dangers, and they can be directly copied in. Since the first element to be considered has no successor inside the system boundary, it uses violations of the system-level constraints as its successor dangers.

```
1  package PCAPump_Interface
2  public
3  with PCA_Shutoff_Types, MAP_Properties, MAP_Error_Properties,
     ↪  PCA_Shutoff_Patient;
4
5    device ICEpcaInterface
6    features
7      TktsIn:in event data port PCA_Shutoff_Types::Ticket;
8      DrugFlow:out feature;
9    properties
10     MAP_Properties::Component_Type => actuator;
11   annex EMV2 {**
12      use types MAP_Errors, PCA_Shutoff_Errors;
13      use behavior PCA_Shutoff_Errors::PumpStatus;
14     error propagations
15       TktsIn:in propagation {TktTooLong, TktTooShort, ErraticTkt, NoTkt,
     ↪  EarlyTkt, LateTkt};
16       DrugFlow:out propagation {TooMuchAnalgesic};
17     end propagations;
18   **};
19   end ICEpcaInterface;
```

Figure 4.13: *The PCA Pump interface type in AADL*

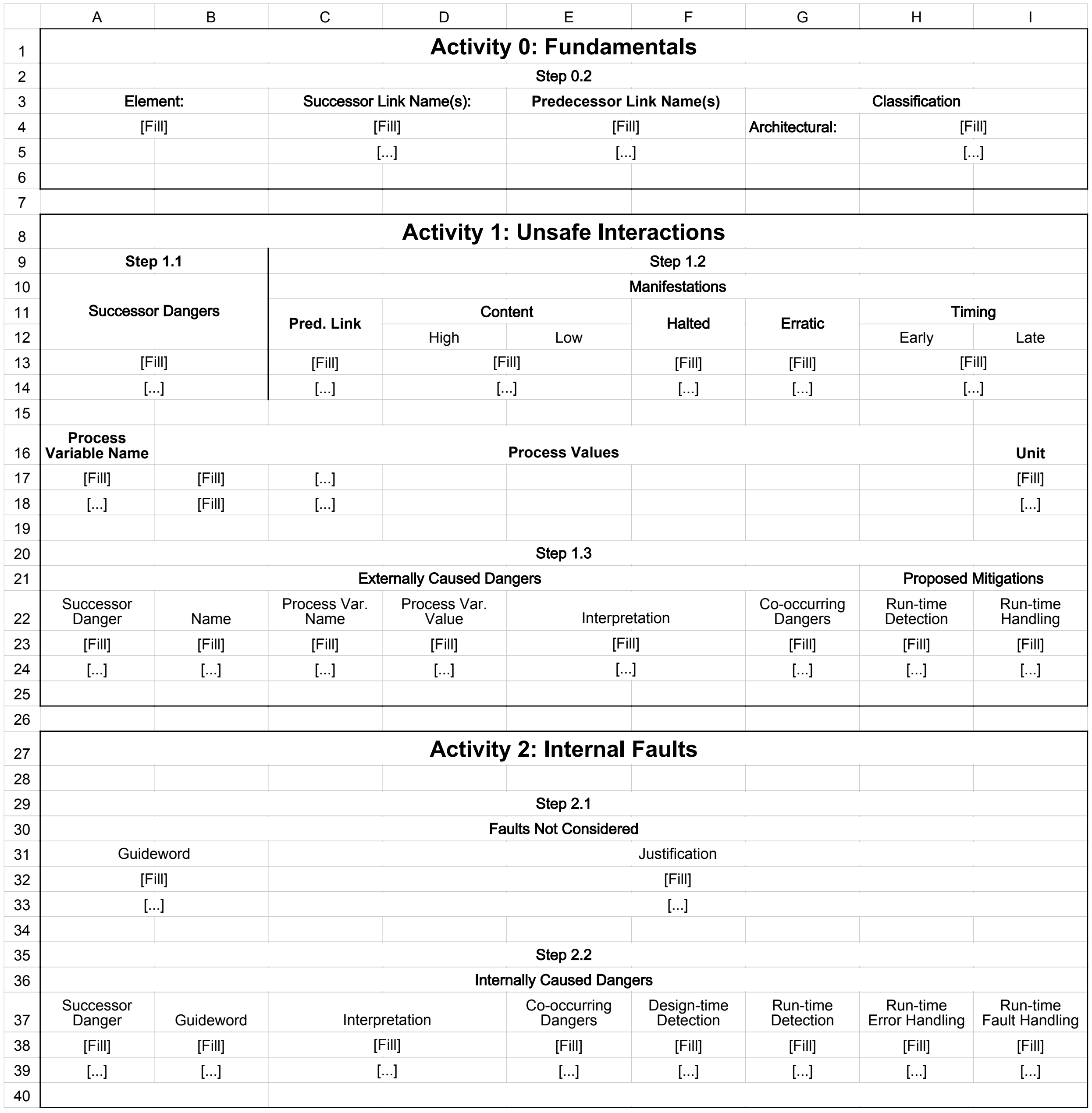
In the PCA Interlock example, the PCA pump's successor danger is violating the system-level safety constraint by administering narcotic when the patient is not healthy enough to tolerate it. In M-SAFE this would be documented by writing the constraint's name in cell A13, or by creating a reference to the associated constraint's cell. Figures 4.15 and 4.16 show an example of the blank worksheet and part of the full PCA pump analysis. In T-SAFE successor dangers are generated automatically from the error types that propagate out of a component. See line 16 of Figure 4.13, which shows the `TooMuchAnalgesic` error type being propagated out on the pump's `DrugFlow` port.

Document the Process Model

Leveson, citing Ashby, explains that controlling a process requires four conditions: a) a *goal* condition (what the controller is trying to achieve), b) an *action* condition (how the controller can affect the system's state), c) an *observability* condition (how the controller can sense the system's state), and d) a *model* condition, which is some anticipated relation

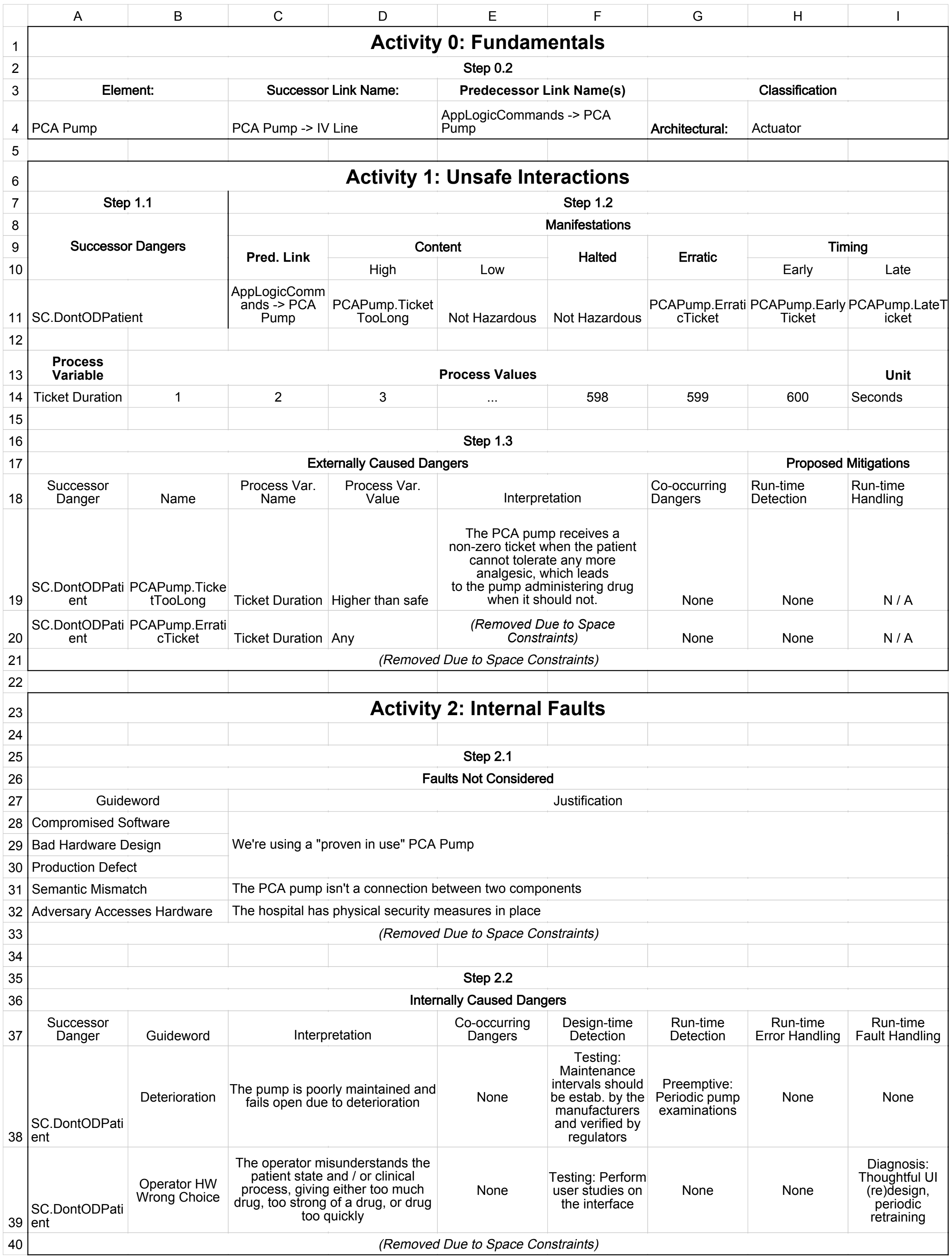
```
device implementation ICEpcaInterface.imp
  subcomponents
    Tkt:data PCA_Shutoff_Types::Ticket
    {MAP_Error_Properties::ProcessVariable => true;};
  annex EMV2 {**
    use types MAP_Errors, PCA_Shutoff_Errors;
    use behavior PCA_Shutoff_Errors::PumpStatus;
    error propagations
      flows
        -- Externally errors
        LongTktOD:error path TktsIn{LongTkt}->DrugFlow{TooMuchDrug};
        ErraticTktOD:error path TktsIn{ErraticTkt}->DrugFlow{TooMuchDrug};
        LowTktsSafe:error sink TktsIn{TktTooShort};
        NoTktsSafe:error sink TktsIn{NoTkt};
        -- Internal faults
        DeteriorationLeadsToOD:error source DrugFlow{TooMuchDrug} when
  ↪  {Deterioration};
        CosmicRayLeadsToOD:error source DrugFlow{TooMuchDrug} when
  ↪  {CosmicRay};
    end propagations;
    component error behavior
      events
        -- Detectable external problems
        TimeoutVio:error event {EarlyTkt};
        TimestampVio:error event {LateTkt};
        -- Detectable internal problems
        PumpDeteriorates:error event {Deterioration};
      transitions
        SwitchToKVO:Normal -[ TimeoutVio or TimestampVio ]-> PermanentKVO;
    end component;
    properties
      MAP_Error_Properties::RuntimeErrorDetection => [
        ErrorDetectionApproach => Concurrent;
        Explanation => "Minimum sep. between messages detect early arrivals";
      ] applies to TimeoutVio;
      MAP_Error_Properties::RuntimeErrorHandling => [
        ErrorHandlingApproach => Rollforward;
        Explanation => "The pump switches into a fail-safe mode";
      ] applies to SwitchToKVO;
      MAP_Error_Properties::ExternallyCausedDanger => [
        ProcessVariableValue => reference(Tkt);
        ProcessVariableConstraint => "Too high for patient's status";
        Explanation => "The ticket has a time value that is too long";
      ] applies to LongTktOD;
  **};
  end ICEpcaInterface.imp;
end PCAPump_Interface;
```

Figure 4.14: *The PCA Pump interface implementation in AADL*



	A	B	C	D	E	F	G	H	I
1	**Activity 0: Fundamentals**								
2	**Step 0.2**								
3	**Element:**		**Successor Link Name(s):**		**Predecessor Link Name(s)**		**Classification**		
4	[Fill]		[Fill]		[Fill]		**Architectural:**	[Fill]	
5			[...]		[...]			[...]	
6									
7									
8	**Activity 1: Unsafe Interactions**								
9	**Step 1.1**		**Step 1.2**						
10			**Manifestations**						
11	**Successor Dangers**		**Pred. Link**	**Content**		**Halted**	**Erratic**	**Timing**	
12				High	Low			Early	Late
13	[Fill]		[Fill]	[Fill]		[Fill]	[Fill]	[Fill]	
14	[...]		[...]	[...]		[...]	[...]	[...]	
15									
16	**Process Variable Name**	**Process Values**							**Unit**
17	[Fill]	[Fill]	[...]						[Fill]
18	[...]	[Fill]	[...]						[...]
19									
20	**Step 1.3**								
21	**Externally Caused Dangers**							**Proposed Mitigations**	
22	Successor Danger	Name	Process Var. Name	Process Var. Value	Interpretation		Co-occurring Dangers	Run-time Detection	Run-time Handling
23	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]		[Fill]	[Fill]	[Fill]
24	[...]	[...]	[...]	[...]	[...]		[...]	[...]	[...]
25									
26									
27	**Activity 2: Internal Faults**								
28									
29	**Step 2.1**								
30	**Faults Not Considered**								
31	Guideword		Justification						
32	[Fill]		[Fill]						
33	[...]		[...]						
34									
35	**Step 2.2**								
36	**Internally Caused Dangers**								
37	Successor Danger	Guideword	Interpretation		Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
38	[Fill]	[Fill]	[Fill]		[Fill]	[Fill]	[Fill]	[Fill]	[Fill]
39	[...]	[...]	[...]		[...]	[...]	[...]	[...]	[...]
40									

Figure 4.15: *The component worksheet used in M-SAFE.*



	A	B	C	D	E	F	G	H	I
1	**Activity 0: Fundamentals**								
2	**Step 0.2**								
3	**Element:**		**Successor Link Name:**		**Predecessor Link Name(s)**		**Classification**		
4	PCA Pump		PCA Pump -> IV Line		AppLogicCommands -> PCA Pump		**Architectural:**	Actuator	
5									
6	**Activity 1: Unsafe Interactions**								
7	**Step 1.1**		**Step 1.2**						
8			**Manifestations**						
9	**Successor Dangers**		**Pred. Link**	**Content**		**Halted**	**Erratic**	**Timing**	
10				High	Low			Early	Late
11	SC.DontODPatient		AppLogicCommands -> PCA Pump	PCAPump.TicketTooLong	Not Hazardous	Not Hazardous	PCAPump.ErraticTicket	PCAPump.EarlyTicket	PCAPump.LateTicket
12									
13	**Process Variable**	**Process Values**							**Unit**
14	Ticket Duration	1	2	3	...	598	599	600	Seconds
15									
16	**Step 1.3**								
17	**Externally Caused Dangers**							**Proposed Mitigations**	
18	Successor Danger	Name	Process Var. Name	Process Var. Value	Interpretation		Co-occurring Dangers	Run-time Detection	Run-time Handling
19	SC.DontODPatient	PCAPump.TicketTooLong	Ticket Duration	Higher than safe	The PCA pump receives a non-zero ticket when the patient cannot tolerate any more analgesic, which leads to the pump administering drug when it should not.		None	None	N / A
20	SC.DontODPatient	PCAPump.ErraticTicket	Ticket Duration	Any	*(Removed Due to Space Constraints)*		None	None	N / A
21	*(Removed Due to Space Constraints)*								
22									
23	**Activity 2: Internal Faults**								
24									
25	**Step 2.1**								
26	**Faults Not Considered**								
27	Guideword		Justification						
28	Compromised Software		We're using a "proven in use" PCA Pump						
29	Bad Hardware Design								
30	Production Defect								
31	Semantic Mismatch		The PCA pump isn't a connection between two components						
32	Adversary Accesses Hardware		The hospital has physical security measures in place						
33	*(Removed Due to Space Constraints)*								
34									
35	**Step 2.2**								
36	**Internally Caused Dangers**								
37	Successor Danger	Guideword	Interpretation		Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
38	SC.DontODPatient	Deterioration	The pump is poorly maintained and fails open due to deterioration		None	Testing: Maintenance intervals should be estab. by the manufacturers and verified by regulators	Preemptive: Periodic pump examinations	None	None
39	SC.DontODPatient	Operator HW Wrong Choice	The operator misunderstands the patient state and / or clinical process, giving either too much drug, too strong of a drug, or drug too quickly		None	Testing: Perform user studies on the interface	None	None	Diagnosis: Thoughtful UI (re)design, periodic retraining
40	*(Removed Due to Space Constraints)*								

Figure 4.16: *A partial M-SAFE component worksheet for the PCA Pump used in the PCA Interlock scenario.*

between the actions and their effects [30, 95]. This model condition can also be thought of as a combination of the natural relation between a system's monitored and controlled variables (which Parnas and Madey term the NAT relation) and the current observable state of the system (i.e., Parnas and Madey's vector $\underset{\sim}{m}^t$) [94].

As systems are decomposed into subsystems (an idea more fully explored in Section 5.4), it follows that what is considered an actuator at one level of abstraction may in fact be an entire subsystem—that contains a controller—at a lower level of abstraction. Thus, though traditional control theory speaks to process models existing in controlling elements (i.e., their architectural classification is "controller"), we recognize that it may sometimes be useful for actuators and sensors to have a process model as well. To that end, in SAFE, process models can be documented on any component, but are only considered required for controllers.

In the PCA Interlock scenario, for example, we can think of the PCA pump as having a model of the patient's health that has been simplified to the ticket value, and a goal to not damage the patient's health. If the value is zero, the patient is unhealthy and the pump should not run; if the ticket value is non-zero then the pump can run safely. In SAFE a process model is documented by naming its composite variables, listing their possible values, and—if possible—giving the variables' units. For example, the PCA pump's process model might be called "Ticket Duration," and range from 1 to 600 seconds.

In M-SAFE this would be written using rows 17 and 18 of the worksheet, see Figure 4.15. Figure 4.16 shows the PCA Pump's process model on line 14. In T-SAFE this is done by creating a datatype and tagging it with the `ProcessVariable` property, as in lines 3 and 4 of Figure 4.14. Then, the datatype definition should include the representation of the data (e.g., floating point, string, character, etc.), the unit, and—if the process variable is numeric—the range of acceptable values. An example is shown in Figure 4.17. Note that these datatype definitions have only been slightly modified from Figure 3.2 to include the `Measurement_Unit` subproperty. Other subproperties, such as the

```
package PCA_Shutoff_Types
public
with Data_Model;

  data SpO2
  properties
    Data_Model::Data_Representation => Float;
    Data_Model::Real_Range => 0.0 .. 100.0;
    Data_Model::Measurement_Unit => "Percent";
  end SpO2;

  data Ticket
  properties
    Data_Model::Data_Representation => Integer;
    Data_Model::Integer_Range => 0 .. 600;
    Data_Model::Measurement_Unit => "Seconds";
  end Ticket;

end PCA_Shutoff_Types;
```

Figure 4.17: *Datatypes used in the PCA Interlock's Process Model*

IEEE11073_Nomenclature::OID from Figure 3.2, can be added as necessary without affecting the T-SAFE report generation.

4.3.2 Deriving an Element's Dangers

The next step, which has only one task, is to consider how errors from previous components could cause the current element to exhibit its successor dangers. This is done by first selecting one of the component's predecessor links (which were identified in the second step of Activity 0) and then considering what would happen if the incoming messages were erroneous according to any of the six failure domains discussed in Section 4.1.2. This consideration will result in a judgement, by the analyst, of whether or not input failing according to the particular domain will *manifest* as a problem leading to a successor danger. This judgement will be that the combination of failure domain and input is either not dangerous, or a valid manifestation.

An input and failure domain pairing that are incorrect but not unsafe should be labelled

```
package MAP_Errors
public
annex EMv2
{**
error types

  Content : type;
  High : type extends Content;
  Low : type extends Content;

  Timing : type;
  Early : type extends Timing;
  Late : type extends Timing;

  Halted : type;
  Erratic : type;

end types;
**};
end MAP_Errors;
```

Figure 4.18: *The six failure domains from Avižienis et al. encoded in AADL's EMV2*

"not dangerous." Take for example the PCA Interlock scenario where the only hazard being considered is the overdose of the patient: if the tickets sent to the pump are too short—i.e., the patient could safely tolerate more analgesic—then there is no danger. The patient will be in unnecessary pain, but he will not be at risk of overdose. If, on the other hand, the pairing would lead the component to an unsafe state, then a valid manifestation has been found. For example, if the value of the incoming tickets is too high—i.e., the ticket is too long—then the analyst has found an externally-caused danger. In this case, she should simply record a human-readable name for the manifestation; further analysis and explanation will be performed in the next step.

Manifestations in M-SAFE are recorded in Cells D-I of row 13 (and additional rows for each additional predecessor link) of the worksheet in Figure 4.15. The previously discussed "not dangerous" interaction is recorded in cell D11 of Figure 4.16, while the overlong ticket danger is documented in cell C11.

In T-SAFE manifestations are recorded in two steps, using AADL's EMV2 error types

```
package PCA_Shutoff_Errors
public
with MAP_Errors;

annex EMV2
{**
error types

  -- Pump Manifestations
  LongTkt : type extends MAP_Errors::High;
  TktTooShort : type extends MAP_Errors::Low;
  NoTkt : type extends MAP_Errors::Halted;
  ErraticTkt : type extends MAP_Errors::Erratic;
  EarlyTkt : type extends MAP_Errors::Early;
  LateTkt : type extends MAP_Errors::Late;

end types;
**};
end PCA_Shutoff_Errors;
```

Figure 4.19: *The PCA Pump's possible manifestations, extending from the base failure domains in Figure 4.18*

and their propagations:

1. *Create Manifestations:* First, manifestations must be created, which is itself a two part process:

   (a) *Create Error Types:* We leverage the error type system to declare error types, and error types should be linked to their respective failure domains via type extension. Figure 4.18 shows the six base failure domains encoded as AADL EMV2 types. Figure 4.19 shows the usage of these types by a developer of the PCA Pump's interface. The names of the extended types should be short, human-readable descriptions equivalent to manifestation names written in M-SAFE.

   (b) *Declare In Propagation:* All six component-specific error types should then be declared in the component type's incoming propagations, even if they are not hazardous given the implementation of the component. This is done since the type of the component may be refined into different implementations, some of which may propagate out successor dangers given a particular input failure where a different, better implementation would not. An example of this input propagation declaration is shown in line 15 of Figure 4.13.

2. *Specify Caused Successor Dangers:* Second, the component implementation should specify which input errors cause which successor dangers. This is done using EMV2's `error paths`: line 11 of Figure 4.14 shows the link between the PCA pump receiving an overlong ticket and the overadministration of analgesic. Incoming errors that are not dangerous are recorded using `error sinks`; see line 13 of Figure 4.14 for a specification of the previously-discussed "short tickets" issue.

4.3.3 Documenting External Interactions

The final step in Activity 1 is to fully document the connection between each particular manifestation and the successor danger or dangers it would cause. Broadly, this documentation has three parts: a) identifying the state of the component and the worst-case environmental state, b) providing a narrative description of how the manifestation causes the successor danger, and c) specifying any applicable run-time detection or handling steps. These three tasks will be repeated for each pairing of one manifestation and one of the successor dangers it would cause. That is, if there are n manifestations which each cause m successor dangers, these steps will be repeated once per possible pairing, or $n \times m$ times.

Identifying Component and Environment States

After selecting an unexamined manifestation and successor danger pairing, an analyst's first task is to fully identify the component and environment states that describe the potential danger. Recall that SAFE uses Leveson's two-part notion of hazard, which we extend to the component level as undesirability (see Section 4.1.1). The goal of this task is document both the system and worst-case environment states necessary for this successor danger to occur.

The state of the controlled process does not need to be explicitly given as it is implicit in the successor danger: a successor danger traces to a particular safety constraint violation, which itself traces to a hazard, and that hazard is associated with a worst-case environmental state. Leveson explains that safety constraint violations come about as a result of either inappropriate commands[8] being provided or appropriate commands not being followed [30]. Errors causing the former can often be stated in terms of a process model mismatch, i.e., a situation where a component's process model is out-of-sync with the actual state of the controlled process.

Thus, in M-SAFE, we ask analysts to provide—when possible—semi-formal statements involving the state of the process variables relative to the state of the controlled process. In the PCA Interlock scenario, for example, the "ticket duration" process variable might be described as having a value that is "too high." Here, "too high" implicitly refers to the state of the controlled process, i.e., the patient's respiratory health status. An example of this documentation is given in cells C19 and D19 of Figure 4.16. In T-SAFE, we ask the analyst to reference the relevant process variable using an AADL `reference` property, and then to specify the constraint on its state using a string, as in lines 39 and 40 of Figure 4.14. We recognize that these constraints would ideally be fully formal, and note that there is similar research in this area which allows the modeling and verification of process variables as LTL properties using XSTAMPP and SPIN [96].

The state of the system includes more than the state of the process model: simultaneously occurring errors should also be documented. In M-SAFE these co-occurring dangers are documented in column G of the externally caused dangers section of Figures 4.15 and Figures 4.16. Note that each safe and unsafe occurrence of a manifestation should have its own row, so if some danger A requires another danger B to cause successor danger C, four entries will be created in the externally caused dangers section: a) C, caused by A with

[8]Leveson uses the term "control action"

co-occurring B; b) no danger, when A occurs alone; c) C, caused by B with co-occurring A; and d) no danger, when B occurs alone. Documenting co-occurring dangers is considerably more straightforward in T-SAFE, as the left-hand side of error path declarations can contain multiple, comma-separated error types. Thus, an analyst need only document the collective occurrence with one error path, after which analysis can proceed as normal.

Providing a Narrative Description

At this point in Activity 1 Step 2, the analyst should have a clear understanding of exactly how the current manifestation causes the given successor danger. This task involves simply documenting that understanding in a human-readable format.

What we know about our running example involving the PCA pump's use in the PCA Interlock scenario might be phrased as a) the manifestation of "TicketTooLong" on the pump's incoming-tickets link, b) the successor danger of "TooMuchAnalgesic" leaving on the PCA Pump-IV Line link, c) the system state "PCA Pump receives a ticket with a non-zero length of time," and d) the associated worst-case environment state "The patient is at risk of respiratory distress, and cannot safely tolerate more analgesic." Thus, the analyst should phrase all of this together into a succinct description like "The PCA pump receives a non-zero ticket when the patient cannot tolerate any more analgesic, which leads to the pump administering drug when it should not."

In M-SAFE, this would be written in Cell E23 of the worksheet, under the "Interpretation" heading, see Figures 4.15 (worksheet) and 4.16 (PCA Interlock example). In T-SAFE, this would be written in the `Explanation` subproperty of the `ExternallyCausedDanger` property, which is applied to the relevant error path. The property type specification is shown in lines 7-11 of Figure 4.20, and lines 38-42 of Figure 4.14 show a completed example.

Specifying Run-Time Detections and Compensations

The last thing an analyst should do when considering a particular manifestation and

```
property set MAP_Error_Properties is

-- Other properties removed for space

ErrorHandlingApproachType : type enumeration (Rollback, Rollforward,
  ↪  Compensation);

ExternallyCausedDanger : record (
  ProcessVariableValue : reference(data);
  ProcessVariableConstraint : aadlstring;
  Explanation : aadlstring;
) applies to (all);

RuntimeErrorDetection : record (
  ErrorDetectionApproach : enumeration (Concurrent, Preemptive);
  Explanation : aadlstring;
) applies to (all);

RuntimeErrorHandling : record (
  ErrorHandlingApproach : MAP_Error_Properties::ErrorHandlingApproachType;
  Explanation : aadlstring;
) applies to (all);

end MAP_Error_Properties;
```

Figure 4.20: *Property types used in Activity 1 of SAFE*

successor danger pairing is think about how their co-occurrence might be avoided. Avižienis et al. explain error detection and handling techniques, and we provide the names of these techniques as guidance to the analyst.

Error detection, which "identifies the presence of an error" is broken down into two categories in [3]:

1. *Concurrent Detection:* This "takes place during normal [operation]," and typically requires some sort of designed-in detection mechanism.

2. *Preemptive Detection:* This "takes place while [operation] is suspended," and typically requires routine inspections of the element.

Avižienis and his co-authors divide error handling, which "eliminates errors from the system state," into three categories [3]:

1. *Rollback:* This technique "brings the system back to a saved state that existed prior to error occurrence," and requires a system that can save its state, also known as "checkpointing."

2. *Rollforward:* This involves using a "state without detected errors [as the] new state," and requires the system to have a safe state. This may not be feasible, since some systems will not have a default safe state.

3. *Compensation:* Here, "the erroneous state contains enough redundancy to enable the error to be masked," which requires a system designed with redundancy.

Not all of these techniques are available for every system design, and some may be prohibitively difficult. Leveson argues that error handling strategies relying on compensation can be particularly complex, even to the detriment of the overall system's safety, writing that [97]:

> Although redundancy provides protection against accidents caused by individual component failure, it is not as effective against hazards that arise from the interactions among components in the increasingly complex and interactive systems being engineered today. In fact, redundancy may increase complexity to the point where the redundancy itself contributes to accidents.

If an analyst determines that the current manifestation and successor danger pairing can be detected and/or handled by the system, she should write down the technique (if applicable) and then provide a short, human-targeted description. For example, when considering the issue of early message arrival, the analyst might specify that a concurrent strategy involving minimum message inter-arrival times would enable detection of early messages. Then, if the system supports network enforcement of these specifications, the offending messages could be dropped and the system itself rolled forward into a safe state. We note that the MDCF uses this detection approach—though it does not automatically roll forward—which is enabled by its MIDAS networking [19].

In M-SAFE these explanations, prefaced by the name of the technique (if applicable) are documented in cells H23 and I23 of Figure 4.15. T-SAFE allows considerably more flexibility, at the cost of greater complexity. Detectable problems are first specified using component-specific error events, which bind the error type associated with the manifestation to a named error event; see lines 22-23 of Figure 4.14. Technique names and human-readable descriptions are specified using the `RuntimeErrorDetection` property, which is specified in lines 13-16 of Figure 4.20. An example usage of this property is shown on lines 30-33 of Figure 4.14. Compensations for these events are specified using the component's `component error behavior`'s `transitions` section, which essentially allows the creation of state machine-like transitions that describe the component's implementation-specific behavior in the presence of errors. Error handling technique names and human readable descriptions are specified in a manner similar to detections, i.e., by using the `RuntimeErrorHandling` property which is shown in lines 18-21 of Figure 4.20. The handling properties should then be applied to the named transitions between error-behavior states that are guarded by occurrence of the previously declared detections; see line 27 of Figure 4.14.

At this point, the documentation of this manifestation and successor danger pairing is complete. The analyst should pick a new, undocumented pairing, or move on to another available step, e.g., Activity 2's first step on this element, or Activity 1 on a different element.

4.4 Activity 2: Internally Caused Faults

SAFE's Activity 2 focuses on identifying and documenting faults that occur within a component or by non-element entities (adversaries, physical objects, etc.) in its environment. These faults are linked to the successor dangers identified in Activity 1's first step, so beginning Activity 2 requires that step's completion. This activity consists of two steps: the elimination of classes of faults based on properties of the element, and then the documentation of any faults that were not eliminated.

4.4.1 Eliminating Classes of Faults

As discussed in Section 4.1.3, Avižienis et al. identify eight fault classifications, of which SAFE uses six. Based on these categories, we have developed the following questions which an analyst can use to eliminate various faults from consideration. Note that the numbers used in this list correspond to the fault numbers from Table 4.1.

1. *Phase of Creation or Occurrence:* Should faults from the element's development be considered?
   - *Yes:* Development and operational faults
   - *No:* Operational faults only (Remove 1-7)
2. *Dimension:* Does the element involve hardware, software, or both?
   - *Hardware:* Hardware only (Remove 1-3,13-15)
   - *Software:* Software only (Remove 4-12)
   - *Both:* Both hardware and software
3. *Phenomenological Cause 1:* Will the hardware elements be protected from natural phenomena?
   - *Yes:* Natural faults excluded (Remove 7-9)
   - *No:* Natural faults included
   - *No hardware elements:* Natural faults excluded (Remove 7-9)
4. *Phenomenological Cause 2:* Does the element receive input from directly from a human operator that is not modeled as an element?
   - *Yes:* Human-made operational faults included

- *No:* Human-made operational faults excluded (Remove 10-11,14-15)

5. *Objective 1:* Is it possible that an adversary could gain access to the element during development?

    - *Yes:* Malicious development-time faults included
    - *No:* Malicious development-time faults excluded (Remove 3-4)

6. *Objective 2:* Is it possible that an adversary could gain access to the element during operation?

    - *Yes:* Malicious runtime faults included
    - *No:* Malicious runtime faults excluded (Remove 12-13)

7. *Interaction:* Have the two components joined by this connection either worked together before or been developed together?

    - *Yes:* Interaction faults excluded (Remove 16-18)
    - *No:* Interaction faults included
    - *N/A:* If the current element is a connection, interaction faults should be excluded (Remove 16-18)

Any faults eliminated by considering these questions should be documented, along with a justification. For example, the PCA pump used in a particular instantiation of the PCA Interlock scenario might be one that has been safely used for years, so things like bad hardware design (fault 6) need not be considered. Similarly, the pump is not a connection, so its developers do not need to consider interaction faults.

In M-SAFE, this documentation is done in the "Faults Not Considered" section of the report, which is shown in rows 31-33 of the worksheet in Figure 4.15 and rows 27-33 of the PCA Interlock example in Figure 4.16. In T-SAFE, this documentation is done with the `EliminatedFaults` property, the declaration of which is shown in lines 16-19 of Figure

4.21). Note that, as in M-SAFE, multiple faults may be excluded with a single explanation, as the `FaultTypes` subproperty is a list.

4.4.2 Documenting Internal Faults

The documentation of internal faults is similar to documenting unsafe external interactions, as described in Section 4.3.3. The differences are that a) faults, rather than manifestations, are paired with successor dangers, b) process variable names and values are not documented, c) mechanisms for design-time detection are considered, and d) additional runtime handling techniques are available. The first difference is self-explanatory, since this activity is focused on faults rather than manifestations. Note, though, that in T-SAFE, error sources are used instead of error paths, since faults occur without being triggered by an error event's arrival (see lines 16 and 17 of Figure 4.14). The second difference is also straightforward: the process model of a component is updated entirely using information arriving from other components; if the behavior of those components is not considered (as is the case in Activity 2) then a component's process model can be ignored. The final two differences merit more discussion, though.

Specifying Design-Time Detections

While all classes of dangers may be detectable at runtime, some faults can be detected while a system is still being designed. These are dangers that come about due to problems in an element's development, and there are several techniques that can be used to detect them. Avižienis et al. identify five techniques, broken down into two categories [3]:

- *Dynamic Verification:* These techniques involve executing the system.
    - *Symbolic Execution:* Symbolically executing a system involves using symbols rather than concrete values. As system execution progresses, the symbols become increasingly constrained by the statements that make up the current execution

path. The path-specific collection of constraints is referred to as a *path condition*, and can be tested at any point to determine if the path is viable or if it violates certain analyst-specified properties.

  - *Testing:* Perhaps the most common form of detecting problems at design time, testing consists of simply providing some known inputs to a system and then verifying that it behaves as expected. While testing can be specialized in a number of ways (by domain, stage of development, etc.) and is in general quite flexible, it lacks the analytic power of the other verification techniques. In particular, arguments for completeness of test coverage can be difficult to make.

- *Static Verification:* These techniques do not involve system execution, but rather analysis proceeds on either models or descriptions of a system.

  - *Model Checking:* This involves building a model of a system in some modeling language, and then verifying certain properties about that model. One challenge with this approach is ensuring that the model of a system aligns with the system as it is built.
  - *Static Analysis:* This is an umbrella term for any of a number of techniques which examine static descriptions of a system. Many static analyses are built into compiler toolchains to catch relatively simple coding errors.
  - *Theorem Proving:* A more sophisticated technique, theorem proving requires stating and proving claims about a system. Those statements need to be checked not only for their provability, but also—like the models used in model checking—for their adherence to the system's actual behavior.

Documenting which strategy an analyst thinks is best for detected design-time problems requires specifying both the detection technique (if applicable) and providing a short narrative description. For example, the best approach to ensure that the PCA Interlock's app logic is free from built-in bugs (fault 1 in Table 4.1) might be formal verification, since it is a relatively small piece of software that is vitally important. In M-SAFE, the analyst would write something like "Formal Verification: Since the app logic should be relatively simple but needs a very high level of assurance, the core algorithms should be formally verified." This should be entered in the Design-time Detection section of the worksheet, which is F38 in Figure 4.15.

Like Activity 1's specification of runtime error detection and handling, specification of design-time fault detection in T-SAFE is slightly more involved. As in Activity 1, the documentation of detectable design-time faults involves specifying an event, in the component's error behavior declaration, which is associated with the fault's type; see line 25 of Figure 4.14, which specifies that the `PumpDeteriorates` event is associated with the `Deterioration` fault type. The analyst would use the `DesignTimeFaultDetection` property specified on lines 5-8 of Figure 4.21 to associate a detection mechanism and narrative explanation with the fault's occurrence.

Additional Run-Time Handling Techniques

Faults which are detectable at runtime are documented similarly to detectable errors, except that an analyst can declare an additional technique for rectifying the underlying fault. Avižienis et al. explain four such "fault handling" techniques [3]:

- *Diagnosis:* This technique involves identifying and recording the cause of the problem, and can be difficult to automate.
- *Isolation:* This involves physically or logically "excluding the components," which will make the fault dormant (i.e., extant but harmless).
- *Reconfiguration:* This involves the swapping in of spare components or the reassignment of "tasks among non-failed components." Both this technique and the previous

```
property set MAP_Error_Properties is

-- Other properties removed for space

DesignTimeFaultDetection : record (
  FaultDetectionApproach : enumeration (StaticAnalysis, TheoremProving,
  ↪  ModelChecking, SymbolicExecution, Testing);
  Explanation : aadlstring;
) applies to (all);

RuntimeFaultHandling : record (
  FaultHandlingApproach : enumeration (Diagnosis, Isolation,
  ↪  Reconfiguration, Reinitialization);
  ErrorHandlingApproach : MAP_Error_Properties::ErrorHandlingApproachType;
  Explanation : aadlstring;
) applies to (all);

EliminatedFaults : record (
  FaultTypes : list of reference({emv2}** error type);
  Explanation : aadlstring;
) applies to (all);

end MAP_Error_Properties;
```

Figure 4.21: *Property types used in Activity 2 of SAFE*

one overlap somewhat with compensation-based error handling approaches, since they rely on redundancy.

- *Reinitialization:* This technique involves resetting (i.e., "rebooting") the system, in the hopes that it is in a valid state after having been restarted.

When addressing a particular fault, an analyst may decide that either fault or error handling techniques (i.e., those discussed in Section 4.3.3) would be individually preferable, or that their combined use would be best. In M-SAFE these approaches are documented using columns H and I in the "Internally Caused Dangers" section of the worksheet in Figure 4.15.

Consider, for example, a situation where the clinician misunderstands the patient's health and provides an overly-strong prescription. Since this clinician isn't modeled (at the current level of abstraction), this problem would be considered a fault. Its solution might be would be two-fold: First, the app should have a carefully-designed user interface (UI) that would make such mistakes difficult to commit; this itself a topic of study, see e.g., [98]. Second, training for use of the PCA Interlock should be periodically re-performed to account for any incorrect adaptations clinicians may mistakenly make; Leveson writes that "Systems and organizations continually experience change as adaptations are made in response to local pressures and short-term productivity and cost goals." [30] An example of M-SAFE's documentation of this fault handling technique is shown in cell I39 of Figure 4.16.

Documentation in T-SAFE is similar to that of the detection mechanisms specified previously, except that the `RuntimeFaultHandling` property (which is specified in lines 10-14 of Figure 4.21) is used. Like the detection documentation, the the handling documentation would be applied to the component error behavior events associated with the fault's type.

4.5 Assessment

In this section we present the conjectures we make about our hazard analysis techniques, which are divided into those regarding objective and subjective attributes. The discussion of objective attributes is designed to orient the reader to other popular hazard analysis techniques and to position our work relative to them, while the section covering subjective attributes presents a postulated comparison and ranking of the techniques according to specific desiderata. We compare our manual and tool-assisted processes to the three hazard analysis techniques introduced in Section 2.2.2 (FTA, FMEA, and STPA) as well as FMEA reports that are autogenerated from EMV2 annotations on AADL models (see Section 2.3.3).

4.5.1 Objective Attributes

In this section we explain the cells in Table 4.4 that are annotated with a superscript number. The numbers reference comments which provide a justification for the annotated

Name	Errors	Faults	Phase	Qua[L] / Qua[N]t	Math	Detail	[T]op-[D]own / [B]ottom-[U]p
FMEA	P[1]	P[2]	PD-DD	L/N	Y	High	BU
EMV2 FMEA	P[1]	P[2]	DD[3]	L/N	N[4]	High	BU
FTA	P	F	PD-DD	L/N	Y	Med/High[5]	TD
STPA	F[6]	P[7]	CD[8]-PD-DD-Oper[9]	L	N	Med/High[5]	TD
M-SAFE	F[10]	F[11]	PD[12]-DD	L	N	Med/High[5]	TD & BU[13]
T-SAFE	F[10]	F[11]	DD[14]	L	N	Med/High[5]	TD & BU[13]

Table 4.4: *Summary of objective attributes of major hazard analyses discussed in this dissertation, with references to explanations in the text of Section 4.5.1. Abbreviations used: P – Partial, F – Full, PD – Preliminary Design, DD – Detailed Design, CD – Conceptual Design, Oper – In Operation. Adapted from Table 3.4 of [7, pg. 47]*

STPA's Hazardous Control Action	Avižienis et al.'s Service Failure Mode
Not Providing Causes Hazard	Halt Failures
Providing Causes Hazard	Erratic Service
Wrong Timing/Order Causes Hazard	Early/Late Timing Failures
Stopped Too Soon or Applied Too Long	Early/Late Timing Failures

Table 4.5: *A mapping from STPA's* ways control actions can be hazardous *to Avižienis et al.'s* service failure modes. *Hazardous Control Action terms are from Figure 8.4 in [30, pg. 219], Service Failure Modes are from Fig. 8 in [3, pg. 9]*

entry; table entries without numbers are considered to be self-explanatory.

1. *FMEA – Full Detection of Errors:* FMEA would not typically catch, for example, component interaction errors.

2. *FMEA – Full Detection of Faults:* FMEA would not typically catch, for example, faults in software.

3. *EMV2 FMEA – Detailed Design Only:* Because EMV2 annotations are added to AADL system descriptions, which cannot easily be created for preliminary designs, an EMV2-supported FMEA cannot be done on a preliminary design. Note that this is

STPA's Fault-Related Guideword	Avižienis et al.'s Fault Class
Flaws in creation (Controller)	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11
Component failures (Controlled Process)	11, 12, 13, 14, 15
Process changes (Controller)/Incorrect modification or adaptation (Controller)/Changes over time (Controlled Process)	19, 20, 21, 29, 30, 31

Table 4.6: *A mapping from STPA's fault-related guidewords to Avižienis et al.'s fault classes. The STPA guidewords are from Figure 8.6 in [30, pg. 223], Avižienis et al.'s terms are from Figure 5 in [3, pg. 6]*

mitigated somewhat by the ease with which system models that are built in AADL can be updated. Additionally, it may be possible to capture a preliminary design in AADL by using appropriate abstractions.

4. *EMV2 FMEA – No Math Required:* As EMV2 is tool-supported, it can produce the estimates of a hazard's likelihood (and impact, if the necessary annotations for a FMECA are added) without requiring the analyst to perform the underlying calculation himself.

5. *Several – Medium/High Level of Detail:* These hazard analyses can be performed at either a moderate or high level of detail, depending on which the analyst chooses.

6. *STPA – Full Detection of Errors:* We believe that the classes of unsafe control actions identified in STPA's first step can be argued as equivalent to Avižienis et al.'s service failure modes using the mapping in Table 4.5. There is an important caveat to this mapping, though, in that Leveson's terms are specific to binary control actions. Extensions to ranged control actions (e.g., value too high or too low) are natural, however, and guide-phrases for analyzing complex (i.e., aggregate or parametric) control actions have been documented by Thomas and Leveson [99].

7. *STPA – Partial Detection of Faults:* While Leveson does a good job of covering the

space of potential errors, her guidewords for faults are quite abstract and do not approach the level of coverage provided by Avižienis et al. STPA's set of causality guidewords are not fault specific: i.e., the terms for errors, like "Delayed operation" are mixed in with terms for faults, like "Flaws in creation." While STPA's fault terms arguably include 22 of Avižienis et al.'s fault classes (see Table 4.6 for the full mapping), they do so at such a high level of abstraction that actually achieving full coverage is unlikely. For example, the STPA term "Component failures" might be argued to cover classes 11-15 of Avižienis et al.'s faults, but it would require considerable analyst skill to look at that term and derive faults from all five fault classes. See Section 4.5.2 for a full discussion of the difference in difficulty between SAFE and STPA.

8. *STPA – Applicability in Conceptual Design:* As it is described in [30], STPA is most applicable to preliminary and detailed designs, but recent work by Fleming extended the analysis to conceptual designs [100].

9. *STPA – Applicability in Operation:* Though we primarily discuss STPA in this dissertation, Leveson has also developed *Causal Analysis based on STAMP* (CAST) which can be used for post-hoc reasoning [30].

10. *SAFE – Full Error Coverage:* We use Avižienis et al.'s service failure modes for our analysis of errors that propagate into an element. The authors argue that there are two failure domains, content and timing, and that failures can be classified into one or both of those domains [3]. Since a service failure in a preceding element is equivalent to an error, we claim complete, though admittedly abstract, coverage of failures.

11. *SAFE – Full Fault Coverage:* We use a reduced set of Avižienis et al.'s fault classes for our analysis of root causes [3]. We further supplement these classes with three more designed to detect faults arising from element interactions, and we claim that we achieve the best possible coverage of faults using this combined set.

12. *M-SAFE – Applicability in Preliminary Design:* Since the manual version of our process only requires names and box-and-line diagrams for system descriptions, it can be applied to a preliminary design which exists at a more abstract level of specification than, say, a model built using our subset of AADL.

13. *SAFE – Top-Down and Bottom-Up Nature:* The new process described in this dissertation contains both top-down and bottom-up steps. Step 0 involves working backwards from the accidents and hazards that need to be avoided, while step 2 involves taking faults and considering how they would impact the rest of the system. Step 1 exists between the two notions, as it involves marrying the top-down notion of successor dangers with the bottom-up concept of manifestations.

14. *T-SAFE – Exclusive Applicability to Detailed Design:* As the tool-assisted version of our process relies on a system model built in AADL, it necessarily requires a detailed design, since preliminary designs cannot easily be built in AADL. Like the EMV2-supported FMEA, though (see comment 3), we note that these designs are more easily updated than manually-specified detailed designs.

4.5.2 Subjective Attributes of Previous Hazard Analyses

Name	Analytic Power	Time Req.	Complexity	Skill Req.
FMEA	– –	*	++	*
EMV2 FMEA	–	+	++	+
FTA	–	–	*	–
STPA	–	+	+	–
M-SAFE	*	*	*	*
T-SAFE	+	+	*	*

Table 4.7: *Summary of conjectures regarding subjective attributes of major hazard analyses discussed in this dissertation*

In this section, we establish and explain our arguments regarding the subjective ratings of four attributes of four previously-existing hazard analysis techniques, as summarized in Table 4.7. The notation used is one of comparisons, where + signifies a relative improvement, – signifies a relative degradation, and * signifies a rough equivalence[9].

- *Analytic Power:* The degree to which an analyst, following the specified process, is able to detect both errors and faults in a system.
- *Time Required:* The amount of time required by an analyst to perform a "complete" analysis of a system using the specified process.
- *Complexity:* The cognitive load of the analyst as she performs the analysis.
- *Skill Required:* Also referred to as difficulty, this is the degree to which analyst skill with the technique (as opposed to with the system being analyzed) affects the quality of the analysis.

Failure Mode and Effects Analysis

Failure Mode and Effects Analysis (FMEA)—which is also sometimes known as Failure Mode, Criticality, and Effects Analysis (FMECA) if the impacts of failures are examined—is typically performed by working through a worksheet and considering the effect on a system if its component parts failed in particular ways. As with FTA, there are many versions of the analysis, and we consider here the process as described by Ericson [7].

- *Analytic Power:* FMEA excels at discovering and documenting problems resulting from elements that have known rates and patterns of failure, e.g., mechanical components. Unfortunately, it is less applicable to more modern, software-based electromechanical devices as well as the sociotechnical environments in which the systems are used.

[9]Note that these comparisons should not be read as value judgements, i.e., FMEA has considerable analytic power and we are not making any claims about the technique's overall quality, only the four specified attributes relative to the other analyses discussed in this dissertation.

- *Time Required:* Performing a FMEA consists of working through a worksheet, and so the time required is bounded. Additionally, component interactions are not considered as deeply as in, e.g., STPA or our technique, so system elements can be considered, to some extent, in isolation.
- *Complexity:* As a worksheet- and process-based technique, FMEA places relatively low demands on an analyst's cognitive load.
- *Skill Required:* FMEA is fairly straightforward, and so it has a relatively low difficulty, though skill with mathematics is sometimes required to calculate system failure probabilities from the probabilities of faults in system components.

EMV2 Supported FMEA

The architecture modeling language AADL's error modeling annex (EMV2) can be used to automatically generate FMEA-like reports from annotated architectural models (see Section 2.3.3). This yields a number of improvements over a manually performed FMEA, most of which stem from the architectural integration and automated report generation.

- *Analytic Power:* Though the analytic power of AADL's EMV2 is not strictly greater than that of a manual FMEA, it is in practice because of the ease of keeping an AADL model and its EMV2 annotations in sync. Since updating a system model nearly always either automatically updates or breaks the EMV2 annotations (which generates warnings in the OSATE editor), the risk and effort involved in synchronizing a model and its analysis is drastically reduced compared to a manual FMEA.
- *Time Required:* As annotations can be added directly to a system model, and since report generation is nearly instantaneous, producing the FMEA-like reports using EMV2 takes considerably less time than other, manual techniques.
- *Complexity:* The complexity for an EMV2-aided FMEA is roughly equivalent to the manual process, since the same concepts are involved.

- *Skill Required:* Less expertise on the part of the analyst is required, since the OSATE (or other AADL) tooling takes care of some of the trickier parts of the analysis (e.g., the statistics/mathematics) automatically.

Fault Tree Analysis

As explained in Section 2.2.2, there are a number of ways to perform Fault Tree Analysis (FTA) with varying levels of rigor. For the purposes of this evaluation, we assume that the technique used is the one described by Ericson in [7]. That is, there are a variety of nodes including events, failures, conditions, etc., and that sophisticated statistical analysis can be performed on the resulting fault tree.

- *Analytic Power:* Fault-tree analyses can, in the hands of a skilled analyst, find large numbers of faults and errors, as there is no inbuilt preference for particular classes of problems. That is, unlike FMEA which is better suited to finding hardware faults, an FTA can theoretically find nearly any type of problem.

- *Time Required:* While shallow FTAs can be quickly performed on a preliminary system design, the more complete analyses can take considerable time. There may also be significant rework required when considering several top-level hazards (or "Undesired Events" in the language of [7]) since trees do not typically share branches. Additionally, it is up to the judgement of the analyst when the analysis is complete, so estimating time or progress can be difficult.

- *Complexity:* The cognitive load required of an analyst performing an FTA is typically fairly light, as the process is somewhat repetitive and there are well defined construction rules. More advanced rules are available (see for example Section 11.5.4 of [7]) as are cut-set generating algorithms (e.g., MOCUS and bottom-up) which can increase the complexity.

- *Skill Required:* The analytic power of FTAs scales up with the skill of the analyst, so this technique can be classified as quite difficult. Though Ericson and others provide an excellent guide, mastering the more advanced gates, cut-set algorithms, and statistical formulae required are not easy.

System Theoretic Process Analysis

As a relative newcomer to the world of hazard analyses, STPA is still under a great deal of active research. While the primary text for the technique remains Leveson's 2011 book, a number of new studies and techniques have been published since the book's release that expand the core process in a number of ways [30, 100, 90, 85, 80]. It was the primary inspiration for our technique, though, and is arguably the state-of-the-art analysis for many domains.

- *Analytic Power:* STPA, like FTA, has considerable analytic power. It has been shown to uncover more errors, in less time, than FMEA [80]. That said, it does not provide guidewords that cover the full scope of fault classes treated by Avižienis et al. (see note 7 in Section 4.5.1).

- *Time Required:* STPA is, when compared to FTA or FMEA, a quick analysis. The unsafe control action table, from step one, can be filled out relatively quickly, and as the process does not involve advanced mathematics and statistics only qualitative information is required. The technique also has the advantage of being applied at various levels of abstraction (and even early or late in the system development process, see notes 8 and 9 in Section 4.5.1), so it can produce valuable results quickly.

- *Complexity:* STPA is not a terribly complex analysis, though there are some terminology and system-theoretic concepts which must be understood. Most of the analysis is compiled into free-form reports, and the results of steps one and two, even as produced by Leveson's students, can vary considerably according to the needs of the system.

See, for example, the “Fluidic Controller” in Figure 20 of [80] as compared to the “TTTPS Control System” in Figure 9.2 of [30], or the tabular step two format used in Table 11 of [85] and Table 7 of [80] as compared to the annotated control loop style of Figure 8.7 of [30]).

- *Skill Required:* STPA’s difficulty is higher than the technique in this dissertation, owing somewhat to STPA’s low complexity. Though *Engineering a Safer World* is lengthy and provides numerous examples, at no point does it lay out a concrete worksheet or low-level process for the technique, relying instead on numerous examples [30]. Unfortunately, those examples vary according to the system, and so some skill level with the technique is required to get a high-quality, repeatable result.

4.5.3 Subjective Evaluation of SAFE

In Table 4.7, we claim—relative to STPA, as described in [30]—that our manual process provides increased analytic power with decreased difficulty at a cost of increased time and complexity. Our tool assisted process further increases upon the analytic power of the manual process, and decreases the time required while maintaining complexity and difficulty. Note that we compare our techniques directly to STPA in this section, as a six-way comparison (additionally involving FTA, FMEA, and EMV2-FMEA) would be impractical.

In summary, we argue that our technique fares well when compared to STPA’s analysis of the software- and hardware-based elements of socio-technical systems because it is, in several important ways, less ambiguous. Where STPA has two steps (and derivation of a system’s “Fundamentals”) given as narrative descriptions, SAFE (as specified in Appendix A) has three high level activities, which are divided into seven steps, which are further subdivided into thirty-four individual tasks. These steps specify not only the work that should be done by an analyst but also the order in which it should be done. Additionally, STPA’s guidewords are not as specific as those that we derive from Avižienis et al.’s work, nor does it delineate between errors and faults as cleanly as SAFE. Overall, both the high-level organization of SAFE and the directions pertaining to individual tasks have been designed to be as unambiguous as possible.

Analytic Power

Our assertion of increased analytic power stems from three improvements over STPA: a comparatively firm theoretical grounding, incorporation of Avižienis et al.'s taxonomy, and—in the tool-supported version of the process—a deep architectural integration.

Theoretical Grounding The theoretical backing of this work is explained more fully in Chapter 5. There, we present a set of formalisms that take as their foundation Leveson's conceptualization of a hazard—that of a system and environment state pairing which will lead to harm—that we believe are more useful than existing formalizations, e.g., [90]. We then connected our formalization to two other, pre-existing mathematical systems: Wallace's fault propagation and transformation calculus and Shankar's lazy compositional verification [35, 84]. Further, the relation between elements in the control structures is specified to be intransitive interference, and is grounded in Halpern and Pearl's notion of actual cause theory (as explained by Leitner) [91, 83, 89]. We use intransitive interference rather than the important but ultimately unmechanizable system theoretic model of causality described by Leveson and further expanded upon by Masys, who extensively uses Latour's Actor-Network Theory [30, 34, 101]. This enables the step-by-step process described in Chapter 4 which, when compared to the much more high-level and abstract set of instructions given in [30], will lead to more complete, repeatable analyses by virtue of having less room for analyst creativity and inconsistency.

Taxonomy of Secure and Dependable Computing A second reason that we claim our new process has more analytic power than STPA is because of our incorporation of Avižienis et al.'s taxonomy. We use several concepts from their work, including fault classes, failure domains, fault tolerance techniques, and fault tolerance coverage [3]. These cover, and indeed can to some extent map to[10], other sets of guidewords, such as those from STPA's first and second steps, concepts arising from Dolev and Yao's saboteur model, and AADL's error modeling library [30, 93, 33]. What Avižienis et al.'s work provides, however, is a clean delineation between internally and externally caused errors which correspond cleanly to steps one and two of the process described in this dissertation. This correspondence provides a degree of completeness not found in the other sets of guidewords: AADL's EM annex and Dolev and Yao's work are error-only, while Leveson's second-step terms are highly abstract and include both faults and errors. Avižienis et al.'s work also incorporates, as first class citizens, fault classes relating to security, which begins to bridge the gap between safety and security analyses (itself a significant challenge, see e.g., [102]).

Architectural Integration The third factor that increases the analytic power of the tool supported version of the hazard analysis is its deep architectural integration. While manual hazard analyses can be quite powerful, they are done without the considerable assistance made possible by integrated development environments, e.g., code completion, type checking, etc. Additionally, integrated hazard analyses like FMEA with EMV2 and T-SAFE are much easier to keep synchronized with the model of the system being analyzed, since there is only one artifact to keep updated. In fact, Feiler has used the phrase "single source of truth" to refer to an AADL-based project model repository [66]. What's more, the subset of AADL used to construct the system model that the hazard analysis integrates with can be used, as described in Section 3.4, for code generation, virtually guaranteeing that the system model operated on by the analysis is in sync with the system as constructed. When this is compared to SAFE's manual process or STPA, which rely on tools like spreadsheet software or whiteboards, the advantages of architectural integration become apparent.

[10]See, for example, Table 4.6 which gives a mapping from STPA's fault-related causality guidewords to some of Avižienis et al.'s fault classes

Skill Required

SAFE requires less analyst skill than STPA in order to achieve quality results for two reasons. First, we have provided a very specific, step-by-step process; and second, the guidewords used can be further refined by domain experts to tailor the analysis to a particular class of system.

Specific Process The specific process we described in Chapter 4 (and fully specified in Appendix A) provides a fairly rigid set of steps that constrain an analyst's possible actions to a small (typically single) number of next steps. This was done to reduce the creativity and skill required by the technique so that it is less difficult. While systems engineering knowledge, skill with the technique, and creativity are still necessary, it is to a significantly reduced degree as compared to STPA. The flexible formatting of STPA's documentation can be quite useful to a skilled analyst, but can also greatly impede understanding by those who are not experts.

Guideword Refinement As described in Section 4.1.3, the fault guidewords used in step two of SAFE can be tailored by domain experts to provide greater specificity to an analyst who is considering faults in an element. This might be done by, e.g., a standardization body like AAMI or IEC as a way to "bake" domain knowledge into the process. This goes beyond reducing skill required with the hazard analysis technique and actually reduces, to a small degree, the amount of domain expertise an analyst would need to perform his work.

Time Required

Compared to STPA, the manual version of our process can take more or less time. More analyst time is required as a result of the depth of the analysis, but this is remedied, to a varying extent, by: analyzing only externally caused dangers (i.e., not considering faults), parallelizing the process, and—in the tool-supported version of the process—automated report generation.

Thoroughness One of STPA's great strengths is its myriad aspects of flexibility, not the least of which is its ability to operate smoothly at a number of different levels of abstraction. Part of this ease comes from the open-ended formatting of the documentation generated by the first and second steps—an analyst is free to write as little or as much as she would like about a given component, hazard, or control action. This lets her focus her work exactly where she would like, and it means there is very little time spent analyzing irrelevant faults or parts of the system. The drawback to this flexibility, though, is an increase in difficulty and a reduction in the process's repeatability. We have prioritized these traits over the time required, though, and so SAFE takes more time as a result.

One remedy to this situation is that the second step to our analysis is severable, though the fault-finding benefits would of course not be realized. Still, the first step alone is a powerful process, and results in what are essentially "contracts" for each element (in the spirit of Meyer's *design by contract*—see, e.g., [103]). Skipping the second step speeds up the analysis considerably, and the analysis of any or all components can be returned to at a future time if desired.

Parallelization One unique feature of our analysis is its step towards compositionality which can enable several analysts to work in parallel. While the process is not fully compositional—and indeed, safety analyses may never be—only the first part of the analysis must be performed before individual elements can be analyzed separately. Specifically, all of step 0 and the first part of step 1 must be done either on the system as a whole, or in a backwards-moving linear process. The results of performing step 1.1 on a given element, though, can be handed off to a different analyst without damaging the quality of the results in the final report. Thus, if time is critical and there are multiple analysts, significant time savings can be realized.

Report Generation The increased time potentially required by our process can be further mitigated, however, by the report generation capabilities utilized in the tool-supported version of our process: once the annotations are in place, producing the report is nearly instantaneous. Though there is an argument to be made that the AADL system descriptions and our annotations are unwieldy when compared to the hand-drawn system diagrams and manually formatted tables used by STPA, the true power of automatic report generation is shown when a report needs to be modified or updated. Small tweaks to a system's design can ripple through an analysis requiring a number of changes in disparate places in the full report. Finding all of these changes manually is at best time consuming, and at worst so difficult as to desynchronize the analysis from the system being analyzed—a significant safety risk. With automated report generation, though, these changes are either automatically propagated throughout the report or revealed as automatically-detected syntax errors, e.g., when annotations apply to no-longer-extant system elements. Thus, the system being analyzed and the analysis itself are always kept synchronized.

Complexity

STPA has its own ontology which an analyst must understand before competently performing an analysis. We recognize that our process builds on that with new terminology (e.g., element, link, danger, etc.), relationships (e.g., successor, predecessor, etc.), and ways of looking at a system (e.g., the controlled process as a component that exists partially inside and partially outside a system's boundary). These additional concepts must be learned, understood, and kept in mind by an analyst. This increases the cognitive load placed on an analyst relative to STPA, though we have mitigated it to some extent by providing a thorough, step-by-step process description and set of worksheets.

4.5.4 Threats to Validity

Though we have attempted to be impartial in our analysis, and wherever possible based our work on well-regarded, pre-existing literature, there are three not-insignificant threats to the validity of the evaluation in this section: the first two are threats to internal validity, while the third is a threat to external validity. First, we have created SAFE and our efforts to be impartial may not have been enough, leading to a biased analysis. Second, we have based our evaluation of STPA on the publications available which are almost exclusively from Leveson's students. It is possible that their evaluations have been influenced by their proximity to the creator of the process. Third, Table 3.4 in Ericson's [7]—from which our Table 4.4 is derived—is not explained directly in the text. Thus, we have had to make some inferences based on our understanding of the techniques listed. Further, we adapted the table significantly, and it is possible that we have misconstrued Ericson's intent in doing so.

Chapter 5

Theoretical Foundations

5.1 Introduction

In this section, we sketch a generalized notion of hazard that can support the device- and app-based MAP vision of systems development—where a component may at one point be a standalone system with its own notion of harm, but may at a later time be a subcomponent that contributes to a harm in a broader system. This generalized concept informs the full hazard analysis approach in Chapter 4 that (if developed further) would enable analysts to reuse significant portions of a component's hazard analysis when determining the safety of a system of which it later becomes a part.

We believe that the appropriate starting point for such a development has appeared both in the medical standardization literature (i.e., IEC 80001) and in modern hazard analyses (i.e., STPA) [32, 30]. This starting point is the idea is that the occurrence of harm is a twofold notion: not only must a system be in a particular, worst-case state, but so too must its environment. Leveson uses the example of a train's doors: they are only in a hazardous state if they are open (system state: doors open) when the train is moving (environment state: train moving). Though this idea is a straightforward one, formalizations of system safety do not incorporate it; both the system safety and formal methods communities typically use their own, single-level concept of hazard. System safety standards (e.g., IEC 60601, IEC 61508 and ISO 14971 [49, 46, 31]) typically define a hazard as a "potential source of harm," though Leveson notes that "...most every system state has the potential to do harm or can lead to an accident" [30, p. 184].

In the formal methods community, however, safety properties are typically explained as system states that should always be avoided. Even though the definitions used are very precise, the concept named "safety" is rarely linked to an explicit concept of hazard or an observable interaction between a system and its environment that causes harm. Thus, many of the rigorous mathematical concepts from that domain cannot be directly applied to support automation of—or rigorous reasoning about—safety assessment steps in system engineering. This is not to say that we believe hazard analysis can be completely mechanized, but rather that providing mathematically founded definitions is a prerequisite to improved automation, precision, and clearer methodologies.

5.1.1 Hierarchical Depth, Component Role, and Undesirability

We take as this chapter's goal, then, to formalize the definition of hazard from Leveson's *Engineering a Safer World* in order to develop a vocabulary and reference model that relate an entity to its hierarchical environment. It is our hope that in doing so we can provide a) a clearer statement of hazard-related concepts in a hierarchical systems context, b) a basis for greater automation (which may come as a bridge between the system safety and formal methods communities), and c) allow reuse of risk management assets in order to enable compositional approaches to safety assessment like the one described in Chapter 4. Leveson's definition is "[a] system state...that, together with a particular set of worst-case environmental conditions, will lead to an accident." [30] It is not within the scope of this work to defend the validity of this definition, but as it has been proven through Leveson's work and similar notions appear in some safety standards, we take it as axiomatic. This definition gives us a clear partition between the system and its environment, and by making explicit the environment's role, it associates a notion of harm with some event that crosses the system boundary. This definition works well for components that exist at the penultimate level of a system's hierarchy—the top-level sensors, controllers, and actuators that directly sense or modify the state of the environment. But it is unsatisfactory for lower levels: it is likely that, for example, our system's controller component is itself composed of sub-sensors, -actuators, and -controllers that all interact with the component's local environment, but not the top-level system environment (i.e., all these sub-sensors may do is "sense" messages arriving on the network, rather than directly monitoring some aspect of the controlled process).

We cannot consider these subcomponents in the context of the top-level environment or we will lose the ability to characterize a component's properties using only local concepts—and reasoning locally about a component is the key to global compositional reasoning. What, then, can we know about a component without considering the full system of which it is a part? Rather than marry our components tightly to a specific controlled process, we recognize that components view the state of the system through the lens of their *role*: an actuator's view of the system state is simply "should actuate" or "should not actuate[1]." As part of this work, we have focused on five such roles. First, a special role, *top*, that includes the system and its environment, and four that the top-level component decomposes into (which come from STPA) [30]: *sensor*, *actuator*, *controller*, and *controlled process*. Consider the train speed sensor from Leveson's example: if it fails to correctly inform the door controller that the train is moving, that will clearly contribute to the previously discussed hazard. But this same sensor might be used in a different application that requires knowledge of a train's motion, and—if certain conditions are met—it stands to reason that some of the original safety analysis may be reusable. We formalize this intuition by introducing a generalization of Leveson's hazard, which we term undesirability.

Definition 1: ***Undesirability***—A component state that, together with a particular set

[1]In [30], Leveson gives a number of hazard identification guidewords; ultimately, though, they all condense down to "actuated when inappropriate" and "failed to actuate when necessary."

of worst-case conditions of its environment, will produce an unwanted, observable effect.

The foundational issues addressed by this definition are that undesirability manifests as a) a pair of one component state and one state of its environment that is b) observable (i.e., the component is somehow affecting its environment) and c) unwanted. In fact, a hazard can be seen as a special case of this definition, where our component is the conventional system, the accident (i.e., harm) is the unwanted effect, and the observer is the controlled process. In this work, we have developed formalisms that result from the natural progression of this thinking. First, we allow reasoning about what observable, abstract system states a component can be in. Second, we allow the underlying notions of undesirability to be linked to a component's interaction points (i.e., the ways in which the component can affect its environment via communication, energy dissipation, etc.). Third, we discuss how these links can drive analysis of a) component- and system-level safety using formalisms from the formal methods community, and b) impacts of internal faults or externally-produced errors. Then, to the extent that these abstractions are generic (and subject to underlying assumptions), the analysis is re-usable: an actuator which responds to external commands need not have its safety-related aspects re-examined if only the source of the commands change. That is, while we do not claim to have a fully compositional process for system safety (nor do we claim that such a concept is even possible), we believe that the work in this chapter is an important step toward supporting re-use of component-level safety-related reasoning.

5.2 Process

In this section we explain specifically how the compositional aspects of our hazard analysis process from Chapter 4 work. Table 5.1 has examples of the results of these steps having been applied to the PCA Interlock scenario. Note, however, that we only discuss the SpO_2 sensor (leaving out other possible physiological parameters like Respiratory Rate and $ETCO_2$). These steps would be repeated at various levels of the system hierarchy, but for discussion

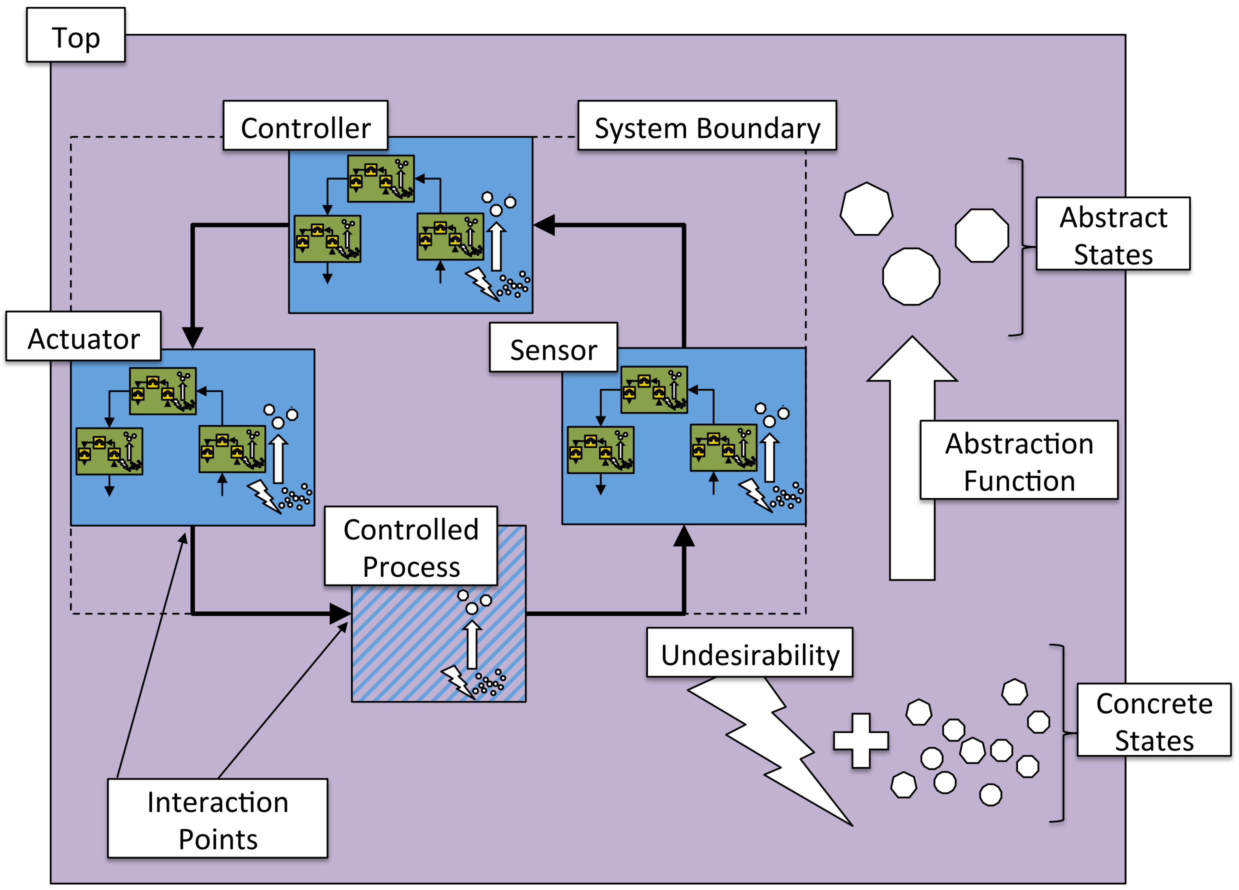


Figure 5.1: *Semantic objects in our formalism, labels used in our definitions are shown in Figure 5.2*

we arbitrarily chose a single component of each role and traced them though the full process.

1. For the purpose of these formalizations, we term the current element under analysis m. The important aspect of this step, for the formalisms presented here, is to find a delineation between sub-elements inside an element's boundary and those outside of it, i.e., those of its environment. As a result of this step, an analyst should be able to identify using Definition 2 the component's name (m), set of concrete states (S), role (r), interaction points (I), and if applicable subcomponents (Sub).

2. When given a state of component m and a state of its environment, an analyst will determine whether or not the pairing is both wanted and observable according to

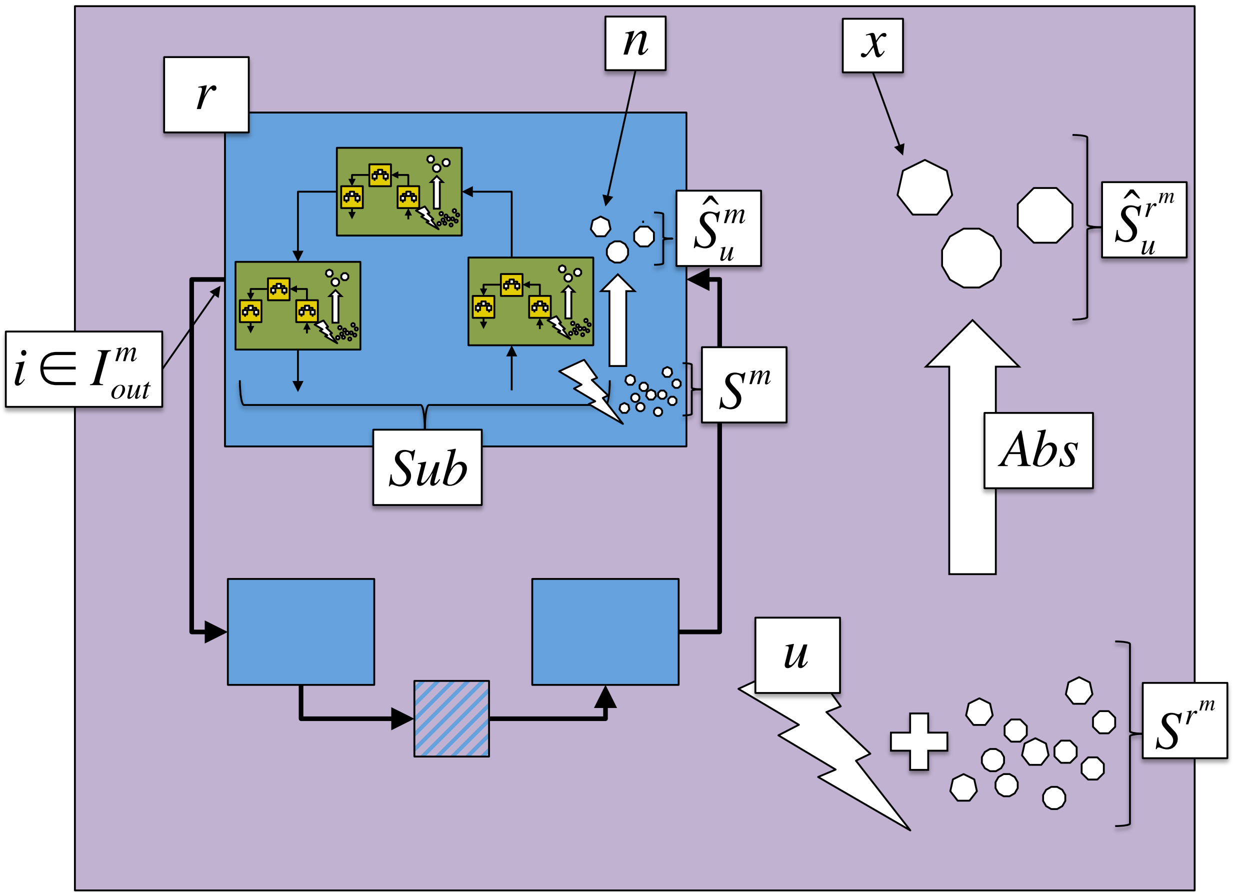


Figure 5.2: *Labels used by our formalism, descriptive labels are shown in Figure 5.1*

some notion of undesirability u using Definition 3: Concrete Undesirability. This corresponds to Definition 1: Undesirability from Section 5.1.1. Note that u is a not a state or set of states, but rather a concept which we use to index our later definitions. There are two potential sources for this notion of undesirability, depending on if an analyst is working with a component in isolation (i.e., without a surrounding system context) or one that is part of a full system.

(a) For isolated components, our analyst would anticipate the use of her device and identify a corresponding harm, i.e., an analgesic pump manufacturer would anticipate that overadministration would be a potential source of harm. This is similar to how device manufacturers currently have in mind an *intended use* when devel-

Name	**Sensor**	**Controller**	**Actuator**	**Process**
		Definition 2: Component		
m	Pulse Oximeter	Interlock App	PCA Pump	Patient
S	$\{\dots\}$	$\{\dots\}$	$\{\dots\}$	$\{\dots\}$
r	Sensor	Controller	Actuator	Process
I	$\{(Blood, in), (SpO_2, out)\}$	$\{(SpO_2, in), (PumpCmd, out)\}$	$\{(PumpCmd, in), (Analgesic, out)\}$	$\{(Analgesic, in), (Blood, out)\}$
Sub	$\{BloodMonitor, SpO2Calc, SpO2Send\}$	$\{SpO2Monitor, PumpCmdLogic, CmdSend\}$	$\{CmdMonitor, PumpLogic, PumpMotor\}$	N/A
		Definition 3: Concrete Undesirability		
u	Overestimating patient health	Enabling the pump erroneously	Pumping when unsafe	Overadmin. of Analgesic
		Definition 5: Abstraction		
$\hat{S}^m_u$	$\{1\%, 2\%, \dots\}$	$\{CmdPumpEnable, CmdPumpDisable\}$	$\{GiveDrug, NoDrug\}$	$\{Healthy, Risk, Overdose\}$
$\hat{S}^{r^m}_u$	$\{1\%, 2\%, \dots\}$	$\{ShldEnabPump, ShldntEnabPump\}$	$\{ShldGiveDrug, ShldntGiveDrug\}$	N/A
		Definition 6: Abstract Undesirability		
n	95%	$PumpEnable$	$GiveDrug$	N/A
$\boldsymbol{S}^m_{u,n}$	$\{94\%, 93\%, \dots\}$	$\{ShldntEnabPump\}$	$\{ShldntGiveDrug\}$	
$\mathbb{S}^m_{u,n}$	$\{95\%, 96\%, \dots\}$	$\{ShldEnabPump\}$	$\{ShldGiveDrug\}$	
		Definition 8: Avoidance		
$\mathbb{I}^m_u$	$\{Blood\}$	$\{SpO_2\}$	$\{PumpCmd\}$	N/A

Table 5.1: *Examples of the formalisms applied to components of the PCA Interlock scenario*

oping a particular device, see the discussion of ISO 14971 in Section 2.2.5. Thus, for isolated components, u is a violation of one of the component's safety-related functional goals and in the language of SAFE is a *successor danger*.

(b) For a full system, our analyst would identify a notion of harm, which is equivalent to an Accident in SAFE or STPA, or an Unintended Consequence in IEC 80001 [30, 32]. Then, they would "push down" that top-level notion into individual elements. In the manual version of SAFE, this is done when the analyst imports the Successor Dangers from a successor component, see Section 4.3.1. For example, in the PCA interlock scenario, the system is designed to avoid an overdose. Thus, the PCA pump running when the patient is at risk of an overdose would be the actuator-specific version of that harm (see Table 5.1 for more examples). Unlike in (a), here u is a system-level notion, and each subcomponent must take into account the behavioral aspects of the rest of the system—this is similar to how device manufacturers judge whether or not a particular software or hardware subcomponent is fit for use in a given device.

3. Using the relation established in Definition 3, we now establish an equivalence relation using Definition 4: Distinguishability that allows us to group the states of either an element or its environment. These groups can then be abstracted into a set of equivalence classes using Definition 5: Abstraction. This is simply a formalization of the implicit, intuitive step that analysts perform now in many hazard analyses when they create a mental model of how their system interacts with a notion of harm. STPA uses the term "Process Model" for this concept, and we re-use the term, requiring the creation of a process model for controller components in SAFE (see Section 4.3.1).

   Consider again Leveson's moving train example from [30]. Rather than differentiate between cases where the train's doors are 24 or 25 inches apart, the doors are said to be "open" and the states are equivalent to the notion of a passenger falling out. Similarly, regardless of if the train is moving at 60 or 61 miles per hour, we simply say the train is "moving" (and these names are what we mean by representatives of the equivalence classes identified in Definition 5). Even in more traditional hazard analyses (e.g., FTA or FMEA), huge numbers of concrete states are grouped together by their observable effect on the fault, failure, or safety problem being considered (see, e.g., [7]). We believe that analysts will benefit from formal guidelines that speak to the appropriateness of their mental models, and that this formalization is necessary before suitable tooling can be developed.

4. For each abstract state of the component, use Definition 6: Abstract Undesirability to consider which states of the component's environment (as viewed through the component's role) are undesirable. That is, given our definition of undesirability from Section 5.1.1, what are the "worst case conditions" of our component's environment associated with this system state and notion of unwanted, observable behavior? If our analyst is working with a full system, this step is fairly straightforward: it is undesirable for the pump to have a nonzero ticket duration when the patient cannot tolerate more analgesic (i.e., the environment is in the abstract state "the pump should not run"). Similarly, because higher SpO_2 means that our patient's respiratory status is healthier, it is undesirable for the pulse oximeter to overstate the patient's actual blood-oxygenation level—understating the value may be unwanted for other reasons (e.g., avoiding underinfusion), but in this example we focus on overinfusion. If our analyst is working with a component in isolation, though, this step requires anticipating a class of harm and considering how the component's observable behavior could contribute to that harm. Pairs of the form (*abstract component state*, *undesirable abstract environment state*) that exist at the penultimate level of the system hierarchy are equivalent to the system's hazards.

5. Finally, determine which interaction points (typically an element's ports) carry information about the component's environment by using Definition 7: Environmental Awareness, and verify that it is possible for the component to "know" when it is in any potentially undesirable state of its environment using Definition 8: Avoidance. In order for a component to meet its safety-related goals, it is necessary[2] for it to be informed of the state of its environment so it can avoid the specified notion of undesirability; here we ask the analyst to identify which incoming interaction points carry

[2] Assuming correct behavior and freedom from side-channel interference; Section 5.5 discusses a calculus for when this assumption is not met

environmental information relevant to u. This is equivalent to identifying what the component requires from the environment, a common step in requirements-gathering methodologies, e.g., "2.2.4 'Choose Monitored Variables'" in [104] or determining the members of the vector $\underset{\sim}{i}^t$ in [94].

Having completed this process, our analyst will know the subset of her component's interaction points that need to be connected in a fully instantiated system in order to avoid u. That information can then be used to either argue that a composed system (in which every component has been similarly analyzed) is safe (Section 5.4), or to consider what happens when the required information is not provided (Section 5.5). After this process has been completed once, the resulting requirements for avoidance of u (i.e., $\mathbb{I}_u^m$) can be used to argue that a component's safety-related goals are met when it is reused in other systems.

5.3 Formalisms

Our process relies on seven formalisms, which we now document and explain. An instantiation of the architecture from Figure 2.3 corresponding to the PCA interlock app from Section 2.1.5 is shown in Table 5.1.

Definition 2: *Component*—As discussed in step one of the process, we define a component as a five-tuple, (m, S, r, I, Sub) where:

- m is a unique identifier
- S is the set of concrete states that the component can be in
- $r \in \{sensor, controller, controlledprocess, actuator, top\}$ is the components role
- $I \in (name, dir)$ is the set of *interaction points* that the component uses to communicate with other components in its environment
    - $name$ is a unique identifier for the interaction point

  - $dir \in in, out$ is a direction

- Sub is the set of subcomponents that this component decomposes into

We write I^m_{in} and I^m_{out} to denote the set of m's interaction points where $dir = in$ or out, respectively. We denote the concrete states of m as S^m, and we write the concrete environmental states of m as S^{r^m}, which reflects our findings that a component's view of its environment role-specific.

Definition 3: ***Concrete Undesirability***—Given a component m, $n \in S^m$ (a concrete state of m), $x \in S^{r^m}$ (a concrete state of m's environment), and u (a notion of undesirability); we define the relation (using a double-struck turnstile: $\Vdash$ to denote undesirability) $(n, x) \Vdash u$, where:

- $(n, x) \Vdash u$ signifies that the observable aspects of n are undesirable with regards to u when m is in state n and its environment is in state x

- $(n, x) \nVdash u$ signifies that the observable aspects of n are not undesirable with regards to u when m is in state n and its environment is in state x

When this relation holds, it means that m is observably undesirable according to u when it is in state n and its environment is in state x; thus, this definition represents a formalization of Definition 1.

Definition 4: ***Distinguishability***—As discussed in step three of the process, we want to—when given two concrete states (which we term n_1 and n_2 if they are states of a component and x_1 and x_2 if they are states of a component's environment)—be able to say if the states are *equivalent* to one another according to u. That is, we define the equivalence relation $\sim_u$ where:

- $n_1, n_2 \in S^m : n_1 \sim_u n_2 \iff \forall x \in S^{r^m}, (((n_1, x) \Vdash u) \wedge ((n_2, x) \Vdash u)) \vee (((n_1, x) \nVdash u) \wedge ((n_2, x) \nVdash u))$

- $x_1, x_2 \in S^{r^m} : x_1 \sim_u x_2 \iff \forall n \in S^m, (((n, x_1) \Vdash u) \wedge ((n, x_2) \Vdash u)) \vee (((n, x_1) \nVdash u) \wedge ((n, x_2) \nVdash u))$

Intuitively, the first relation holds if two component states produce the same result under $\Vdash$ for all environmental states of u (that is, the states are indistinguishable to u). Similarly, the second relation holds if two environmental states produce the same result under $\Vdash$ for all component states.

Definition 5: ***Abstraction***—As discussed in step four of the process, given a component m, and a notion of undesirability u, we (using the notation for a collection of equivalence classes of $S/{\sim_u}$ from, e.g., Beachy and Blair [105]) partition the concrete states of the component and its environment into a set of representative abstract states, which we denote with a $\hat{}$. That is, $Abs(m, u) = (\hat{S}^m_u, \hat{S}^{r^m}_u)$, where:

- $\hat{S}^m_u = S^m/{\sim_u}$
- $\hat{S}^{r^m}_u = S^{r^m}/{\sim_u}$

For convenience, we often want to speak of individual (representative) abstract states, even though $\hat{S}^m_u$ and $\hat{S}^{r^m}_u$ are sets of equivalence classes. When we write $n \in \hat{S}^m_u$, we mean that n is a representative of the equivalence class $[n] \in \hat{S}^m_u$. For example, for the pulse oximeter, we write "1%" as the canonical representative of the equivalence class of concrete sensor readings that would be reported to the clinician through a pulse oximeter's display panel as 1%. On the other hand, the abstract states for the pump (relative to the notion of pumping when unsafe) are *GiveDrug*, which abstracts all the states where the pump is running, and *NoDrug*, which abstracts the states where it is not pumping. Examples of this notation (and this definition) are in Table 5.1.

Definition 6: ***Abstract Undesirability***—Corresponding to process step four, given an abstract state n of component m and a notion of undesirability u, we (using the convention that a doublestruck letter (e.g., $\mathbb{S}$) signifies desirability while a boldface letter (e.g., $\boldsymbol{S}$) signifies undesirability), define $Undes(n) = (\hat{\boldsymbol{S}}^m_u, \hat{\mathbb{S}}^m_u)$ where:

- $\hat{\boldsymbol{S}}^m_{u,n} = \{x \in \hat{S}^{r^m}_u | (n,x) \Vdash u\}$

- $\hat{\mathbb{S}}^m_{u,n} = \{x \in \hat{S}^{r^m}_u | (n,x) \nVdash u\}$

This relation uses $\Vdash$ to split (according to u) the abstract environmental states of component m (when it is in state n) into two subsets: those that are undesirable ($\hat{\boldsymbol{S}}^m_{u,n}$) and those that are not ($\hat{\mathbb{S}}^m_{u,n}$). For example, given the abstract pulse oximeter state of 95%, and wanting to avoid overestimating the patient's respiratory health, any environmental state where the patient's true SpO_2 is below 95% would be undesirable. Similarly, if the pump is running (*GiveDrug*), it is clearly undesirable for the patient to be in such a state that more analgesic will lead to an overdose (*ShldntGiveDrug*). Table 5.1 gives examples of $\hat{\boldsymbol{S}}^m_{u,n}$ and $\hat{\mathbb{S}}^m_{u,n}$ for some components of the PCA Interlock scenario.

Abstraction functions necessarily destroy information about their input in order to produce their (simplified) outputs. The end result of definitions 5 and 6 is a collection of abstract states, in $\hat{\mathbb{S}}^m_{u,n}$ and $\hat{\boldsymbol{S}}^m_{u,n}$, that preserve the undesirability (or its absence) from the concrete states of m relative to u. That is, these functions preserve only whether or not a particular component state is undesirable relative to some notion that the analyst wants to avoid. If an analyst were to compare a real-world system to the output of Definition 6, she would note that the *only* correspondence between the two would be the undesirability of the system's component's states relative to u.

Definition 7: *Environmental Awareness*—Given a subset of component m's incoming interaction points $J \subseteq I^m_{in}$ and a subset of the concrete states of m's environment $X \subseteq S^{r^m}$, we define the relation $J \Rrightarrow X$, where:

- $J \Rrightarrow X$ signifies that m will know when it is in any state $x \in X$ if every $i \in J$ is connected when the system is instantiated.

- $J \not\Rrightarrow X$ signifies that m will not know when it is in one or more $x \in X$ even if every $i \in J$ is connected when the system is instantiated.

Intuitively, this relation describes which interaction points carry information that can inform the component what state its environment is in. J is a set of input interaction points (e.g., “SpO2”) and X is a set of environmental states (e.g., “Patient’s blood-oxygen saturation is 98%”).

Definition 8: ***Avoidance***—As discussed in the final step of the process, given a component m and a notion of undesirability u, we define $Avoid(m, u) = \mathbb{I}^m_u$, where:

- $\mathbb{I}^m_u = \{J \in I^m_{in} | \forall n \in \hat{S}^m_u, J \Rrightarrow \hat{\boldsymbol{S}}^{r^m}_{u,n}\}$

This definition leverages the set of undesirable abstract states ($\hat{\boldsymbol{S}}^m_{u,n}$) from Definition 6 and the relation ($\Rrightarrow$) from Definition 7 to derive the set of input interaction points necessary to avoid a particular notion of undesirability. The final row of Table 5.1 has examples of its application to the components of the PCA interlock scenario. For example, in order for the pulse oximeter to avoid overestimating the patient’s respiratory health, a system composition would need to provide the device with access to the patient’s blood. Similarly, in order for the PCA pump to avoid running when it should not, it must have access to pump commands from a controller.

5.4 Compositionality

In this section, we outline a preliminary avenue of investigation into how the definitions we have previously established might be used as a bridge between the hazard analysis and formal methods communities. Recall from Section 5.1 that the two communities use the term “safety” in different ways: it is used by practitioners of formal methods to refer to avoidance of a particular system state in all possible executions of a system, and by hazard analysts to refer to the more general concept of the absence of harm.

Here, we sketch a correspondence by showing how our formalisms from Section 5.3, which is derived from the hazard analysis community’s definition of hazard, can be used in a formal methods context. The context we have chosen to use is Shankar's "Lazy Compositional Verification," which is a compositional verification technique described in [84]. There are other compositional verification techniques, most notably Assume-Guarantee reasoning, which we could have used, but we found lazy compositional verification particularly well suited to providing "a useful decomposition of the verification task" [106, 107, 84]. We note that Shankar's work has been used in formalizing other compositional domains, e.g., the Multiple Independent Levels of Security (MILS) project [108]. The sketched correspondence has four steps:

1. *Establish a "baseline" system:* First we describe a system containing two elements, a hypothetical "SafePCA" device and a patient.

2. *Decompose one component into two subcomponents:* Then, we decompose the SafePCA device into an application and a standard PCA pump.

3. *Prove that the subcomponents collectively refine the original:* Next, we show that the system comprised of the patient, application and standard PCA pump is equivalent to the baseline system.

4. *Further refine one subcomponent with an implementation:* Finally, we further refine the application—from a specification down to an implementation—and show that this refinement is valid.

In this section, we make the following simplifying assumptions:

- *Communication is via shared variables:* In many MAP implementations, including the MDCF, communication takes place via ports and channels. However, for the purpose of this section, we model communication via shared variables.

- *Elements behave correctly:* In the real world, of course, components and connections are fallible. However, our goal in showing the safe decomposition of a system is not to

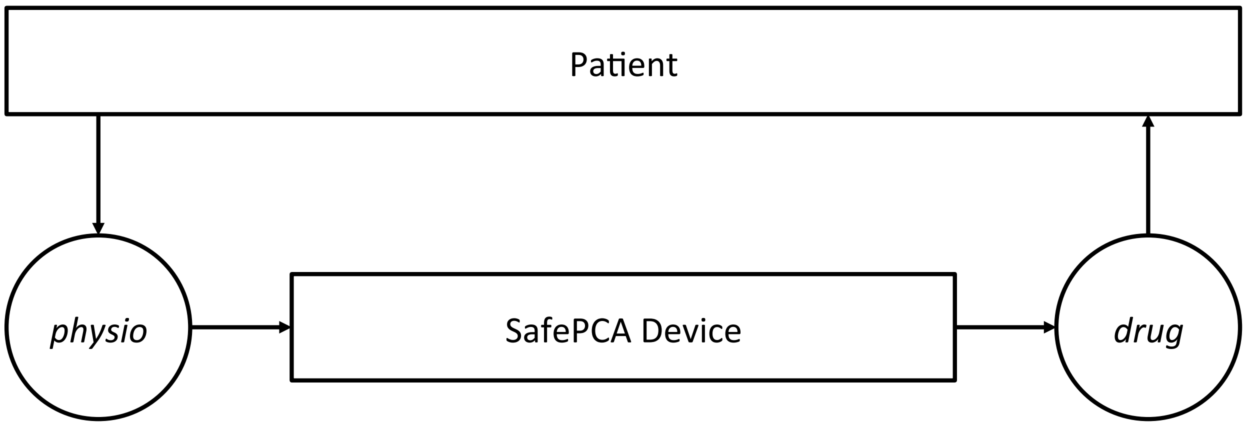


Figure 5.3: *A hypothetical "SafePCA Device" in its environment, derived from Figure 3 in [84]*

consider behavior in the presence of faults, but rather the validity of the decomposition. We discuss a formalism for modeling systems in the presence of incorrect behavior in Section 5.5.

- *There is no platform:* The notion of a platform, while critical to the MAP vision, is elided here for simplicity's sake: components communicate directly with one another.
- *We use the abstract states of the systems directly:* In order to emphasize the compositional aspects, we have de-emphasized the clinical detail of our system by replacing the specific parameters consumed and produced by each component (e.g., SpO_2, $ETCO_2$, pump tickets, etc.) with aggregate parameters.
- *We consider only ordering, not timing:* The formalisms in this chapter do not include a notion of time, so our invariants are stated in terms of ordering.

5.4.1 A Baseline System

Here we introduce a system designed to roughly correspond to the PCA interlock scenario discussed in Section 2.1.5. The scenario has been simplified somewhat, though. Instead of an app and a PCA pump, we introduce a combination component referred to as a "SafePCA" device that consumes our physiological parameter and administers doses of analgesic when safe (see Figure 5.3. Similarly, instead of sensors and a patient, we have one component titled "Patient," that consumes doses of analgesic and produces physiological values directly. Unlike the system described by Arney et al. in [26], our pump either administers a dose each "step" or it doesn't; we note that this can be thought of as a ticket-based system where the ticket length is invariably the duration of the system's most fine-grained notion of time, and tickets are always consumed when available.

Preliminaries

Both the baseline system described here and its decomposition (described below) use two variables, *physio* and *drug*, as well as queues recording their history. These variables use the equivalence classes produced by Definition 5: Abstraction where $m = SafePCA$ and $Patient$ and $u = overdose$ (which are reprinted here from Table 5.1). Additionally, we note that the SafePCA device really acts as both an actuator and a controller, so we could use either $\hat{S}^{Actuator}_{overdose} = \{ShldGiveDrug, ShldntGiveDrug\}$ or $\hat{S}^{Controller}_{overdose} = \{ShldEnabPump, ShldntEnabPump\}$. As they are equivalent there is no reason to use or the other, and so we arbitrarily choose the former.

- $\hat{S}^{SafePCA}_{overdose} = \{GiveDrug, NoDrug\}$
- $\hat{S}^{Actuator}_{overdose} = \{ShldGiveDrug, ShldntGiveDrug\}$
- $\hat{S}^{Patient}_{overdose} = \{Healthy, Risk, Overdose\}$

So the variables and their possible values are:

- *physio*: Aggregate physiological value, $physio \in \{Healthy, Risk, Overdosed\}$
- *physioh*: Queue of physiological values
- *drug* Whether or not the pump administers analgesic, $drug \in \{GiveDrug, NoDrug\}$
- *drugh* Queue of administered doses

SafePCA Description

We first focus specifically on the hypothetical SafePCA device. As an objective, we would like to be able to prove that the device avoids overdose given correct input. From the preliminaries, we know that the device will take in physiological values and produce doses of analgesic, i.e., $I_{in}^{SafePCA} = physio$ and $I_{out}^{SafePCA} = drug$. Using these interaction points as the observable aspects of the device and its environment in Definition 3, we get: $(Dose, PhysioVal) \Vdash overdose \iff (Dose = GiveDrug \wedge PhysioVal \in \{Risk, Overdose\})$. That is, given a dose and a physiological value, the pairing of the two is undesirable with regards to an overdose if and only if the SafePCA device is running and the patient is either at risk of, or already suffering from, an overdose. In more natural language, this might be stated as the pump is administering analgesic while the patient is showing signs of respiratory distress—certainly an undesirable situation.

The Next-State relation In Shankar's [84], elements are defined as asynchronous transition systems, which are expressed as triples of the form $\langle \Sigma; I, N \rangle$ where Σ is a state type (and is typically elided, we follow this convention here), I is an initial state or set of states, and N is a next-state relation. For the SafePCA device, we define N as the function run, which determines whether or not it is appropriate to allow a dose of analgesic if the previous ticket has been consumed:

$$run \triangleq \begin{cases} drug = \bot \\ \wedge physio' = physio \\ \wedge drug' = p2d(physio) \\ \wedge physioh' = physioh \\ \wedge drugh' = enqueue(p2d(physio), drugh) \end{cases}$$

Note that incoming values are written as $variableName$, while outgoing values are denoted with an apostrophe, i.e., $variableName'$. Thus, any expressions with a left-hand value of an incoming variable should be read as a guard on the relation, while expressions with an outgoing variable on the left-hand side are the results of the transition having occurred.

run relies on the helper function $p2d(physio)$ which uses its parameter (a physiological reading) to determine whether or not a dose of analgesic should be allowed:

$$p2d(PhysioVal) \triangleq \begin{cases} PhysioVal = Healthy : GiveDrug \\ PhysioVal \in \{Risk, Overdose\} : NoDrug \end{cases}$$

The Initial State The SafePCA device does not issue a dose of analgesic at first, and initializes the history of doses $init_{SafePCA} \triangleq (drug = NoDrug \wedge drugh = enqueue(NoDrug, empty))$

These functions then combine to the full specification of the SafePCA device: $SafePCASpec \triangleq \langle init_{SafePCA}, run \rangle$.

Avoidance of Undesirability We now want to establish that this component will avoid overdosing the patient. This argument is established by two claims: first, that the SafePCA pump has the necessary awareness of its environment (i.e., it upholds Definition 8: Avoidance), and then that, by using the environmental information in its next-state relation, it avoids undesirability (i.e., it upholds Definition 3: Concrete Undesirability).

Claim 1. $I_{in}^{SafePCA} \subseteq \mathbb{I}_{overdose}^{SafePCA}$

Proof Sketch. From Definition 5: Abstraction, we know that $Abs(SafePCA, overdose) = (\hat{S}_{overdose}^{SafePCA} = \{GiveDrug, NoDrug\}, \hat{S}_{overdose}^{actuator} = \{ShldGiveDrug, ShldntGiveDrug\})$. We establish this claim in two cases, one for each state in $\hat{S}_{overdose}^{SafePCA}$.

1. *NoDrug:* From Definition 6: Abstract Undesirability, we know that $\hat{\boldsymbol{S}}_{overdose,NoDrug}^{SafePCA} = \varnothing$, i.e., that if the SafePCA device isn't administering analgesic, it is never at risk of overdosing the patient. Thus, this case is trivial: the device requires no information about the patient since this state is always not undesirable.

2. *GiveDrug:* From Definition 6, we know that $\hat{\boldsymbol{S}}^{SafePCA}_{overdose,GiveDrug} = \{ShldntGiveDrug\}$, i.e., if the SafePCA device administers a dose when it should not, the patient will be overdosed. The device should not administer a dose when $physio \in \{Risk, Overdose\}$, so we need to find $J \subseteq I^{SafePCA}_{in} \Rightarrow ShldntGiveDrug$, i.e., some subset of the incoming interaction points to the element should carry either *physio* itself or other values that would enable the calculation of *physio*. As the SafePCA device has only one incoming interaction point which carries *physio*, this requirement is satisfied.

□

Claim 2. $(Dose, PhysioVal) \Vdash overdose \iff (Dose = GiveDrug \wedge PhysioVal \in \{Risk, Overdose\})$

Proof Sketch. We establish this claim by contradiction. We assume that there is a SafePCA-caused overdose, which means that the device administers analgesic ($dose = GiveDrug$) while the patient's health is in a deteriorated state ($physio \in \{Risk, Overdose\}$). From inspection of the next-state relation, we see that *dose* is the result of evaluating $p2d$ with some physiological value as input. Thus, our assumption implies that $p2d(PhysioVal \in \{Risk, Overdose\}) \overset{?}{=} GiveDrug$, which by inspecting the definition of $p2d$ and substituting the actual for the formal parameter leads to $Risk \overset{?}{=} Healthy \vee Overdose \overset{?}{=} Healthy$, which is clearly false and we have arrived at a contradiction.

□

Patient Description

The SafePCA device operates in the context of its environment, which we term the "Patient." While in the real world no patient provides machine-readable physiological values, Shankar's logic allows each element to specify its environment as an abstract component that can be refined lazily, i.e., at a later time.

The Next-State relation As environments are simply abstract components, they too are specified via next-state relations and initial states. Note that, in order to avoid overconstraining the environment the SafePCA device operates in, we do not specify any particular pharmacokinetics (i.e., the patient's *physio* value need not depend on the administration of analgesic).

$$metabolize \triangleq \begin{cases} drug \neq \bot \\ \wedge drug' = \bot \\ \wedge physioh' = enqueue(physio' \in \{Healthy, Risk, Overdose\}, physioh) \\ \wedge drugh' = drugh \end{cases}$$

Initial state We assume the patient is initially at risk of overdose: $init_{Patient} \triangleq (physio = Risk \wedge physioh = enqueue(Risk, empty))$.

Thus the full specification of the SafePCA device's environment is $\langle init_{Patient}, metabolize \rangle$.

Analysis

We now establish an invariant on the system composed of our SafePCA device and patient: that *drugh* is equal to the result of applying $p2d$ on each element of *physioh*. Using Shankar's notation from [84], the closed (i.e., no non-specified environmental transitions allowed) interpretation of some element P in the context of its environment E is written as $[\![P//E]\!]$. We also adopt a notation where an overbar denotes a truncate if necessary to equalize the lengths of the queues[3]. Further, if the overbar is annotated with a function-name superscript, then that function is applied to each element of the supplied queue, i.e.:

$$\overline{drugh} = \overline{physioh}^{p2d} \triangleq \begin{cases} |physioh| = |drugh| : drugh = \{x \in physioh | p2d(x)\} \\ |physioh| = |drugh| - 1 : drugh = \{x \in dropLast(physioh) | p2d(x)\} \end{cases}$$

[3]We note that this is intuitively similar to Shankar's use of the overbar in [84].

Claim 3. $[\![SafePCASpec // PatientSpec]\!] \vDash$ **invariant** $\overline{drugh} = \overline{physioh}^{p2d}$

Proof Sketch. We establish this claim by induction on the length of $drugh$, denoted $|drugh|$. Note that the specification of the SafePCA device and its environment disallow either to run twice in succession: run writes to $drug$ but requires it to be cleared before it executes again, and $metabolize$ clears $drug$ but requires it to have a value before it executes again.

Base Case ($|drugh| = 0$): This case, which compares two empty queues, is trivially true.

Inductive Case ($|drugh| = n > 0$): Here we split this case into two subcases, depending on the nth value of $drugh$:

- *NoDrug:* If the nth value of $drugh$ is $NoDrug$, then the nth value of $physioh$ is, by inspection of $p2d$, an element of the set $\{Risk, Overdose\}$. Applying $p2d$ to such a value will produce $NoDrug$, and the case is complete.

- *GiveDrug:* This case is virtually identical to the previous case: the nth value of $drugh$ was produced by $p2d$, and so its equalling $GiveDrug$ implies that the nth value of $physioh$ is $Healthy$. The result of applying $p2d$ to such a value will necessarily produce $GiveDrug$.

□

5.4.2 Compositional Approach: App

In the actual PCA interlock scenario, there is no SafePCA device—instead there is some app logic and a standard PCA pump. In this and the next section, we decompose the SafePCA device into these two components (see Figure 5.4 for the app's view of itself in its environment). Then, in Section 5.4.4 we will show that the app logic and a basic PCA pump together uphold the same system-level invariant as the SafePCA device.

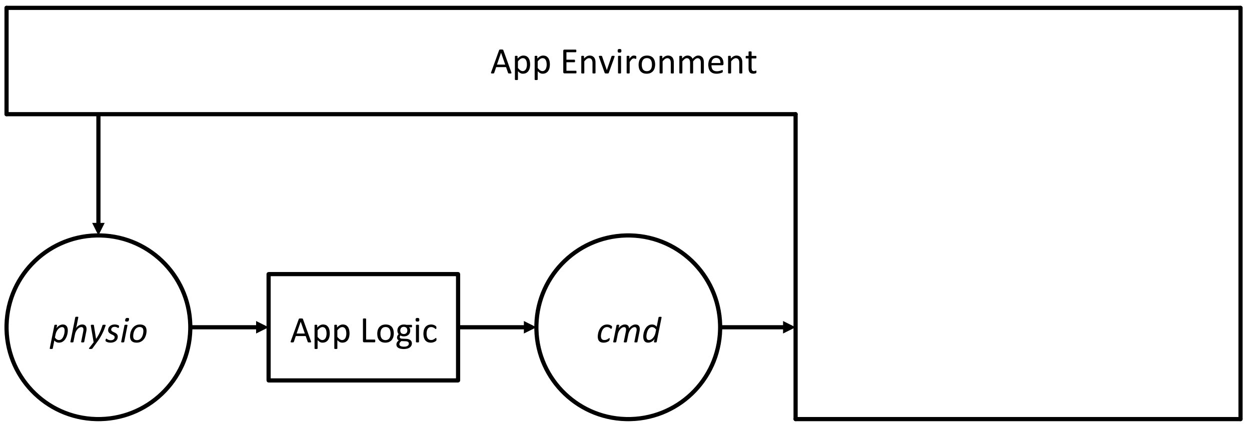


Figure 5.4: *The app logic in its environment, derived from Figure 4 in [84]*

Preliminaries

These components use the two communication and two history variables used by the SafePCA device, and use an additional variable to communicate pump commands from the app to the pump. The values of this command variable can be, according to Definition 5: Abstraction either $CmdPumpEnable$ or $CmdPumpDisable$.

- *cmd*: Command to either enable or disable the pump $cmd \in \{CmdPumpEnable, CmdPumpDisable\}$
- *cmdh* Queue of pump commands

App Description

While the high-level safety goal of the app-and-pump system remains the same as the SafePCA device—to avoid overdosing the patient—the actual undesirability relation (from Definition 3) of the app is in terms of commands to the pump instead of doses of analgesic administered: $(AppState, PhysioVal) \Vdash overdose \iff (AppState = CmdPumpEnable \wedge PhysioVal \leq 25)$. This is because the app logic cannot overdose the patient by itself, only send commands that, if followed by a PCA pump, will cause an overdose.

The Next-State relation Once again we specify the next-state relation and initial state. These are almost identical to the relations used by the SafePCA device, except with commands produced instead of doses of analgesic.

$$execute \triangleq \begin{cases} physio \neq \bot \\ \wedge cmd' = p2c(physio) \\ \wedge physio' = \bot \\ \wedge physioh' = physioh \\ \wedge cmdh' = cmdh \end{cases}$$

execute uses the helper function $p2c(physio)$ which converts a physiological value into a safe pump command:

$$p2c(PhysioVal) \triangleq \begin{cases} PhysioVal = Healthy : CmdPumpEnable \\ PhysioVal \in \{Risk, Overdose\} : CmdPumpDisable \end{cases}$$

The Initial State The app logic starts out by assuming that the patient is not well enough to receive a dose: $init_{App} \triangleq physio = Risk$. Thus the full specification of the app is the combination of the initial state and the next-state relation: $\langle init_{App}, execute \rangle$.

Avoidance of Undesirability The claims and proofs for the app's avoidance of undesirability are virtually identical to the claims and proofs for the SafePCA device, and are not restated here.

App Environment Description

As the app logic is similar to the SafePCA device, so too are its environmental constraints. The app environment's next-state relation and initial state are:

Next-State Relation

$$process \triangleq \begin{cases} physio = \bot \\ \wedge cmdh' = enqueue(cmd, cmdh) \\ \wedge cmd' = \bot \\ \wedge physioh' = enqueue(physio' \in \{Healthy, Risk, Overdose\}, physioh) \end{cases}$$

$$init_{Env} \triangleq (physioh = enqueue(Risk, empty), cmdh = empty)$$

Thus, the full specification of the app's environment is $\langle init_{Env}, process \rangle$.

Analysis

The invariant for the app logic is, once again, virtually identical to the invariant for the SafePCA device: $[\![AppSpec // AppEnvSpec]\!] \vDash \textbf{invariant } \overline{cmdh} = \overline{physioh}^{p2c}$. The proof of the invariant is also virtually identical, with *cmdh* and *p2c* instead of *drugh* and *p2d*. Additionally, the variable that disallows two sequential executions of either *execute* or *process* is *physio*.

5.4.3 Compositional Approach: Pump

Now we examine the second component in the SafePCA's decomposition: a PCA pump. This component takes in commands (as produced by the app logic) and produces doses of analgesic when commanded to (see Figure 5.5). As an actuator, it has a fairly straightforward notion of undesirability: the pump should not run when it is commanded not to: $(PumpState, CmdState) \Vdash overdose \iff (PumpState = GiveDrug \wedge CmdState = CmdPumpDisable)$.

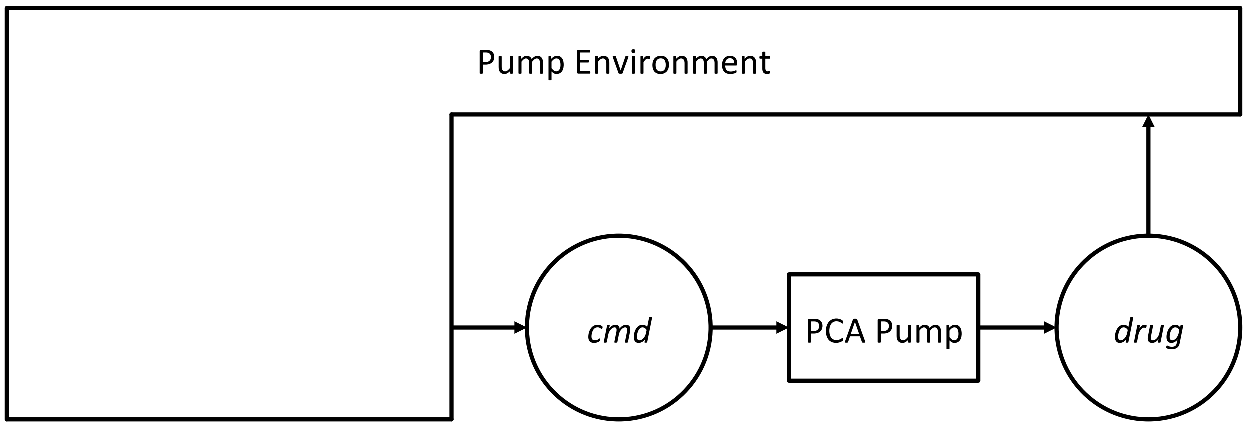


Figure 5.5: *A basic PCA pump in its environment, derived from Figure 4 in [84]*

Pump Description

The pump's next-state relation checks to make sure that there is a command and that the last dose or non-dose of drug has been consumed by the environment. Then, it converts the current command into a dose or non-dose of analgesic.

Next-state relation

$$deliver \triangleq \begin{cases} drug = \bot \\ \wedge cmd \neq \bot \\ \wedge drug' = c2d(cmd) \end{cases}$$

The pump uses an auxiliary function, $c2d$, to convert pump commands into doses (or non-doses) of analgesic:

$$c2d(CmdState) \triangleq \begin{cases} CmdState = CmdPumpDisable : NoDrug \\ CmdState = CmdPumpEnable : GiveDrug \end{cases}$$

Initial state Initially, the pump administers a non-dose: $init_{Pump} \triangleq (drug = NoDrug)$. Thus the full pump specification is: $\langle init_{Pump}, deliver \rangle$.

Avoidance of Undesirability The claims and proofs for the app's avoidance of undesirability are virtually identical to the claims and proofs for the SafePCA device, and are not restated here.

Pump Environment Description

The pump's environment specification is less straightforward than those discussed previously. It is an abstract component that consumes the drug that was administered, and then produces another command. The environment does not dictate which command should be issued (i.e., there is no modeling of the patient's pharmacokinetics), which makes sense as we are using a general purpose PCA pump that simply runs when commanded to do so.

Next-state relation

$$receive \triangleq \begin{cases} drug \neq \bot \\ \wedge drugh' = enqueue(drug, drugh) \\ \wedge cmd' = CmdPumpEnable \vee CmdPumpDisable \\ \wedge cmdh' = enqueue(cmd, cmdh) \end{cases}$$

Initial state The pump's environment begins by initializing the queues, and recording an initial non-dose in the command history (though note that we do not set a value for *cmd* since it is not read before its first write): $init_{PumpEnv} \triangleq (drugh = empty \wedge cmdh = enqueue(CmdPumpDisable, empty))$. Thus the full specification of the pump's environment is: $\langle init_{PumpEnv}, receive \rangle$.

Analysis

The invariant for the pump is similar to those already established, that: $[\![PumpSpec // PumpEnvSpec]\!] \vDash$ **invariant** $\overline{drugh} = \overline{cmdh}^{c2d}$. The pump's logic has two guard variables, *drug* and *cmd*,

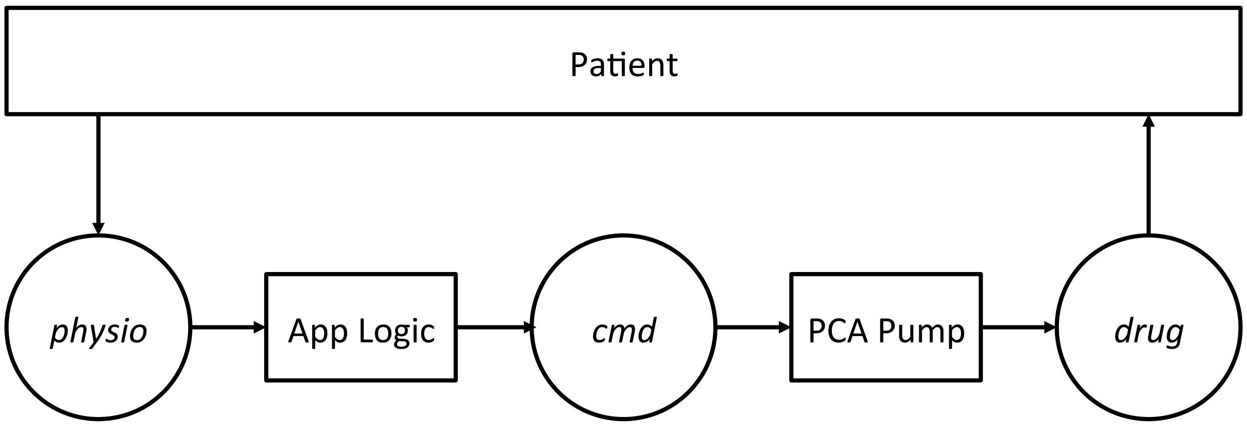


Figure 5.6: *The app logic and pump in their combined environment, derived from Figure 4 in [84]*

which must be unset and set (respectively) for the pump to run, and we note that the relationship between these two elements is more complex than the app or SafePCA device and their respective environments. While *drug* is always unset by the environment, *cmd* cannot actually be unset by the pump's environment. This guard exists, though, to ensure that any other environment avoids calling the pump without a valid command. Each pump action is still necessarily followed by a single environment action, though, so the proof strategy remains virtually identical.

5.4.4 Analyzing the Composed System

Now we consider the combination of the app logic and pca pump components shown graphically in Figure 5.6. We first want to establish that the combination of the two upholds the same invariant as the SafePCA device from Section 5.4.1 and then we must prove that their combination (referred to as the *open co-imposition* of the components) is not inconsistent. That is, we show that it is actually possible to create an environment that simultaneously meets the requirements of the app logic and the pump.

Establishing the Invariant

Having already shown that $[\![SafePCASpec//PatientSpec]\!] \vDash$ **invariant** $\overline{drugh} = \overline{physioh}^{p2d}$, we would like to re-establish it in the context of the composition of our app logic and basic pca pump. We will abbreviate each component in its environment, following Shankar's conventions, to $[\![A^e]\!]$ and $[\![P^e]\!]$ where $[\![A^e]\!] = [\![AppSpec//AppEnvSpec]\!]$ and similarly $[\![P^e]\!] = [\![PumpSpec//PumpEnvSpec]\!]$.

We cannot prove $\vDash [\![(AppSpec \| PumpSpec)//(AppEnv \wedge PumpEnv) \supset A^e$ because the specification of *AppSpec* is not strong enough to be used to prove *PumpEnv* (as, e.g., *AppSpec* places no restrictions on *drug*) and *PumpSpec* is similarly too weak to prove *AppEnv* (as, e.g., *PumpSpec* places no restrictions on *physio*). Thus, we cannot use the invariants established for the app and the pump as global invariants of the composed system.

However, as Shankar points out in a similar example, we can use the open co-imposition of the two systems. The computations of the open co-imposition of some programs P_1 and P_2 with their respective environments E_1 and E_2 "contain actions corresponding to a) P_1 but respecting E_2, b) P_2 but respecting E_1, and c) Environment actions respecting E_1 and E_2." We conclude from his second theorem that the open co-imposition of the two systems: $[\![A^e \times P^e]\!] = [\![((AppSpec \wedge PumpEnvSpec)\|(PumpSpec \wedge AppEnvSpec))//(AppEnvSpec \wedge PumpEnvSpec)]\!]$ implies the conjunction of their invariants: $[\![A^e \times P^e]\!] \supset [\![A^e]\!] \wedge [\![P^e]\!]$, which is strong enough for us to re-establish the original invariant in the system composed of the app logic and pump.

Claim 4. $[\![A^e \times P^e]\!] \vDash$ **invariant** $\overline{drugh} = \overline{physioh}^{p2d}$.

Proof Sketch. This claim is proved with fundamental mathematical properties and Shankar's second theorem. We begin with the two invariants:

1. $[\![A^e \times P^e]\!] \vDash$ **invariant** $\overline{cmdh} = \overline{physioh}^{p2c}$
2. $[\![A^e \times P^e]\!] \vDash$ **invariant** $\overline{drugh} = \overline{cmdh}^{c2d}$

Shankar's second theorem states (in part) that: $\models [\![P_1^e \times P_2^2]\!] \supset [\![P_1^e]\!]$. Shankar also states that the open co-imposition operator commutative, so we know that $\models [\![P_1^e \times P_2^2]\!] \supset [\![P_2^e]\!]$ as well. The conjunction of the invariants, then, is:

$$\overline{cmdh} = \overline{physioh}^{p2c} \wedge \overline{drugh} = \overline{cmdh}^{c2d}$$

Which by substitution can be reduced to:

$$\overline{drugh} = \overline{\overline{physioh}^{p2c}}^{c2d}$$

By inspection of $p2c$, $c2d$, and $p2d$, we can see that $\forall PhysioVal, c2d(p2c(PhysioVal)) = p2d(PhysioVal)$, and so, after simplification, we are left with the desired invariant. □

Establishing Consistency

Consistency may seem intuitive in this case, which is clear from the clear correlation between the observable states of the controller (i.e., $\{CmdPumpEnable, CmdPumpDisable\}$) and those of the pump's environment ($\{ShldGiveDrug, ShldntGiveDrug\}$). However, the details are complex enough that an explicit proof is necessary.

The full expansion of the open co-imposition of our app and pca pump contains three terms (see above). The third term is the conjunction of the environment actions of the two systems, and we note that it is possible to create an abstract component that enforces the specifications of both the app and the pump's environments that does not contain a contradiction. This is only due to the careful specification of the environments, though, and does not follow naturally from the equivalent observable states.

Claim 5. $AppSpec \wedge EnvSpec \not\Longrightarrow \bot$

Proof Sketch. We establish this claim by contradiction. We assume that $AppSpec \wedge EnvSpec \Longrightarrow \bot$, but we show a new environment component, $MetaEnv$, that complies with the specifications of both $AppEnv$ and $PumpEnv$, contradicting the premise.

$$MetaEnv \triangleq \begin{cases} physio = \bot \\ \wedge drug \neq \bot \\ \wedge cmdh' = enqueue(cmd, cmdh) \\ \wedge cmd' = \bot \\ \wedge physioh' = enqueue(physio' \in \{Healthy, Risk, Overdosed\}, physioh) \\ \wedge drugh' = enqueue(drug, drugh) \\ \wedge drug = \bot \end{cases}$$

$$Init_{Meta} \triangleq (physioh = enqueue(Risk, empty) \wedge$$
$$drugh = empty \wedge cmdh = enqueue(CmdPumpDisable, empty))$$

physio and *physioh* are only written to or read from by the app logic's environment. Similarly, the pump's environment modifies *drug* and *drugh*, but the app's doesn't. Thus, their conjunction is trivial. *cmdh* is written to in the same way by both the app and the environment so its conjunction is similarly trivial. *cmd*, however, is something of a special case: the conjunction of the app and the pump's expressions is $cmd' = (CmdPumpDisable \vee CmdPumpEnable) \wedge \bot$ which simplifies to $cmd' = \bot$. □

It is possible for specifications to be inconsistent, however. This would happen in the PCA interlock scenario, for example, if the app and pump's environment specifications were less carefully crafted, e.g., if the pump's environment always set the value of *cmd*, then the resulting conjunction would be unsatisfiable. The potential for inconsistency, though, does not lead to a proof rule to ensure consistency. Shankar writes that "since specifications can be partial, it makes sense to conjoin the environment constraints to the component specification rather than discharge them as proof obligations." He then goes on to explain that "a more detailed implementation will have to satisfy the higher-level specification of the component as well as the constraints on the component imposed by the other components in the combined system." We note that while we do not expect analysts to typically perform this style of reasoning by hand, the existence of multiple levels of refinement is even more rare as the typical refinement path will be simply from specification (derived from some system model) to implementation, obviating the need for intermediate consistency checks.

What's more, decompositions are typically checked lazily when using Shankar's style of reasoning. As he explains, this is because lazy compositional verification is more appropriate "for compositional verification where the point is to achieve a useful decomposition of the verification task," which aligns quite well with the goals we set out with when finding a decomposition technique. That is, we had hoped the hazard analysis process described in this work would allow us to move towards a compositional approach to hazard analysis rather than (the admittedly similar topic of) a specification technique for safety-critical components. Shankar also notes that "assume-guarantee specifications are more appropriate for writing blackbox characterizations of open components" than lazy compositional verification [84].

5.4.5 Refining a Component

In Sections 5.4.2 and 5.4.3 we have discussed decomposing one component specification into two subcomponents. A similar notion is that of *refinement*, where one component can be shown to exhibit all the observable properties of another. A common refinement pattern is showing that an implementation (P) refines a specification (Q): $\models [\![P]\!] \supset [\![Q]\!]$. In this section we present an implementation of the app logic conforms to the specification from Section 5.4.2. The implementation is written in java, and presented here in Figures 5.7 – 5.10.

```java
import java.util.ArrayDeque;
import java.util.Random;

public class AppInEnvironment {

    private enum CmdType { DOSE, NODOSE, UNSET };

    private Integer physio;
    private ArrayDeque<Integer> physioh;
    private CmdType cmd;
    private ArrayDeque<CmdType> cmdh;
    private Object sync = new Object(); // Used for atomic regions
    private AppLogic app; // The app logic itself
    private AppEnvironment env; // The environment's logic
    private Thread appThd, envThd; // Threads for running app and env
    private Random r = new Random(); // Used for bounded random physiovals

    public static void main(String[] args) throws Exception {
        AppInEnvironment aie = new AppInEnvironment();
        Thread.sleep(5000); // Let the threads run for a while
        aie.app.keepRunning = false; // Turn off the app
        aie.env.keepRunning = false; // Turn off the environment
        verifyInvariant(aie); // Check our invariant
    }

    public AppInEnvironment() {
        app = new AppLogic(); // Create app object
        env = new AppEnvironment(); // Create env object
        app.init(); // Enforce app's initialization predicate
        env.init(); // Enforce env's initialization predicate
        appThd = new Thread(app);
        envThd = new Thread(env);
        appThd.start(); // Start the app in its own thread
        envThd.start(); // Start the env in its own thread
    }

    private static CmdType p2c(int physioVal) {
        if (physioVal > 50)
            return CmdType.DOSE;
        else
            return CmdType.NODOSE;
    }
```

Figure 5.7: *An implementation of the preliminaries and setup code that enable an implementation of the specification in Section 5.4.2*

App Logic Implementation

The setup logic, shown in Figure 5.7, consists of the entry point into the logic (`main(String[] args)`

```
44 private static void verifyInvariant(AppInEnvironment aie) throws Exception{
45     // Wait for the threads to terminate
46     while (aie.appThd.isAlive() || aie.envThd.isAlive()) {
47         Thread.sleep(100);
48     }
49
50     // Verify that the queues are roughly the same size
51     if (aie.physioh.size() > aie.cmdh.size() + 1
52             || aie.physioh.size() < aie.cmdh.size() - 1) {
53         assert false : "Queue size difference > 1!";
54     } else if (aie.physioh.size() != aie.cmdh.size()) {
55         // Truncate the last physio value if necessary
56         if (aie.physioh.size() > aie.cmdh.size()) {
57             aie.physioh.removeLast();
58         } else {
59             assert false : "More commands than physiological values!";
60         }
61     }
62
63     // Check the invariant, bail out if there's a mismatch
64     while(!aie.physioh.isEmpty()){
65         if (p2c(aie.physioh.remove()) != aie.cmdh.remove()) {
66             assert false : "Invariant violated!";
67         }
68     }
69 }
```

Figure 5.8: *Java code to check that the invariant from Section 5.4.2 is maintained by the implementation*

a constructor, and the implementation of *p2c*.

The invariant is checked by `verifyInvariant(AppInEnvironment aie)` in Figure 5.8, which uses Java **`assert`** expressions. Note that the truncate function, denoted in Sections 5.4.1 – 5.4.3 by an overbar, is performed by lines 56-57. Of the three **`assert`** expressions, the first two are fail-fast checks for serious problems, while the third **`assert`**, on lines 64-68, is the actual implementation of $\overline{cmdh} = \overline{physioh}^{p2c}$.

The app logic itself is shown in Figure 5.9. The initialization predicate, $init_{App}$, is encoded in `init()`, and *execute* in `run()`. The environment logic is shown in Figure 5.10, and its initialization predicate and next-state relation are implemented in a similar style to the app.

```java
private class AppLogic implements Runnable {
    public boolean keepRunning = true; // Flag to terminate execution

    public void init() {
        physio = 25; // Assume the patient isn't in great shape to start
    }

    @Override
    public void run() {
        while (keepRunning) { // Check that we haven't been told to stop
            synchronized (sync) { // Enter atomic region
                if(physio != null){
                    cmd = p2c(physio); // Get the appropriate command
                    physio = null; // Mark the physio value as consumed
                }
            } // Leave atomic region, but stay in the loop
        }
    }
}
```

Figure 5.9: *An implementation of the app logic that refines the specification from Section 5.4.2*

We verified, to the exhaustion of 8 GiB of memory and more than 48 hours, our implementation's correctness using the Java Pathfinder (JPF) tool [109]. We verified that the program is free of both deadlocks and uncaught exceptions (including the assertions used in our invariant-enforcement method, see Figure 5.8). We created and verified similar implementations of the pump and combined app-pump system as well as their respective environments.

Verifying the Refinement

We would like to claim that our implementation, and in particular the java implementation of *execute* from Figure 5.9, is valid. In order to establish this claim, we will use Shankar's third theorem, which is:

```
private class AppEnvironment implements Runnable {
    public boolean keepRunning = true; // Flag to terminate execution

    public void init() {
        physioh = new ArrayDeque<Integer>(); // Initialize queue
        physioh.add(25); // Record initial patient state
        cmdh = new ArrayDeque<CmdType>(); // Initialize queue
    }

    @Override
    public void run() {
        while (keepRunning) { // Check that we haven't been told to stop
            synchronized (sync) { // Enter atomic region
                if(physio == null){
                    cmdh.add(cmd);
                    cmd = CmdType.UNSET;
                    physio = r.nextInt(99) + 1; // Generate int 0 < x < 101
                    physioh.add(physio); // Record the generated val
                }
            } // Leave atomic region, but stay in the loop
        }
    }
}
```

Figure 5.10: *An implementation of the environment that refines the specification from Section 5.4.2*

$$\frac{\begin{array}{l} [\![P]\!] \vDash \textbf{invariant } r \\ [\![Q]\!] \vDash \textbf{invariant } p \\ \vdash p(s) \land r(s,s') \land N_P(s,s') \supset N_Q(s,s') \\ \vdash I_P(s) \supset I_Q(s) \end{array}}{\vDash [\![P]\!] \supset [\![Q]\!]}$$

Here, P = AppInEnvironment.java and $Q = (AppSpec // AppEnvSpec)$. While Shankar's theorem allows us to use invariants p and r, since our implementation is so directly based on the specification, they end up not being necessary, so we can simply use $\top$. After simplification, then, we must show that the next-state relation of P (i.e., `run()`, from lines

80-92 of Figure 5.9) implies *execute*. It's straightforward to see that it does, though, based on inspection of the two relations. Similarly, we must show that the initialization predicate of the implementation implies that of the specification, but again, this is shown by simple inspection.

We note that there is a corollary of Shankar's third theorem which allows one to show that $\llbracket P \| Q \rrbracket \supset \llbracket (P \wedge E) \| Q \rrbracket$. This requires only a global invariant on the $\llbracket (P \wedge E) \| Q \rrbracket$ term, and can be used to remove the conjoined environments resulting from the open co-imposition operator. That is, had we needed to simplify $\llbracket A^e \times P^e \rrbracket = \llbracket ((\mathit{AppSpec} \wedge \mathit{PumpEnvSpec}) \| (\mathit{PumpSpec} \wedge \mathit{AppEnvSpec})) // (\mathit{AppEnvSpec} \wedge \mathit{PumpEnvSpec}) \rrbracket$ to $\llbracket (\mathit{AppSpec} \| \mathit{PumpSpe}$ $\mathit{PumpEnvSpec}) \rrbracket$ in order to prove Claim 4, we could have used this corollary.

5.5 Fault Propagation and Transformation

In the previous section, we argued that if some components' specifications can be proven to collectively uphold some invariant, then that invariant can be relied upon. This overlooks the obvious problem, though, of the variability of the real world: components fail in myriad ways, and then those failures propagate throughout the system. As discussed in Section 2.2.3, Wallace has proposed the Fault Propagation and Transformation Calculus (FPTC) to reason about these situations and the systemic effects of individual element failure. This section first presents a brief example of the FPTC applied to the PCA Interlock scenario, and then discusses differences between our approach and Wallace's. We conclude by discussing the differences and similarities between Abstract Undesirability from Definition 6 (i.e., $\hat{\boldsymbol{S}}^c_{u,n}$) and some of the error sets produced by the FPTC, and use that discussion to motivate a vocabulary that enables precise discussions of important contrasts between the techniques.

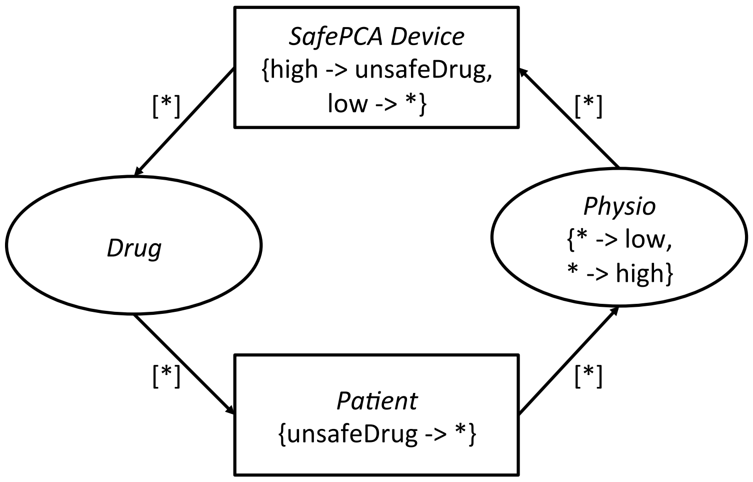


Figure 5.11: *The SafePCA device and environment from Figure 5.3, extended and reformatted to align with Wallace's FPTC. Components are shown as squares, connections as ovals, and the arrows represent the links between components.*

5.5.1 Example System

Figure 5.11 shows the SafePCA and patient from Figure 5.3 after they have been reformatted to align with Wallace's FPTC. This reformatting involves two main steps: First, connections are elevated to the same level as components (a concept discussed in more depth in Section 4.1.5). Then, each element and link is annotated with its failure-related behavior(s). The full syntax of FPTC is not germane to this work, but the behaviors used in elements in Figure 5.11 can be summarized as $\{LHS \rightarrow RHS\}$, where a) the Left-Hand Side (LHS) and Right-Hand Side (RHS) values are specific notions of failure; b) $\rightarrow$ signifies production, i.e., if the component is given the errors listed in the LHS it may produce the errors listed in the RHS; and c) $*$, a special behavior which signifies "no failure."

Thus, the behavior of each component in Figure 5.11 should be interpreted as follows:

- *Patient:* If the patient receives too much drug, they consume it. This does not mean they are unaffected by the failure (indeed, they may be injured or killed), but rather that the error's arrival will not cause the patient to exhibit any other erroneous behavior.

- *Physio:* The physiological readings from the patient may not be transferred between components reliably; i.e., they can be modified to be lower or higher than the actual values. Note that "low" and "high" refer to the reported value's relation to the true quantity, rather than the physiological values themselves.

- *SafePCA Device:* The SafePCA device may "transform" incoming "high" errors into "unsafeDrug" errors, and will consume "low" errors. This aligns with our intuition: if such a device is provided with overly optimistic readings of a patient's health, then it might run when unsafe. If it is given pessimistic values, which would indicate the patient is less healthy than he actually is, then there is no risk of an overdose.

- *Drug:* The connection between the SafePCA device and patient has no logic, so it simply propagates whatever errors come into it.

The links in Figure 5.11 are annotated with sets containing only $\star$, denoting that the system is initialized to include only correct behaviors. From there, Wallace's fixpoint algorithm specifies that possible errors are repeatedly added to each outgoing set until the graph is "stabilized." That is, error "tokens" are added to an element's outgoing link according to the RHS of the element's behavior specification if (and only if) the element's LHS is completely satisfiable given the error tokens on the incoming link.

Which element is selected for evaluation is non-deterministic, though our example from Figure 5.11 is small enough that there is only one possible first step. That step is shown in Figure 5.12. It shows the physiological sensor's production of both improperly high and low values, which illustrates that the FPTC does not determine actual values—since it is impossible for a value to be both too high and too low at the same time—but rather possible errors on outputs.

The improperly high physiological readings will then be transformed by the SafePCA device into unsafe doses of analgesic, or the "unsafeDrug" error. That error will be propagated by the *Drug* connection, which—since it has no explicitly declared behavior—propagates

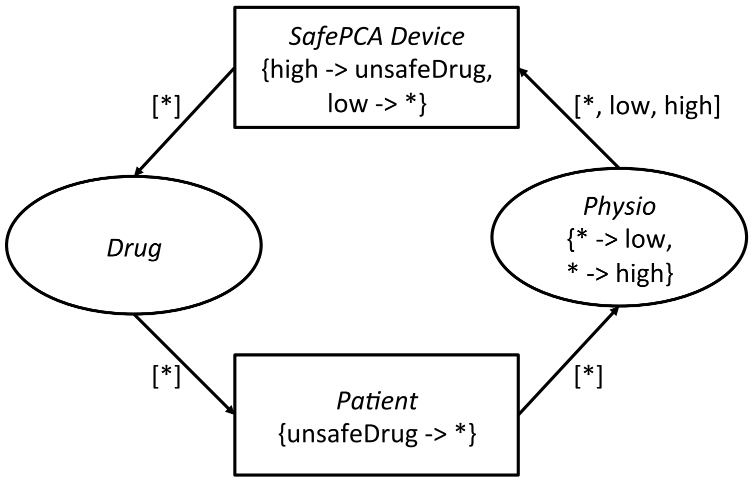


Figure 5.12: *The first step of FPTC on the system from Figure 5.11. The only element that had incoming tokens matching its LHS was the* physio *connection, so tokens matching its behavior's RHSs have been produced on its outgoing link.*

all incoming errors without transformation. The resulting final, stabilized graph is shown in Figure 5.13. Though this system is quite small, is intended to provide a straightforward example of how Wallace's FPTC can be used to determine whether or not a particular error can propagate into, or be propagated out of, a given element.

5.5.2 Differences Found

Though both Wallace's FPTC and SAFE enable component-based arguments for a system's safety, they have three important differences. The first is specific to the formalisms in this chapter, while the second and third are differences between SAFE, as a human-targeted process, and the FPTC, as a calculus composed of a syntax and semantics.

1. *Two-Part Notion of Undesirability:* All of our formalisms center around the two-part concept of undesirability specified in Definition 1. Wallace's formalisms, on the other hand, use the traditional single-state concept with no explicit recognition of the role of an element's environment in a system's overall safety.

2. *Analysis Direction:* The FPTC is a forwards-moving, fixpoint-based analysis that may

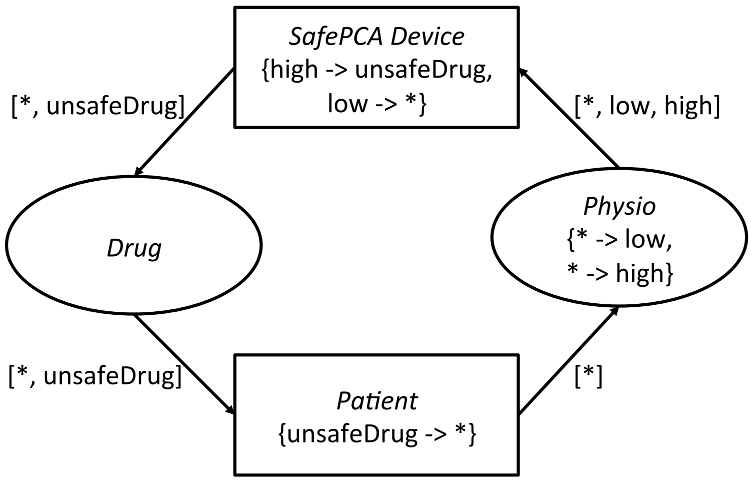


Figure 5.13: *The final version of the FPTC graph first shown in Figure 5.11. The `high` token has been transformed by the SafePCA device into an `unsafeDrug` error, which is then propagated into the patient.*

evaluate an element's behavioral specification a number of times as new errors get added to links. This has the result of identifying only those errors that are actually possible given behavioral descriptions for each system element. SAFE, however, is a backwards-moving process that considers each element only once. Additionally, because it uses our formalisms' notion of undesirability, SAFE is not a true "bottom-up" analysis like the FPTC. Rather, its activities are scoped by the goals of the analyst, which can eliminate consideration faults and errors in a "top-down" style. Ultimately, SAFE is a blend of the two styles of analysis[4].

3. *Additional Annotations:* SAFE enables the specification of a number of annotations beyond error propagations, including human-readable explanations, system and environment state explanations, and detection and mitigation mechanisms. These are not included in the FPTC syntax, though we note that extensions supporting the ideas behind some of these annotations exist. For example, Gallina and Punnekkat's FI^{4}FA extends Wallace's FPTC to include mitigation techniques based on transactions. [111]

[4]There is a similarity, in this regard, between SAFE and Papadopoulos and McDermid's HiP-HOPS technique. [110]

5.5.3 Methodological Discussion and Vocabulary

Differences aside, it seems—intuitively—like there should be some rough equivalence between the FTPC and SAFE/our formalisms in this chapter. That is, both attempt to answer the question "What errors can some particular element cause, and what errors might it receive from other system elements?" In SAFE, this question is explicitly addressed by Activity 1. Wallace, however, presents only a syntax and semantics that can aid in simulating a compositional system as an annotated, directed graph. Once Wallace's fixpoint algorithm has stabilized on the graph, information about a particular element can be determined by looking at its "in" and "out" sets.

Methodological Differences

That said, a particular component's SAFE-calculated successor dangers[5] and its FPTC-calculated "out" set are not equivalent. The former is narrowly focused on the avoidance of particular notions of harm (i.e., undesirability), while the latter is not. Additionally, due to the unification step in the FPTC's semantics, a component's "out" set will not include errors resulting from impossible behaviors: if the error that triggers a particular transformation rule can never arrive at a given component (i.e., the LHS is unsatisfiable), then the error tokens that would be produced by that behavior (i.e., the RHS) will never be propagated. Comparing the benefits and costs of the two methods is less straightforward than might initially be assumed, however, because FPTC and SAFE derive most of their value from different modeling activities. In both techniques, there are essentially two, high-level processes that must be performed:

1. *Modeling:* Both analysis techniques operate on *models* of a system, rather than the system itself. This first step involves creating the values and relations used in the par-

[5]We use terms from Chapter 4 in this section, but we note that a component's successor dangers can be thought of as the collection of element- and environment-state pairings of a particular component, i.e., $(n \in \hat{S}^c_u, x \in \hat{\boldsymbol{S}}^c_{u,n})$.

ticular analysis, i.e., elements, their connection topology, error transformation rules, etc.

2. *Performance:* Once suitable models are in place, the analysis itself can proceed. That can mean calculating the formalisms used in this chapter, or the FPTC's "in" and "out" sets. That is, this step involves the actual execution of the algorithms and calculation of sets, values, and other outputs.

Clearly the second step—*performance*—is fairly prescriptive in both our formalisms and the FPTC. The first step, however, is implicit in Wallace's work, but is vital to producing useful, succinct output. Fully modeling a component's behavior using simple transformation rules is likely to be a time-consuming, error-prone task. Modeling only the "relevant" rules is much more feasible, but a not insignificant challenge lies in determining the relevancy of any particular transformation rule.

SAFE uses a very similar structure for specifying element behavior as the FPTC, where manifestations (and any co-occurring dangers) form the LHS of a component's particular tranformation rule, while the successor danger forms the RHS. As a top-down approach, the methodology proposed in this dissertation leads analysts to avoid modeling irrelevant rules. Both SAFE and the formalisms in this chapter start with a notion to be avoided—safety constraint violations in SAFE and the equivalent notion of undesirability in this chapter—and work backwards. Thus, errors that would not contribute to these notions would not be discovered or documented, and would not clutter up the resulting system model.

Whether or not these errors are truly irrelevant can be difficult to know at system design time, though. Since behaviors that are irrelevant to one safety constraint may be necessary to discover errors that could lead to the violation of a different one, an argument can be made for modeling as much of an element's error-related behavior as possible. This is particularly true in cases where the top-level system safety goals are unknown, or are at least difficult to forecast with an acceptable level of certainty.

A Vocabulary for Comparing Hazard Analysis Assessment Techniques

Ultimately, we cannot draw any firm conclusions on the various differences between SAFE and the FPTC. However, there are a number of terms that we believe are important to future discussions in this area, and so present a vocabulary which will allow for clearer distinctions to be made between SAFE, the FPTC, and other safety-argumentation strategies. There are four key concepts; each exists as a spectrum between two extremes.

1. *Top-Down versus Bottom-Up Styles:* This concept speaks to the direction that the analysis moves in, which is also sometimes phrased as "cause to effect" and "effect to cause," as in, e.g., the HiP-HOPS work [110]. A top-down analysis, like FTA, STPA, or SAFE, starts with a notion of what the analyst would like to avoid: e.g., an undesirable event's occurrence, the violation of a safety constraint, etc. A bottom-up analysis, on the other hand, considers first the ways that a particular element can fail and then the effects of those failures on the rest of the system. Top-down analyses typically move backwards through a system, while bottom-up analyses, like FMEA or the FPTC, are forwards-moving.

2. *Irrelevant versus Impossible Errors:* An error that is potentially exhibited by a particular element may not affect the desired qualities of a system. The error might impact some irrelevant system characteristic (i.e., performance in a non-realtime domain) or its occurrence might be impossible in a real-world version of the system. Top-down analyses like SAFE excel at detecting the first class of errors, since they begin with activities that scope the analysis to problems the analyst would like to focus on. Bottom-up analyses like the FPTC excel at focusing on errors that can actually occur, though, since the inputs to an element are known when its analysis begins. Thus, output errors requiring infeasible inputs can be safely disregarded, since their occurrence is impossible.

3. *Causal versus Caused Errors:* There are essentially two sets of errors that are relevant

to a particular element: those that arrive at the element (i.e., manifestations in SAFE, the FPTC's "in" set) and those that propagate out of it (i.e., successor dangers in SAFE, the FPTC's "out" set). We term the former *causal* errors and the latter *caused*. This distinction exists in most architecture-centric safety argumentation strategies. It also exists in some techniques from the formal methods community: we note the correspondence between causal/caused errors and backwards/forwards slicing of an element's impact in a system's control- or data-flow dependency graph.

4. *Modeling versus Performance Activities:* As discussed in the first part of Section 5.5.3, all safety argumentation strategies have two high-level parts. Often, one of these parts is implicit, i.e., in the FPTC, system modeling is not discussed; in STPA, there is little discussion of how exactly to determine the effects of errors in a given system element.

5.6 Gaps in the Analysis

While we believe that the work in this chapter is illustrative of a potentially fruitful approach to reconciling the work of the hazard analysis and formal methods communities, we recognize that by no means does it completely work out the foundations necessary for such an undertaking. Three tasks stand out as being necessary to enable the full use of the material in this section:

1. *A Full, Coherent Formalization:* In order to speak concretely about a system, we would need a full, end-to-end formalization in place. We believe that our current collection of definitions from Section 5.3 is potentially useful, but we recognize that its utility is distributed throughout a number of definitions rather than realized in a single cogent theory.

2. *A Model of Computation:* As part of 1., our formalizations particularly require a model of computation. We note that in Section 5.4 we used asynchronous transition systems

(as that was the model of computation used by Shankar [84]) but more investigation is needed to determine if that model is the most appropriate for the formalisms from Section 5.3.

3. *Justifications for / Reconciliations of Previous Assumptions:* In the introduction to Section 5.4, we noted a number of simplifying assumptions. While we believe that those assumptions were warranted for our purposes, a full formalization would need to either justify them or obviate their necessity.

   (a) *A Fully Formalized Platform:* As MAP apps rely heavily on the guarantees provided by the platform itself, formalizing its behaviors is required. An ideal formalization would be one that does not have to be tightly integrated with an app's specification, so that properties of the composed system's behavior on different platforms could be more easily verified.

   (b) *Notions of Timing:* As we discussed in the introduction to Section 5.4, our formalization there did not include a notion of timing. Properties of real-world MAP apps are typically stated as timings, however (see, e.g., the quality-of-service properties in Table 3.2) so an ideal formalization would be in a logic that supports notions of timing, e.g., timed finite automata. [112]

   (c) *Communication via Shared Variables:* In Section 5.4, we adopted the model of communication used by Shankar, which is via shared state. Since MAPs are distributed, however, and many—like the MDCF—use publish-subscribe architectures (see [73]), an ideal formalization would need to have a notion of asynchronous, port-based communication.

A full formalization of the hazard-analysis view of system safety would be extremely useful, not only to improving techniques like SAFE, but to broadening the applicability of formal methods-based approaches as well. We believe that the definitions in this chapter, subject to the assumptions and caveats stated above, represent a promising possible approach to such a formalization.

Chapter 6

Evaluation

In this chapter we present an evaluation of both the manual and tool assisted versions of our new hazard analysis technique. The evaluation is presented as the findings produced by our analysis when it is applied to the PCA interlock scenario. Recognizing that the subjectivity of our analysis damages our confidence in it, we also propose a collection of user studies which we would perform in the ideal case.

6.1 Analysis of the PCA Interlock System

In this section, we discuss the results of the analysis of the central control loop in the PCA interlock system, shown in Figure 6.1, including both issues found previously by other researchers and newly identified problems. The full report, including analysis of each element in the loop, is available in Appendix C.

6.1.1 Previously Discovered Issues

As the PCA interlock scenario has been studied extensively, one useful activity is discussing the extent to which our analysis rediscovers problems that have been found by other researchers. The novelty and benefit of the technique described in this dissertation stems

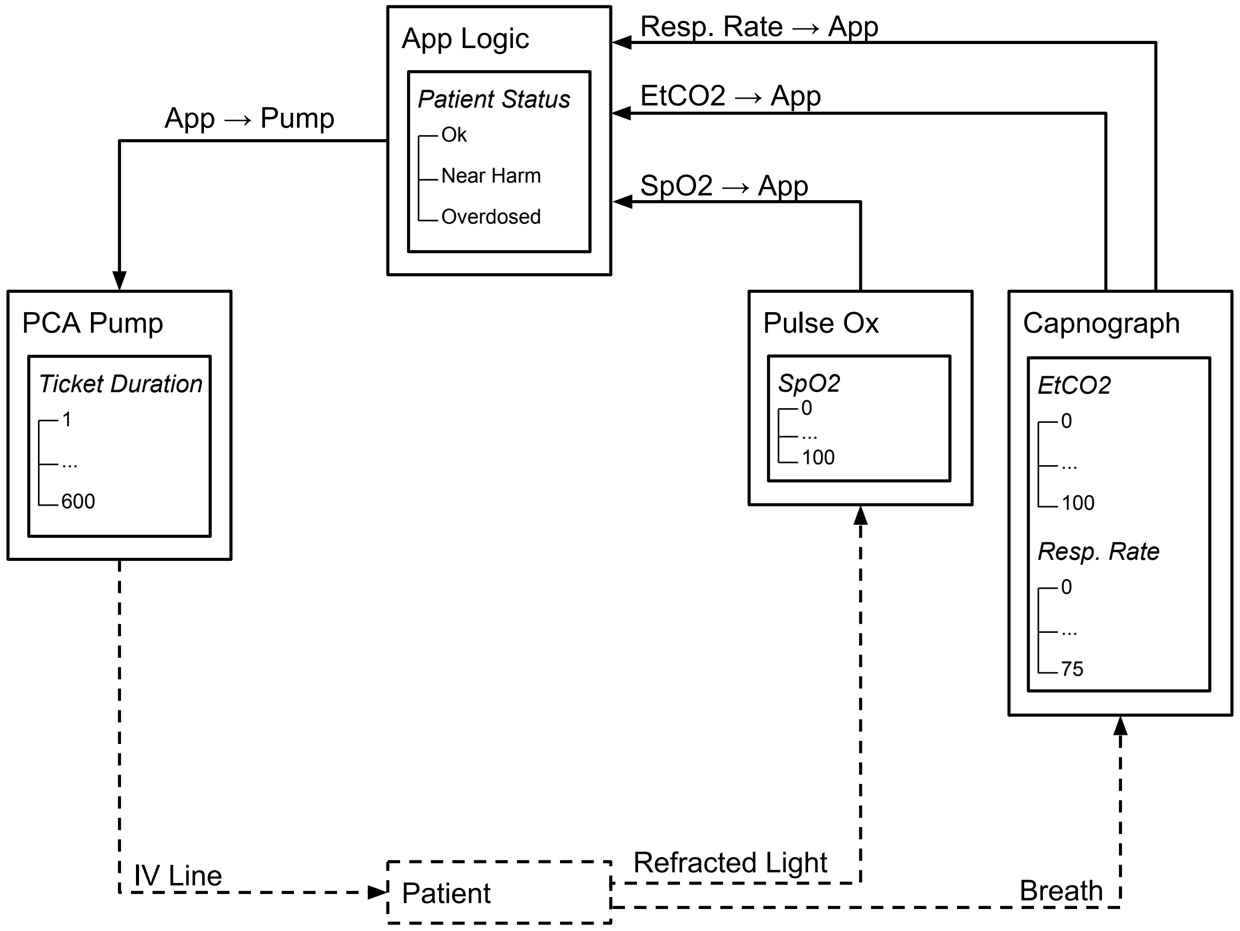


Figure 6.1: *The inner control loop of the PCA Interlock scenario*

less from an analyst's ability to uncover problems that have never been considered before, and more from his or her ability to (comparatively) easily and quickly uncover an extremely broad range of problems that are typically not revealed by a single previously existing hazard analysis technique or mode of thinking. The first two findings addressed here are primarily issues of ensuring correct timing, and the latter four are more focused on security concerns.

Ticket-Based Architecture

The need for "tickets" which enable the PCA pump to run for some window of time, rather than with simple on/off commands, was first suggested by Arney et al. in [27]. The process described in Section 4 would discover the same solution when performing Step 1.2 on the PCA pump. More specifically, assuming an architecture with an event-based channel (where events toggle the state of the pump, from on-to-off or vice-versa), when considering the “Halted” manifestation, an analyst would note that this could leave the pump on for an inappropriate length of time. This problem could be detected at runtime (i.e., using “Concurrent Detection” in the language of [3]) by using a ticket-based architecture.

Quality-of-Service Enforcement

It is generally understood that the correctness of real-time systems depends not only calculating correct values, but also on delivering those values within acceptable time ranges. The need for a MAP to support guarantees of these timing constraints has been discussed for some time; an excellent example is King et al.'s [113]. Our hazard analysis process would discover this requirement in a number of places as the analyst is asked, in each component and connection, to consider the case where incoming messages arrive later than they should. The only way to detect these late messages is to ask developers to specify quality-of-service requirements, and the only way to ensure those requirements are met in a semi-open system like a MAP is to have the platform itself guarantee their enforcement, using a technology like MIDAS [19].

Access Control

Controlling access to MAP apps and devices can be done in a number of ways, ranging from physical security to software-enforced role-based access control (RBAC). Salazar explains the need for, and provides a MDCF-based design of, RBAC in [114]. Our analysis would reveal the need for access control in step 2 (which focuses on faults not caused by other components), with guidewords 10, 12, 13 and 14. All four of these guidewords involve either inadvertent or malicious actions, and they have listed—as typical compensation—access control.

Device Certification and Authentication

While the safe behavior of all elements of a MAP is the goal of the analysis, there is a risk that the end-user of a given app or device may attempt to use a device that has not actually passed any sort of hazard analysis. A very similar concern is that of device authentication, i.e., the need for some assurance that a MAP element is what it claims to be. Salazar argues that existing certification mechanisms can be strengthened considerably by using machine-readable cryptographic certificates that attest to an element's identity and its having passed certification. He motivates, designs, and evaluates a certificate framework for the MDCF that would ensure a device's authenticity as well as its certification by some trusted authority [114].

This is, to some extent, outside the scope of our hazard analysis and would instead be covered by an element's "lifecycle" as addressed by, e.g., IEC 62304 (see Section 2.2.5). That said, in order to prevent compromised hardware and software (guidewords 3 and 4), our process recommends enforcing a chain-of-trust, which could be cryptographic in nature.

Communication Security

Salazar also discusses the need for, implementation of, and design of cryptographic security for inter-element communication in the MDCF [115, 114]. This would be addressed by guidewords 12 and 13 of our process, which suggest—in addition to access control and physical security—cyptography as a possible compensatory measure.

6.1.2 Newly Discovered Issues

There were two potential improvements suggested by our analysis of the PCA interlock system that—to the best of our knowledge—have not been directly addressed previously in the literature. The first improvement is focused on the algorithm itself and the second on the user interface of the app's display and devices. These improvements are not oversights of any of the previous analyses of the PCA interlock app because they are beyond the scope of the previous analyses. Instead, these are dangers that could come about in an app implementation, even if that implementation successfully incorporated the findings of all previous research on the scenario.

Statistical Inference

In Arney et al.'s original work on a ticket-based PCA interlock architecture, the duration of the ticket sent to the PCA pump is to be determined according to a pharmacokinetic model of a typical patient's tolerance for opioid analgesic [27]. Using this model (Arney et al. cite [116] as an example) and the current state of the the values of the physiological parameters (i.e., SpO_2, $ETCO_2$, respiratory rate, and pulse rate), a safe window of time can be calculated.

Arney et al. explain that they "are not aware of a unified model that captures the whole process and is appropriate for the control-theoretic study of the closed loop dynamics," and discuss options for control if such a model were to become available. In the interim, though, we argue that it would be safer for algorithm designers who are using statistical models of "typical" patient pharmacokinetics ([116] uses a 95% confidence interval) to instead use an algorithm that assumes very conservative patient dynamics. Then, in response to physiological inputs the algorithm should use, e.g., Bayesian inference to drift towards more typical values after it has been established that the current patient is not an outlier. That is, instead of assuming a patient has a standard tolerance for analgesic, the PCA interlock app logic should assume that opioid will have an outsize effect and only allow very small doses. Once it has been established (via some sort of statistical inference) that the dose and response are in line with that of a typical patient, the app can enable typical (i.e., larger) doses.

This danger was detected via a careful reading of Arney et al.'s work and its citations [27, 116]. It would be detected by step 2.2 of the hazard analysis process, when considering guideword 2: "Bad Software Design." This problem is exactly what Avižienis et al. discuss when they describe “deliberate, nonmalicious development faults” in that the fault is simply the result of a development choice—to assume typical patient pharmacokinetics—that, eventually, would cause the hazard: the overadministration of analgesic [3]. While the design of such an algorithm is beyond the scope of this work, we recognize that our process has acheived its goal simply by detecting the causal scenario.

Thoughtful User Interface Design

Guidewords 11-14 in step 2.2 of SAFE ask the analyst to consider the result of an operator making either a mistake, or an intentional—but incorrect—choice when using the hardware or software of a system. This points to the need for thoughtful user inteface (UI) design that considers two separate factors: usability and awareness of system state; these factors are discussed in some depth in Section 9.4 of [30].

Usability The design of PCA pumps—only one component of the PCA interlock app—to avoid user mistakes is itself a topic of study [98]. Careful analysis of the app’s display components would also be warranted, and would be useful in increasing the overall safety of the app. Leveson gives some specific guidance on this topic in Section 9.4.6 of [30].

Awareness of System State A second concern, which seeks to avoid incorrect, intentional actions is UI design that makes the operator aware of the effects of her actions. Leveson addresses this topic specifically in Section 9.4.7 of [30], and names four potential problems including “mode confusion,” where an operator thinks the system is in one mode when it is in fact in another. Careful design to show the operator the potential effects of her actions is vital for the overall system safety.

6.1.3 Threats to Validity

The most significant threat to the findings discussed in this section is that the technique has been primarily developed using the PCA interlock scenario as a running example. While we have no doubt benefited from the deep consideration given the scenario by various researchers, we have also had to consciously work against fitting the analysis too closely to the scenario. That is, there is a risk that we only caught the five previously-discovered issues discussed in this section because our analysis was inadvertently designed to focus on those exact issues. Despite the efforts made to generalize the process, we would have more confidence in our findings, and they would be more broadly applicable, if we had more case studies or, at a minimum, a different MAP app than had been used in the process's development.

6.2 Proposed User Study

We note that there are two techniques that could be used to evaluate the hazard analysis presented in this work. The first, which we have discussed in the previous section, is a discussion of our experiences applying the technique to the PCA Interlock system. The second, and ultimately preferable, technique would be a user study, which we describe here.

6.2.1 Methodology

We propose providing some number of volunteers with a narrative description of a system (e.g., the PCA interlock main control loop examined previously) as well as training on a hazard analysis technique. That is, the independent variable in this study would be the type of analysis training is provided for, e.g., FTA, FMEA, STPA, or the new hazard analysis process described in this work. Then, the dependent variables would be:

1. *Analytic Power (Quantitative):* The number of faults and errors uncovered; discovery of higher numbers of faults and errors would be better.

2. *Analytic Power (Qualitative):* The distribution of faults and errors uncovered, where a more broad distribution across each category is better. Axes of distribution would be seven of the elementary fault classes[1] and the five service failure modes identified in [3]:

    (a) *Phase of Creation:* Faults that occur in development versus those occurring when the system is operating.

    (b) *System Boundaries:* Faults internal to the system versus those outside the system's boundary.

    (c) *Phenomenonological Cause:* Faults made by humans versus those resulting from natural causes.

    (d) *Dimension:* Faults originating in hardware versus those originating in software.

    (e) *Objective:* Malicious versus non-malicious faults.

    (f) *Intent:* Deliberate versus non-deliberate faults.

    (g) *Persistence:* Permanent versus transient faults.

    (h) *Content Errors:* Input that arrived at the correct time, but had an incorrect value.

    (i) *Early Timing Errors:* Input that had the correct value but arrived too early.

    (j) *Late Timing Errors:* Input that had the correct value but arrived too late.

    (k) *Halted Service Errors:* A complete lack of input.

    (l) *Erratic Service Errors:* Input that arrives at the wrong time with values that are incorrect.

[1]Note that Avižienis et al. identify an eighth fault class—Capability—to distinguish between accidental faults and those resulting from incompetence. This is done to aid in the determining of blame, but we agree with Leveson who writes in [30] that "Blame is the enemy of safety. Focus should be on understanding how the system behavior as a whole contributed to the loss and not on who or what to blame for it."

	Analytic Power (Quantitative)	Phase of Creation	System Boundaries	Phenomenological Cause	Dimension	Objective	Intent	Persistence	Content	Early Timing	Late Timing	Halted Service	Erratic Service	Time Required	Complexity
		Analytic Power (Qualitative)													
FMEA	−−−	−−	−−	−	−−	−−	−−	*	−	*	*	*	*	*	+
EMv2 FMEA	−−	−−	−−	−	−−	−−	−−	*	−	*	*	*	*	+	++
FTA	−−	−	−	*	−	−	−	*	−	−	*	*	*	−	*
STPA	−	−	+	*	−	−	*	*	*	*	*	*	*	+	+
Manual HA	*	*	*	*	*	*	*	*	*	*	*	*	*	*	*
Tool HA	*	*	*	*	*	*	*	*	*	*	*	*	*	+	*

Table 6.1: *The hypothesized results of the proposed user study as relative estimates*

3. *Time Required:* The time elapsed between when the system description is provided and when the user declares that the analysis is complete; less time required is better.

4. *Complexity:* A subjective evaluation of how hard it is to perform the technique correctly; less complexity is better.

6.2.2 Hypothesis

Hypothesized results are presented in Table 6.1 (using the same notation as Table 4.7). Rather than attempt to estimate precise values, we position the hazard analysis techniques relative to one another, using M-SAFE as a baseline.

6.2.3 Threats to Validity

Though this study would be of considerably higher quality than the case study evaluation provided in Section 6.1, one significant threat to its validity is that the evaluation is tied to Avižienis et al.'s taxonomy, which has also used to create our analysis technique. The results would be stronger if there were a different—but similarly powerful—classification of faults and errors available, but we are not aware of one.

6.2.4 Further Studies

After performing the initial study, we note that there are several other possible studies that would yield interesting results. These variants would reuse the methodology sketched in Section 6.2.1 but would add additional independent variables. The dependent variables would remain largely the same, though to save time/effort, analysis of the qualitative variables could be skipped.

- *Skill Required:* Here the amount of training provided to the users would vary (in addition to the analysis techniques), and the resulting effects on the analytic power would be particularly interesting. We would hypothesize that the impact of reduced training would be more substantial on techniques like STPA and FTA than on the more rigidly-defined processes like SAFE or FMEA.

- *Domain:* Here we would vary the domains from which the system descriptions are gathered. That is, different groups of users would have descriptions of, e.g., automotive, avionics, and/or medical systems. We would hypothesize that FMEA would require less time when analyzing hardware-dominated domains, and our process would work better than others in domains with looser coupling between components.

- *Phase:* Here the system descriptions provided to the users would vary in terms of development phase, i.e., some users could have conceptual, preliminary, or detailed system designs. While we hypothesize that all techniques would show more analytic power on more detailed system descriptions, we also believe that flexible analyses like STPA and FTA would show more power than FMEA and our process when analyzing more preliminary designs. Further, we believe that the tool-supported processes like EMv2-derived FMEA and the tool-supported version of our process would be nearly unworkable.

One final study that would be particularly useful would be to compare the results of SAFE-trained analysts not against otherwise-inexperienced volunteers trained in other hazard analysis techniques but against experts instead. Positive results from this study, which is arguably biased against SAFE, may be more convincing to experts who are well-versed in STPA, FMEA, FTA, or other more well-established hazard analysis techniques.

Chapter 7

Future Work and Conclusions

In this section we enumerate possible next steps for the three technical contributions discussed in this dissertation, and then provide concluding remarks. This section can be thought of as the results of performing an informal "gap analysis," i.e., what desiderata implied by our original goal of an application development environment for MAP apps are not included in this work?

7.1 Future Work

7.1.1 MDCF Architect

There are three primary directions in which work on the MDCF Architect might proceed: expanding the input language, retargeting the code generator, or enhancing the tooling with additional verification activities.

Expanding the Modeling Language

Despite the fact that the subset used by the MDCF Architect (which is the subject of Chapter 3) is narrowly tailored to the MAP domain, we believe there is still room for expansion.

One possibility is to support richer data type modeling, which could enable features like aggregate data types. It is likely that developers of real-world medical devices will want to bundle physiological readings with metadata like timestamps or patient data, and our language should be able to support that. A second possibility would be to support the modeling of component *modes* as high-level descriptors of the state an app might be in, e.g., "initializing," "self-maintenance," or "operating." These features are already supported in the full AADL, so expanding our treated subset to include them should be fairly straightforward.

Many features that are already modelable in AADL are difficult to implement in Java and XML, i.e., data flow restrictions such as in, out, or in/out method parameters. Other possible languages could include those designed for safety-critical systems (e.g., SPARK [117]); languages designed for high-reliability distributed systems (e.g., Erlang [118]); or a purpose-built, domain-specific language. Such a language should support features like pattern matching, a well-defined real-time computational model, and ease of verification/analysis.

Retargeting the Code Generator

Additional MAPs exist, like those built by Draeger or DocBox [23]. It would be highly beneficial to app and device developers if components could be specified in a single common language and then retargeted automatically to other MAPs, and we believe that the MDCF Architect provides a suitable platform for this task. Though code generation currently only supports the MDCF, the translator could be retargeted to these or other MAP implementations, with the goal of helping the different device and app developers in the ICE ecosystem move towards a common model of computation and communication. This would potentially enable the significant code and analysis reuse that the MAP vision relies on.

Additional Verification Capabilities

Since so much information regarding a component is specifiable in the MDCF Architect's language, it seems natural to evaluate additional verification techniques. While the MDCF

Architect should be extended to support a number of different testing techniques, one particularly promising feature is fault injection. Developers already specify their component's input and output message types and rates of delivery, as well as information regarding the effects of incoming errors (by using the T-SAFE-supported EMV2 annotations). It seems natural to build software tooling that could verify these claims by injecting generated faults into a component and then ensuring that the effects on the component's output are as stated in its specification.

Other verification techniques might require extensions to the input language to allow specification of *contracts*. Software contracts enable assertions of various desired system properties to be generated and checked automatically, and they have considerable benefits for system development [103]. Larson et al. have implemented a sophisticated contract specification and verification tool on top of AADL called BLESS; future work should consider its use in the specification of MAP app behavior and properties [119].

7.1.2 The SAFE Process

As discussed in Section 6.2, a full user study is needed to completely evaluate the claims made in this work regarding SAFE's quality. Beyond this test, though, future work should examine how SAFE enables (or impedes) analyses of various system desiderata. One particularly interesting aspect of SAFE is its ability to neatly consider both safety and security concerns simultaneously in both Activity 1 and Activity 2. This is particularly useful given ongoing challenges in uniting the fields; both traditional safety analyses (i.e., FMEA and FTA) as well as more modern ones (i.e., STPA) do not explicitly consider designing systems to be more secure against malevolent adversaries.

SAFE and Security: Activity 1

The compression of possible incoming errors into the failure domains identified by Avižienis et al. aligns very well with concepts that exist in the security domain. Indeed, Dolev and

Yao's well-established threat model is in many ways a recognition of similar styles of failure from an adversarial point of view [93]. We believe that a full mapping between the two is possible, and would like to establish the equivalence of the underlying ideas.

SAFE and Security: Activity 2

There are four fault classes used in SAFE which speak directly to the activities of a malevolent adversary (fault classes 3, 4, 12, and 13 in Table 4.1). These fault classes can be further refined based on a particular attacker model, and recommended compensatory actions can be refined as well. For example, consider fault class 13 which describes the situation where an adversary is able to gain access to one or more of the system's software elements at runtime. If we assume an attacker that is capable only of reading the state of the software, then encrypting any private data may suffice to compensate for the fault. If, however, we assume a more capable adversary that can forge and re-send commands, then more expensive compensations—like cryptographic chains-of-trust and authentication—may be required.

Retargeting the Report Generator

Though we believe the SAFE report format discussed in Chapter 4 is a reasonable hazard analysis format, we recognize that hazard analyses are often used as components of structured arguments of a system's overall safety. These arguments are referred to as *assurance cases*, or sometimes *safety cases* (see Section 2.2.4). It would be useful to generate as much of these cases as is possible using existing T-SAFE annotations. It would also be an interesting research task to examine the overlaps between the notions considered by T-SAFE and those by existing assurance case tooling, and to determine what benefits can be derived from programmatic integrations between the two.

7.1.3 Theoretical Work

Obvious next steps for the theoretical aspects of this work, as specified in Chapter 5, are—in addition to those discussed in Section 5.6—to a) examine its suitability and value when applied to more examples, and b) investigate if the use of assume-guarantee reasoning—instead of Shankar's "lazy compositional" style—would bring any benefits to the compositional aspects of this work. In the longer-term, though, there are two other tasks that we see as particularly interesting: formalizing the semantics of the guidewords we use in SAFE's first and second steps, and linking our work to a more well-defined computational model.

Formalization of Guidewords

The guidewords used in identifying manifestations in SAFE's Activity 1 have no formal definitions. While producing these definitions may be straightforward in some cases, i.e., "Early" or "Value low," others, such as "Erratic," may not be formalizable. Work in this area should likely hew closely to the definitions used in AADL's EMV2 standard, since it contains formalizations of the errors used in its library [33]. Many of those overlap (or refine) the guidewords used by SAFE. Formalizing the fault classes used in Activity 2 will be more challenging, and will likely require a much more well-defined model of computation.

A Formal Computational Model

The formalizations from Chapter 5 rely on an implicit computational model, but making that model explicit would be useful for a number of reasons. Just as we had to adopt the asynchronous transition formalism in order to decompose the analysis task in Section 5.4, so too would we need a computational model for, e.g.: connecting to assume-guarantee reasoning, analyzing "gaps" in SAFE (or other hazard analysis techniques, see Section 5.5.3), or formalizing the definition of the fault classes used in SAFE's Activity 2.

7.2 Concluding Remarks

In this work we have presented three technical aspects which collectively enable a prototype development environment for MAP applications. First, in Chapter 3, we described an input language and code generator for applications that run on the MDCF. Second, in Chapter 4, we described a semi-compositional, component-oriented hazard analysis technique for evaluating the safety-related aspects of these and other safety-critical applications. Third, in Chapter 5, we presented a collection of mathematical definitions derived from a concept grounded in the system safety community and then suggested potential connections between it and two important formalisms from the formal methods community. We believe that this work has advanced the state of the art in this area, and—using the ideas in this dissertation as a stepping-stone—has enabled a number of exciting avenues for future work.

Bibliography

[1] Y. J. Kim, S. Procter, V.-P. Ranganath, J. Hatcliff, and Robby, "Stakeholders in ICE Ecosphere," in *Healthcare Informatics (ICHI), 2015 IEEE International Conference on*, 2015.

[2] ASTM International, "ASTM F2761: Medical Devices and Medical Systems – Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE)," ASTM International, West Conshohocken, PA, 2009. [Online]. Available: www.astm.org

[3] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic Concepts and Taxonomy of Dependable and Secure Computing," *Dependable and Secure Computing, IEEE Transactions on*, vol. 1, no. 1, pp. 11–33, 2004.

[4] S. Procter and J. Hatcliff, "An architecturally-integrated, systems-based hazard analysis for medical applications," in *Formal Methods and Models for Codesign (MEMOCODE), 2014 Twelfth ACM/IEEE International Conference on*, Oct 2014.

[5] J. Rushby, "Understanding and Evaluating Assurance Cases," SRI International, Computer Science Laboratory, SRI International, Menlo Park CA 94025, USA, Tech. Rep. SRI-CSL-15-01, 2015.

[6] S. Procter, J. Hatcliff, S. Weininger, and A. Fernando, "Error type refinement for assurance of families of platform-based systems," in *Computer Safety, Reliability, and Security*. Springer International Publishing, 2015, pp. 95–106.

[7] C. A. Ericson II, *Hazard analysis techniques for system safety*. John Wiley & Sons, 2005.

[8] J. C. Knight, "Safety critical systems: challenges and directions," in *Software Engineering, 2002. ICSE 2002. Proceedings of the 24rd International Conference on.* IEEE, 2002, pp. 547–550.

[9] N. Leveson, "Perspective: The Drawbacks in Using The Term 'System of Systems'," *Biomedical Instrumentation & Technology*, vol. 47, no. 2, pp. 115–118, 2013.

[10] P. H. Feiler, "Model-based validation of safety-critical embedded systems," in *Aerospace Conference, 2010 IEEE.* IEEE, 2010, pp. 1–10.

[11] P. Clements, F. Bachmann, L. Bass, D. Garlan, J. Ivers, R. Little, P. Merson, R. Nord, and J. Stafford, *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2010.

[12] E. Phelps and J. Goldman, "Automated situational analysis for operating room anesthesia monitoring," *Biomedical Sciences Instrumentation*, vol. 28, pp. 111–116, 1991.

[13] J. Hatcliff, A. King, I. Lee, A. MacDonald, A. Fernando, M. Robkin, E. Vasserman, S. Weininger, and J. M. Goldman, "Rationale and architecture principles for medical application platforms," in *Cyber-Physical Systems (ICCPS), 2012 IEEE/ACM Third International Conference on.* IEEE, 2012, pp. 3–12.

[14] P. Checkland, *Systems thinking, systems practice.* John Wiley & Sons, 1981.

[15] P. J. Prisaznuk, "Integrated modular avionics," in *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference, 1992. (NAECON)*, vol. 1, May 1992, pp. 39–45.

[16] J. Rushby, "Separation and Integration in MILS (The MILS Constitution)," SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park CA 94025, USA, Tech. Rep. SRI-CSL-08-XX, 2008.

[17] A. L. King, L. Feng, S. Procter, S. Chen, O. Sokolsky, J. Hatcliff, and I. Lee, "Towards Assurance for Plug & Play Medical Systems," in *Computer Safety, Reliability, and Security.* Springer, 2015.

[18] A. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. Jetley, P. Jones, and S. Weininger, "An open test bed for medical device integration and coordination," in *Proceedings of the 31st International Conference on Software Engineering*, 2009.

[19] Andrew King and Sanjian Chen and Insup Lee, "The MIDdleware Assurance Substrate: Enabling Strong Real-Time Guarantees in Open Systems With OpenFlow," in *17th IEEE Computer Society Symposium on object/component/service-oriented real-time distributed computing (ISORC 2014)*, 2014.

[20] M. D. Petty and E. W. Weisel, "A composability lexicon," in *Proceedings of the Spring 2003 Simulation Interoperability Workshop*, 2003, pp. 181–187.

[21] M. Hapner, R. Burridge, R. Sharma, J. Fialli, and K. Stout, "Java message service," Sun Microsystems, Inc., Tech. Rep. 1.1, April 2002.

[22] G. Pardo-Castellote, "OMG Data-Distribution Service: Architectural Overview," in *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on.* IEEE, 2003, pp. 200–206.

[23] S. Schlichting and S. Pöhlsen, "An architecture for distributed systems of medical devices in high acuity environments," Dräger, Tech. Rep., 2014.

[24] ISO/IEEE, "ISO/IEEE11073-10101 Health informatics – Point-of-care medical device communication - Nomenclature," ISO/IEEE, Tech. Rep., 2004.

[25] ——, "ISO/IEEE11073-10201 Health informatics – Point-of-care medical device communication - Domain information model," ISO/IEEE, Tech. Rep., 2004.

[26] D. Arney, S. Fischmeister, J. M. Goldman, I. Lee, and R. Trausmuth, "Plug-and-play for medical devices: Experiences from a case study," *Biomedical Instrumentation & Technology*, vol. 43, no. 4, pp. 313–317, 2009.

[27] D. Arney, M. Pajic, J. M. Goldman, I. Lee, R. Mangharam, and O. Sokolsky, "Toward patient safety in closed-loop medical device systems," in *Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems.* ACM, 2010, pp. 139–148.

[28] R. R. Maddox and C. Williams, "Clinical experience with capnography monitoring for pca patients," *APSF Newsletter*, vol. 26, p. 3, 2012.

[29] R. W. Hicks, V. Sikirica, W. Nelson, J. R. Schein, and D. D. Cousins, "Medication errors involving patient-controlled analgesia," *American Journal of Health-System Pharmacy*, vol. 65, no. 5, pp. 429–440, 2008.

[30] N. Leveson, *Engineering a Safer World: Systems Thinking Applied to Safety.* MIT Press, 2011.

[31] Association for the Advancement of Medical Instrumentation, "ANSI/AAMI/ISO 14971: Medical devices—Application of risk management to medical devices," ANSI/AAMI/ISO, Tech. Rep., 2000.

[32] ——, "ANSI/AAMI/IEC 80001: Application of risk management for IT Networks incorporating medical devices," ANSI/AAMI/IEC, Tech. Rep., 2010.

[33] SAE AS-2C Architecture Description Language Subcommittee, "SAE Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Language," SAE Aerospace, Tech. Rep., June 2014.

[34] A. J. Masys, "Fratricide in Air Operations. Opening the Black-Box: Revealing the 'Social'," Ph.D. dissertation, University of Leicester, 2010.

[35] M. Wallace, “Modular Architectural Representation and Analysis of Fault Propagation and Transformation,” *Electronic Notes in Theoretical Computer Science*, vol. 141, no. 3, pp. 53–71, 2005.

[36] J. Rushby, “Logic and epistemology in safety cases,” in *Computer Safety, Reliability, and Security*. Springer, 2013, pp. 1–7.

[37] S. E. Toulmin, *The Uses of Argument*. Cambridge University Press, 2003.

[38] T. Kelly and R. Weaver, “The Goal Structuring Notation – A Safety Argument Notation,” in *Proceedings of the Dependable Systems and Networks 2004 Workshop on Assurance Cases*, 2004.

[39] T. P. Kelly, “Arguing safety – a systematic approach to managing safety cases,” Ph.D. dissertation, University of York, September 1999.

[40] L. Feng, A. L. King, S. Chen, A. Ayoub, J. Park, N. Bezzo, O. Sokolsky, and I. Lee, “A safety argument strategy for PCA closed-loop systems: A preliminary proposal,” in *5th Workshop on Medical Cyber-Physical Systems, MCPS 2014, Berlin, Germany, April 14, 2014*, 2014, pp. 94–99. [Online]. Available: http://dx.doi.org/10.4230/OASIcs.MCPS.2014.94

[41] Institute of Electrical and Electronics Engineers, “ISO/IEC/IEEE 15288: Systems and software engineering—System life cycle processes,” ISO/IEC/IEEE, Tech. Rep., 2015.

[42] International Electrotechnical Commission, “ISO/IEC 12207: Systems and software engineering—Software life cycle processes,” ISO/IEC, Tech. Rep., 2008.

[43] A. Bertolino and L. Strigini, “Assessing the risk due to software faults: estimates of failure rate versus evidence of perfection,” *Software Testing, Verification*

and Reliability, vol. 8, no. 3, pp. 155–166, 1998. [Online]. Available: http://dx.doi.org/10.1002/(SICI)1099-1689(1998090)8:3⟨155::AID-STVR163⟩3.0.CO;2-B

[44] P. Bishop, R. Bloomfield, B. Littlewood, A. Povyakalo, and D. Wright, “Toward a formalism for conservative claims about the dependability of software-based systems,” *Software Engineering, IEEE Transactions on*, vol. 37, no. 5, pp. 708–717, 2011.

[45] L. Strigini and A. Povyakalo, “Software Fault-Freeness and Reliability Predictions,” in *Computer Safety, Reliability, and Security*. Springer, 2013, pp. 106–117.

[46] International Electrotechnical Commission, “Iec 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems,” IEC, Tech. Rep., April 2010.

[47] SAE International, “SAE ARP 4761: Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment,” SAE International, Tech. Rep., 1996.

[48] N. Leveson, C. Wilkinson, C. Fleming, J. Thomas, and I. Tracy, “A Comparison of STPA and the ARP 4761 Safety Assessment Process,” Massachusetts Institute of Technology Partnership for a Systems Approach to Safety, Tech. Rep., October 2014.

[49] Association for the Advancement of Medical Instrumentation, “ANSI/AAMI 60601: Medical electrical equipment,” ANSI/AAMI, Tech. Rep., 2013.

[50] ——, “ANSI/AAMI/IEC 62304: Medical device software—Software life cycle processes,” ANSI/AAMI/IEC, Tech. Rep., 2006.

[51] J. W. Liu, *Real-Time Systems*. Prentice Hall, 2000.

[52] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers, Principles, Techniques*, 2nd ed. Addison Wesley, 2007.

[53] International Organization for Standardization / International Electrotechnical Commission, "ISO/IEC 10746: Information technology – Open Distributed Processing," ISO/IEC, Tech. Rep., 1998.

[54] P. F. Linington, Z. Milosevic, A. Tanaka, and A. Vallecillo, *Building Enterprise Systems with ODP: An Introduction to Open Distributed Processing.* CRC Press, 2011.

[55] Institute of Electrical and Electronics Engineers, "ISO/IEC/IEEE 42010: Systems and software engineering—Architecture Description," ISO/IEC/IEEE, Tech. Rep., 2011.

[56] Object Management Group, "Omg unified modeling language (omg uml) superstructure," Object Management Group, Tech. Rep., August 2011.

[57] ——, "OMG Systems Modeling Language (OMG SysML)," Object Management Group, Tech. Rep., June 2012.

[58] Y. Jarraya, M. Debbabi, and J. Bentahar, "On the meaning of sysml activity diagrams," in *Engineering of Computer Based Systems, 2009. ECBS 2009. 16th Annual IEEE International Conference and Workshop on the*, April 2009, pp. 95–105.

[59] D. Latella, I. Majzik, and M. Massink, "Towards a formal operational semantics of uml statechart diagrams," in *Formal Methods for Open Object-Based Distributed Systems*, ser. IFIP The International Federation for Information Processing, P. Ciancarini, A. Fantechi, and R. Gorrieri, Eds. Springer US, 1999, vol. 10, pp. 331–347. [Online]. Available: http://dx.doi.org/10.1007/978-0-387-35562-7_25

[60] X. Li, Z. Liu, and H. Jifeng, "A formal semantics of uml sequence diagram," in *Proceedings of the 2004 Australian Software Engineering Conference*, 2004, pp. 168–177.

[61] P. H. Feiler, D. P. Gluch, and J. J. Hudak, "The Architecture Analysis & Design Language (AADL): An introduction," DTIC Document, Tech. Rep., 2006.

[62] P. H. Feiler and D. P. Gluch, *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis & Design Language.* Addison Wesley, 2012.

[63] F. Cadoret, E. Borde, S. Gardoll, and L. Pautet, "Design patterns for rule-based refinement of safety critical embedded systems models," in *Engineering of Complex Computer Systems (ICECCS), 2012 17th International Conference on*, July 2012, pp. 67–76.

[64] G. Lasnier, B. Zalila, L. Pautet, and J. Hugues, "Ocarina : An Environment for AADL Models Analysis and Automatic Code Generation for High Integrity Applications," in *Reliable Software Technologies Ada-Europe 2009*, ser. Lecture Notes in Computer Science, F. Kordon and Y. Kermarrec, Eds. Springer Berlin Heidelberg, 2009, vol. 5570, pp. 237–250. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-01924-1_17

[65] S. OSATE, "An extensible open source aadl tool environment," *SEI AADL Team technical Report*, 2004.

[66] P. Feiler, J. Hansson, D. de Niz, and L. Wrage, "System architecture virtual integration: An industrial case study," Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, Tech. Rep. CMU/SEI-2009-TR-017, 2009. [Online]. Available: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=9145

[67] J. Hansson, B. Lewis, J. Hugues, L. Wrage, P. Feiler, and J. Morley, "Model-based verification of security and non-functional behavior using aadl," *Security Privacy, IEEE*, vol. PP, no. 99, pp. 1–1, 2009.

[68] B. Kim, L. T. Phan, O. Sokolsky, and L. Lee, "Platform-dependent code generation for embedded real-time software," in *Compilers, Architecture and Synthesis for Embedded Systems (CASES), 2013 International Conference on.* IEEE, 2013.

[69] A. Murugesan, M. W. Whalen, S. Rayadurgam, and M. P. Heimdahl, “Compositional verification of a medical device system,” in *Proceedings of the 2013 ACM SIGAda annual conference on High integrity language technology.* ACM, 2013.

[70] SAE AS-2C Architecture Description Language Subcommittee, “SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Behavior Annex,” SAE Aerospace, Tech. Rep., April 2011.

[71] 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.* ACM, 2013, pp. 65–84.

[72] B. Larson, J. Hatcliff, S. Procter, and P. Chalin, “Requirements specification for apps in medical application platforms,” in *Software Engineering in Health Care (SEHC), 2012 4th International Workshop on.* IEEE, 2012, pp. 26–32.

[73] A. L. King, S. Procter, D. Andresen, J. Hatcliff, S. Warren, W. Spees, R. P. Jetley, P. L. Jones, and S. Weininger, “A publish-subscribe architecture and component-based programming model for medical device interoperability.” *SIGBED Review*, vol. 6, no. 2, p. 7, 2009.

[74] V. P. Ranganath, Y. J. Kim, J. Hatcliff, and Robby, “Communication patterns for interconnecting and composing medical systems,” in *Engineering in Medicine and Biology Society (EMBC), 2015 37th Annual International Conference of the IEEE*, Aug 2015, pp. 1711–1716.

[75] Ivy Biomedical Systems Inc., “Vital-Guard 450C Patient Monitor with Nellcor SpO_2,” Aug 2005.

[76] “Capnostream 20 Bedside Patient Monitor,” http://www.covidien.com/rms/products/capnography/capnostream-20p-bedside-patient-monitor.

[77] J. Siegel, *CORBA 3 fundamentals and programming.* John Wiley & Sons Chichester, 2000, vol. 2.

[78] “Xstream,” http://x-stream.github.io, 2016.

[79] T. Kühne, “Matters of (meta-) modeling,” *Software & Systems Modeling*, vol. 5, no. 4, pp. 369–385, 2006. [Online]. Available: http://dx.doi.org/10.1007/s10270-006-0017-9

[80] V. H. Balgos, “A Systems Theoretic Application to Design for the Safety of Medical Diagnostic Devices,” Master’s thesis, Massachusetts Institute of Technology, 2012.

[81] R. S. Martínez, “System Theoretic Process Analysis of Electric Power Steering for Automotive Applications,” Master’s thesis, Massachusetts Institute of Technology, 2015.

[82] B. Abrecht and N. Leveson, “Systems theoretic process analysis (stpa) of an offshore supply vessel dynamic positioning system,” Massachusetts Institute of Technology Lincoln Laboratory, Tech. Rep., February 2016. [Online]. Available: http://sunnyday.mit.edu/papers/Navy-Final-Report-2016-Feb-17.pdf

[83] J. Y. Halpern and J. Pearl, “Causes and Explanations: A Structural-Model Approach. Part I: Causes,” *The British Journal for the Philosophy of Science*, vol. 56, no. 4, pp. 843–887, 2005. [Online]. Available: http://dx.doi.org/10.1093/bjps/axi147

[84] N. Shankar, “Lazy compositional verication,” in *Compositionality: The Significant Difference.* Springer, 1998, pp. 541–564.

[85] C. L. Thornberry, “Extending the Human-Controller Methodology in Systems-Theoretic Process Analysis (STPA),” Master’s thesis, Massachusetts Institute of Technology, 2012.

[86] M. S. Placke, “Application of stpa to the integration of multiple control systems: A

case study and new approach," Master's thesis, Massachusetts Institute of Technology, 2014.

[87] J. Hatcliff, A. Wassyng, T. Kelly, C. Comar, and P. Jones, "Certifiably safe software-dependent systems: challenges and directions," in *Proceedings of the on Future of Software Engineering*. ACM, 2014, pp. 182–200.

[88] C. J. Walter and N. Suri, "The customizable fault/error model for dependable distributed systems," *Theoretical Computer Science*, vol. 290, no. 2, pp. 1223–1251, 2003.

[89] F. Leitner-Fischer, "Causality Checking of Safety-Critical Software and Systems," Ph.D. dissertation, Universität Konstanz, 2015.

[90] J. Thomas, "Extending and Automating a Systems-Theoretic Hazard Analysis for Requirements Generation and Analysis," Ph.D. dissertation, Massachusetts Institute of Technology, 2013.

[91] R. van der Meyden, "What, indeed, is intransitive noninterference?" in *Computer Security – ESORICS 2007*, ser. Lecture Notes in Computer Science, J. Biskup and J. Lopez, Eds. Springer Berlin Heidelberg, 2007, vol. 4734, pp. 235–250. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-74835-9_16

[92] J. Rushby, "Noninterference, transitivity, and channel-control security policies," SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park CA 94025, USA, Tech. Rep., May 2005.

[93] D. Dolev and A. C. Yao, "On the Security of Public Key Protocols," *Information Theory, IEEE Transactions on*, vol. 29, no. 2, pp. 198–208, 1983.

[94] D. L. Parnas and J. Madey, "Functional documents for computer systems," *Science of Computer Programming*, vol. 25, no. 1, pp. 41–61, 1995.

[95] W. R. Ashby, *An introduction to cybernetics.* Chapman & Hall Ltd., London, 1956.

[96] A. Abdulkhaleq and S. Wagner, "Integrated Safety Analysis Using Systems-Theoretic Process Analysis and Software Model Checking," in *Computer Safety, Reliability, and Security: 34th International Conference, (SAFECOMP)*, F. Koornneef and C. van Gulijk, Eds. Springer International Publishing, 2015, pp. 121–134. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24255-2_10

[97] N. G. Leveson, *Safeware: System safety and Computers.* Reading, Massachusetts: Addison-Wesley Publishing Company, Inc., 1995.

[98] P. Masci, A. Ayoub, P. Curzon, I. Lee, O. Sokolsky, and H. Thimbleby, "Model-Based Development of the Generic PCA Infusion Pump User Interface Prototype in PVS," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Bitsch, J. Guiochet, and M. Kaâniche, Eds. Springer Berlin Heidelberg, 2013, vol. 8153, pp. 228–240.

[99] J. Thomas and N. Leveson, "Performing hazard analysis on complex, software-and human-intensive systems," in *Proceedings of the 29th International Conference on Systems Safety*, 2011.

[100] C. H. Fleming, "Safety-driven Early Concept Analysis and Development," Ph.D. dissertation, Massachusetts Institute of Technology, 2015.

[101] B. Latour, *Reassembling the Social - An Introduction to Actor-Network-Theory.* Oxford University Press, Sep. 2005.

[102] C. W. Johnson, "Organisational, Political and Technical Barriers to the Integration of Safety and Cyber-Security Incident Reporting Systems," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science, F. Koornneef and C. van Gulijk, Eds. Springer International Publishing, 2015, vol. 9337, pp. 400–409. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-24255-2_29

[103] B. Meyer, "Applying "Design by Contract"," *Computer*, vol. 25, no. 10, pp. 40–51, 1992.

[104] D. Lempia and S. Miller, "DOT/FAA/AR-08/32. Requirements Engineering Management Handbook," Federal Aviation Administration, 2009.

[105] J. A. Beachy and W. D. Blair, *Abstract Algebra With a Concrete Introduction.* Prentice Hall, 1990.

[106] J. Misra and K. Chandy, "Proofs of networks of processes," *Software Engineering, IEEE Transactions on*, vol. SE-7, no. 4, pp. 417–426, July 1981.

[107] C. B. Jones, "Tentative steps toward a development method for interfering programs," *ACM Transactions on Programming Languages and Systems*, vol. 5, no. 4, pp. 596–619, Oct. 1983. [Online]. Available: http://doi.acm.org/10.1145/69575.69577

[108] J. Rushby, "A Formal Model for MILS Integration," SRI International, Computer Science Laboratory, 333 Ravenswood Ave., Menlo Park CA 94025, USA, Tech. Rep. SRI-CSL-08-XX, May 2008.

[109] W. Visser, K. Havelund, G. Brat, S. Park, and F. Lerda, "Model Checking Programs," *Automated Software Engineering*, vol. 10, no. 2, pp. 203–232, 2003.

[110] Y. Papadopoulos and J. A. McDermid, "Hierarchically Performed Hazard Origin and Propagation Studies," in *Computer Safety, Reliability and Security: 18th International Conference on (SAFECOMP)*, M. Felici and K. Kanoun, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 1999, pp. 139–152. [Online]. Available: http://dx.doi.org/10.1007/3-540-48249-0_13

[111] B. Gallina and S. Punnekkat, "FI4FA: A Formalism for Incompletion, Inconsistency, Interference and Impermanence Failures' Analysis," in *2011 37th EUROMICRO*

Conference on Software Engineering and Advanced Applications, Aug 2011, pp. 493–500. [Online]. Available: http://dx.doi.org/10.1109/SEAA.2011.80

[112] R. Alur and D. L. Dill, "A theory of timed automata," *Theoretical Computer Science*, vol. 126, no. 2, pp. 183 – 235, 1994. [Online]. Available: http://dx.doi.org/10.1016/0304-3975(94)90010-8

[113] A. L. King, L. Feng, O. Sokolsky, and I. Lee, "Assuring the Safety of On-Demand Medical Cyber-Physical Systems," in *Proceedings of the 1st International Conference on Cyber-Physical Systems, Networks, and Applications*, August 2013, pp. 1–6.

[114] C. Salazar, "A Security Architecture for Medical Application Platforms," Master's thesis, Kansas State University, 2014.

[115] C. Salazar and E. Vasserman, "Retrofitting Communication Security into a Publish/Subscribe Middleware Platform," in *Proceedings of the International Workshop on Software Engineering in Healthcare*, Washington, DC, July 2014.

[116] J. X. Mazoit, K. Butscher, and K. Samii, "Morphine in Postoperative Patients: Pharmacokinetics and Pharmacodynamics of Metabolites," *Anesthesia & Analgesia*, vol. 105, no. 1, pp. 70–78, 2007.

[117] J. Barnes, *High Integrity Software: The* SPARK *Approach to Safety and Security*. Addison-Wesley, 2003.

[118] J. Armstrong, R. Virding, C. Wikström, and M. Williams, *Concurrent Programming in ERLANG*. Englewood Cliffs, New Jersey 07632: Prentice Hall, 1993.

[119] B. R. Larson, P. Chalin, and J. Hatcliff, "BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software," in *NASA Formal Methods: 5th International Symposium, NFM 2013*, G. Brat, N. Rungta, and

A. Venet, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, May 2013, pp. 276–290. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-38088-4_19

Appendix A

SAFE Process

Systematic Analysis of Faults and Errors

Sam Procter, John Hatcliff; Kansas State University

General Tips

1. These instructions are designed to be used with publically-available templates
2. A partially-worked example is also available
3. The templates often assume one element where there may be multiple. In nearly all cases, the analyst can simply add rows.
4. Spreadsheet cells are quite small, so...
   - Cells can be made multiline by setting Text-Wrapping to "Wrap"
   - Using the name as an index, fill in all notes in the "Explanations" Section
5. Reference cells can (and should) be actual references to keep the worksheet elements synchronized
6. When the term "element" is used, it signifies either a component or a connection between two components.
7. When component A is a *predecessor* of component B, A provides input to B. When component A is a *successor* of component B, A gets its input from B.
8. A *link* is the pathway between a component and a connection (or a connection and a component). It is infallible: any failures are considered to be part of either the source or destination element.

Activity 0: Fundamentals

Overview: In this step the analyst fills in basic information about the system, like its name, component pieces, and the problems that need to be avoided. This corresponds to the "Fundamentals" Chapter of Leveson's *Engineering a Safer World*.

0.1: System Fundamentals

Overview: Here, the analyst considers basic information about the system as a whole. In the second substep (described below) she will be directed to consider the individual elements of the system.

1. Identify the System
   1. Guide:
      - This is the name of the system you're considering.
      - Enter the name of the system:
        1. Row: System: (1)

        2. Column: (B)
    2. Example:
        - PCA Interlock
2. Identify Accident Levels
    1. Guide:
        - These are the levels of accidents we'll want to avoid.
        - Enter the name of the accident levels:
            1. Row: Accident levels (5)
            2. Column: Name (B).
        - The reference column will not be used.
        - Names are typically prefixed with "AL."
        - The term "Accident Levels" comes from Leveson (see, e.g., Section 7.1 of *Engineering a Safer World*), but corresponds well to similar notions of loss categorization from other domains:
            - Medical: Qualitative Severity Levels (ISO 14971, Section D.3.4.2)
            - Avionics: Consequences of Failure Conditions (FAA AC 25.1309-1A, Figure 1)
    2. Examples:
        1. AL.Death
        2. AL.Discomfort
3. Identify the System and Environment Elements
    - Research questions:
        - How reasonable is it to identify components without relationships at this point in the process?
    1. Guide
        - This is just a listing of the elements that are under the system designer's control and in the environment
        - Enter the names of the components, one per cell
            1. Column: System (J), starting on row 3
            2. Column: Environment (K), starting on row 3
        - Add more components in more rows as necessary
        - Though this step seems simple, there are actually two tasks being performed, both of which should be carefully considered:
            1. Determining the system boundary: Which components are going to be directly controlled by the system developer and which are not
            2. Determining the level of abstraction: What defines a "component" -- each component could (in all likelihood)

itself be considered a system, with its own (sub)components and environment.

      - There is no right answer, just be able to justify the choices that get made.

   - Avoid the temptation to allocate the components to a control structure -- these lists will get modified in subsequent steps, and it will be easier to simply add / remove components without changing the architecture.

2. Examples
   1. System Components
      - PCA Pump
      - App Logic
      - App Display
      - Patient Sensors
   2. Environment Components
      - Patient

4. Identify Accidents
   1. Guide
      - These are the bad things that may happen. They should be traceable to the accident levels from 2.
      - Enter the name and associated accident level:
        1. Row: Accidents (7)
        2. Column: Name (B), Reference (C)
      - Names are typically prefaced with "ACC."
      - We use Leveson's terminology here as in step 2. Engineering a Safer World defines an Accident in section 7.1 as "An undesired or unplanned event that results in a loss, including loss of human life or human injury, property damage, environmental pollution, mission loss, etc."
      - Accidents are typically a pairing of an environmental component with an accident level. They should not speak to *how* the harm occurs (which is instead covered by a hazard).
        - We use the term in such a way that it is interchangeable with the term Mishap as defined in MIL-STD-882C and D.
   2. Examples
      1. ACC.Patient Dies (AL.Death)
      2. ACC.Patient is in Pain (AL.Discomfort)

5. Identify the System Hazards
   - Research questions
      - Is it possible to consistently identify hazardous component and environmental states at this point in the process?

1. Guide
    - These are ways that the accidents could happen. They should be traceable to accidents from step 3.
    - In addition to the standard name and reference, hazards also involve identifying:
        – A hazardous factor, which will be released in
        – A system state that, by a
        – System Component, that when combined with
        – The worst case state of an
        – Environmental component

      ... that will lead to the referenced accident.
    - There are equivalencies for some of these terms with, eg, Ericson's terminology (See pg 17 of Hazard Analysis Techniques for System Safety):

Our term	*Ericson's Term*
Hazardous Factor	Hazardous Element
System State	Initiating Mechanism
Environmental Component	Target / Threat

    - Enter the hazard and its information:
        1. Row: Hazards (9)
        2. Column: Name (B), Reference (C), Hazardous Factor (D), System Element (E), System Element State (F), Environment Element (G), Environment Element State (H), Manifestation (I)
    - Names are typically prefaced with "H."
2. Examples
    1. H.Patient Overdose (ACC.Patient Dies, Analgesic, PCA Pump, Pumping, Patient, Patient Cannot Tolerate More Analgesic, Improper Transmission)
    2. H.Patient Underdose (ACC.Patient is in Pain, Analgesic, PCA Pump, Not Pumping, Patient, Patient is in pain and can tolerate more analgesic, Delay / Drop)

6. Identify the System Safety Constraints
    1. Guide
        - These are constraints that, if they hold, guarantee the avoidance of the Hazards.
        - They are stated in nearly the same terms as the hazard, but with a minimal change that avoids the hazard (typically a different state of the element on the system boundary)
        - Enter the associated Safety Constraints
            1. Row: Safety Constraints (11)

        2. Column: Name (B), Reference (C)
    - Names are typically prefaced with "SC."
    - Often there is a one-to-one correspondence of hazards to safety constraints (e.g., the safety-constraint is simply a negation of the hazard), but sometimes a number of constraints collectively or individually prevent a single hazard.
    - Note that safety constraints are not simply safety requirements for the system, but rather high-level safety goals that will get discharged to the various components of the system as safety requirements on those components.
        - This generic nature makes it easier to catch less-traditional hazards, ie those resulting from things like: component interactions, degradation of the component over time, or the component's use by other actors as part of a larger process.
2. Examples
    1. SC.Dont Over Administer Analgesic (H.Patient Overdose, Analgesic, PCA Pump, Not Pumping, Patient, Near Harm)
    2. SC.Dont Under Administer Analgesic (H.Patient Underdose, Analgesic, PCA Pump, Pumping, Patient, Healthy but in pain)

7. (Optional) Determine the graphical candidate control structure
    1. Guide
        - This involves the allocation of system components to a structure which will allow each component to get the information it needs about the state of the controlled process to make safe decisions
        - This cannot be done in a spreadsheet format, though Google Spreadsheets allows insertion of diagrammatic drawings into sheets.
        - The control structure can be diagrammed as:
            - Components are drawn as boxes
            - Connections are directional connectors between components
            - System components and connections are drawn with solid lines
            - Environmental components and connections are drawn with dashed lines
        - Note that, in step 1.1.2, the components in this structure will be extended with process models (which cannot be determined at this point in the process)
            - Process models are drawn as solid boxes within components

        - Process variables are collections of process values, which are drawn text within a process model
    2. Examples
        - See the “Control Structure” sheet of the examples.

0.2: Component Fundamentals

Overview: For each element in the system, the analyst now creates a copy of the Element spreadsheet (which won't get filled out completely as part of this step) and fills in basic information. This is effectively the creation of a textual version of the system’s control structure.

1. Identify the element
    1. Guide
        - This is a name for the element under analysis
            - The first element examined should be the element closest to the system boundary (but still inside the system) as identified in Steps 4 and 7 of part 1.
            - Following the first element, the analyst should work backwards up the control structure (so, after examining an element, consider its immediate predecessor)
        - The name for should correspond to either:
            - One of the components inside the system boundary, or
            - A name for a connection between two components, one or both of which must be inside the system boundary
        - Enter a reference to the element:
            1. Row: Element (4+)
            2. Column: (A-B)
        - Note that we deterministically derive the element under analysis -- and examine all components in the control structure -- rather than manually choosing a control action, as in Leveson's *Engineering a Safer World*.
            - Other researchers, like Zahid H. Qureshi, have interpreted Leveson's methodology to involve three elements of the control structure (1. Controller errors, 2. Failure by the actuator to execute a control action, and 3. Bad feedback). In a similar spirit, we interpret Leveson's work as well.
    2. Examples
        - Component: PCA Pump
        - Connection: IVLine, PCA Pump --> Patient
2. Identify the successor link name
    1. Guide

- This is the link between this element and (one of) its successor(s).
- If the element has more than one successor link and plays a different role for those two links, *fill out a copy of the worksheet for each role/link pairing*
  - This step is necessary because a component playing multiple roles is essentially multiple components combined into one "box." The individual components should be considered individually.
- This is essentially the generic form of what Leveson refers to as "control actions" -- it's generic in that we do not restrict ourselves to links that carry control / command messages -- that are *leaving* the component.
- Enter the link name:
  1. Row: Successor Link Name: (4+)
  2. Column: (C-D)

2. Examples
   - Component: PCA Pump -> IV Line
   - Connection: IV Line -> Patient

3. Identify the predecessor link name(s)
   a. Guide
      - This is the set of links between this element and its predecessor (or, if there are multiple predecessor components, all of them). That is, these are the links over which the component's input arrives.
      - This is essentially the generic form of what Leveson refers to as "control actions" that are *entering* the component.
      - Enter the link name:
        1. Row: Predecessor Link Name: (4+)
        2. Column: (E-F)

   b. Examples
      - Component: AppLogicCommands -> PCA Pump
      - Connection: App Logic -> AppLogicCommands
4. Identify the element's classification(s)
   - Research questions:
     - What's a good set of component classifications?
     - What's a good set of connection classifications?
   1. Guide
      - This is the classification of the element according to the plays in the system.
      - Enter the classification

            1. Row: Architectural (4),
            2. Column: (H-I)
        - Architectural classifications should be one of:
            - Sensor
            - Actuator
            - Controller
            - Controlled Process
        - Connection architectural classification should be the classification of the source component and the destination component (eg, Sensor --> Controller)
    2. Examples
        - Component: Actuator
        - Connection: Actuator --> Controlled Process
5. Repeat for the source of all predecessor links
    - By repeatedly applying Step 0.2 to all predecessor links, the analyst will work backwards through the control structure of the application.

Activity 1: Externally Caused Unsafe Interactions

1.1: Deriving the Successor Dangers

Overview: These are the things that can go wrong with the current element's immediate successor (ie, the component that is the destination of the Successor Link identified in 0.2-2). Our whole analysis of a given component will be to avoid these problems.

1. Pull in the Successor dangers
    1. Guide
        - These can typically be imported from the previous worksheet.
            - If this is the first element considered after the full system, then the successor dangers are simply violations of the system's safety constraints (Column B, Row ~11).
            - If this is the second or later element, the successor dangers are the manifestations of the successor component (Columns D-I, Row ~9)
        - Enter references to the dangers:
            1. Row: (13+)
            2. Column: Successor Dangers (A-B)
    2. Examples
        - Component: IVLine.Overinfusion

        - This would be a successor danger for the PCA Pump --- the pump's goal is to avoid the IV line's "overinfusion" danger, by not being in it's *pumping* state when the patient is in the *near harm* state.
    - Connection: H.PatientOverdose
        - This would be a successor danger for the IV Line. Since the line exists at the system boundary, its successor dangers are the system-level hazards.

2. (If Component) Document the Process Model
    1. Guide
        - A process model is a collection of process variables which are essentially collections of abstract states (termed process values) of the component relative to the notion(s) of danger identified in the previous step.
        - The control structure, created in step 0.1.7, should be updated to contain these process variables and their values.
        - Enter references to the dangers:
            1. Row: (17+)
            2. Column: Process Variable (A), Process Value (B-I)
        - Process models are required for controller components, but can be documented for sensors and actuators as well. This stems from the realization that most components are, at a lower level of abstraction, entire systems consisting of internal sensors, controllers, and actuators.
    2. Examples
        - Component: PCA Pump
            - Process Variable: Ticket Duration
            - Process Value: 1, …, 600
        - Connection: N/A

1.2: Deriving the Element's Dangers

Overview: Here the analyst considers if problems with the input to this component would cause problems with its output. This step is similar to, but much deeper than, STPA's Step 1.

Note also that, from this point on, the analysis of one element does not depend on the analysis of its predecessors -- ie, the analysis is compositional from here on out.

1. Select a Predecessor Link
    1. Guide
        - These are the links identified in 0.2-3 (Column E-F, Row 3+)
        - Create a reference to the selected link:

        1. Row: (13+)
        2. Column: Pred. Link (C)
    2. Example
        - Component: AppLogicCommands -> PCA Pump
        - Connection: App Logic -> AppLogicCommands
2. Consider the four manifestations
    1. Guide
        - Here, the analyst should consider if it would be hazardous if the predecessor link exhibited any of the four manifestations
            - Note that we do not consider here if it's possible for the link to exhibit the given manifestations, only if their being exhibited would be hazardous
            - These manifestations are from Avizienis et-al's Basic Concepts and Taxonomy of Dependable and Secure Computing, where they are called Failure Domain's (see Fig. 8, pg 9)
        - The manifestations are:
            - Content -- ie, the value of messages on the link are incorrect, optionally divided further into:
                - High
                - Low
            - Halted -- ie, messages on the link stop arriving
            - Erratic -- ie, messages on the link appear out of the blue
            - Timing -- ie, messages on the link appear at the wrong time, typically divided further into
                - Early
                - Late
        - Record...
            - The result of the manifestation occurring --ie, a new danger-- (typically formatted as ComponentName.NameOfOccurrence)
            - Not Hazardous if messages on the link could not cause this hazard, or
            - Not Applicable if messages on the link could not exhibit this manifestation
            1. Row: (13+)
            2. Column: (D-I, as labelled)
3. Return to step 1.2-1 and repeat for all predecessor links identified in step 0.2-3

1.3 Examining the Externally Caused Dangers

Overview: Here the analyst explains how the successor dangers (identified in step 1.1) could be caused by bad input to the element (ie, the manifestations identified in step 1.2)

1. Select a Successor Danger
    1. Guide
        - The first thing an analyst needs to do is to select one of the successor dangers (identified in step 1.1, stored in column A-B, row ~9+)
        - Enter a reference to the danger:
            1. Row: (23+)
            2. Column: Successor Dangers (A)
        - Each successor danger may have more than one row -- this signifies that multiple errors in the current element will cause the same danger in the successor component
        - In some cases, more than one successor danger will occur simultaneously -- in this case, list all the successor dangers in the table cell.
    2. Examples
        - Component: IVLine.Overinfusion
        - Connection: H.PatientOverdose
2. Record the name of the danger
    1. Guide
        - Each previously-identified danger (from step 1.2-2) should show up at least once in this column
            - In general, though, there is a many-to-many mapping from successor dangers to externally caused dangers
        - Enter the name of the danger
            1. Row: (23+)
            2. Column: Name (B)
    2. Example
        - Component: PCA Pump.Spontaneously Give Drug
        - Connection: IV Line.Overadminister Drug
3. Identify the relevant process variable name and incorrect value
    1. Guide
        - Each component can be thought of as having a model of the controlled process
            - Sensors read the state of the controlled process directly

            - Controllers have a model of the controlled process provided by the sensors
            - Actuators get commands from controllers, which provides a (greatly reduced) view of the state of the controlled process
        - A mismatch between the controlled process state (identified in the previous step) and the component's process model lies at the root of every externally caused danger. This and the previous step combine to make that mismatch explicit.
        - Leveson's *Engineering a Safer World* gives an good primer on process models in Section 4.3 (pages 87-89)
        - Enter the process variable name and value
            1. Row: (23+)
            2. Column: Process Var. Name (C) and Value (D)
    2. Example
        - Component: PatientHealth, Ok
4. Interpret the danger
    1. Guide
        - Since one guideword or manifestation can be interpreted in different ways, the analyst should now provide a concrete interpretation that explains how the danger name in column B causes the successor danger in column A
        - Enter the interpretation of the danger
            1. Row: (23+)
            2. Column: (E-F)
    2. Example
        - Component: The PCA pump receives a command to run even though it is unsafe to do so
        - Connection: There is more analgesic put into the IVLine than the patient can safely tolerate
5. Identify any Co-occurring Dangers
    1. Guide
        - Sometimes dangers will only manifest in the presence of other dangers -- this may be reflected in the natural language of the environmental state and / or interpretation, but should be made explicit here by referring to the other dangers (separated by commas) here.
            - Note that elements with only one predecessor link will not typically have co-occurring dangers

- Each cause of the danger will need its own entry. So, if two components (A and B) have to fail simultaneously for the danger to occur, four rows will need to be created:
  - Two which cause the successor danger:
    - A's failure will have one row in the table (with B's failure as a co-occurring danger)
    - B's failure will have its own subsequent row (with A's failure as a co-occurring danger)
  - Two which are not hazardous:
    - A's failure without a simultaneous failure of B
    - B's failure without a simultaneous failure of A
- Enter the name of co-occurring dangers, or "None" if not required
  1. Row: (23+)
  2. Column: Co-occurring Dangers, (G)
- Dangers are assumed to be combined via AND joins --- more complex relationships (OR, M-of-N, etc.) can be explained in the interpretation / global env. state columns.

2. Example
   - Component: N/A
   - Connection: N/A

6. Run-time Detection
   1. Guide
      - This allows an analyst to specify a mechanism to detect the occurrence of a danger at runtime
      - Avizienis et-al. state that there are two ways errors can be detected at runtime (see Fig. 16, page 16), either
        - Concurrently (as the element is performing its job), or
        - Preemptively (while the element is suspended for testing)
      - Record the mechanism (typically prefaced with either "Concurrent" or "Preemptive"), or "None" if run-time detection is impossible
        1. Row (23+)
        2. Column Run-time Detection (H)
   2. Example:
      - Component: None
      - Connection: Concurrent: Flow metering
7. Run-time Handling
   1. Guide
      - This allows an analyst to specify a mechanism to correct the occurrence of a danger at runtime

- Avizienis et-al. state that there are three ways errors can be handled at runtime (see Fig. 16, page 16),
  - Rollback (restoring the system to a saved state),
  - Rollforward (moving to a state without errors, ie a known-safe state), or
  - Compensation (use redundancy to mask the error)
- Record the mechanism of correction (typically prefaced with "Rollback", "Rollforward", or "Compensation"), or "None" if run-time compensation is impossible
  1. Row (23+)
  2. Column Run-time Handling (I)

2. Example:
   - Component: None
   - Connection: Rollforward: Stop Analgesic Flow

Step 2: Working with Internal Faults

Overview: These are the things that can go wrong with the element itself. There are 18 classes of faults, 15 of which come from Avizienis's work (they're a condensed form of the 31 fault classes from Fig. 5, page 6) and 3 come in response to problems with compositional verification.

Num.	Guideword	Possible Compensation	Description
1	Software Bug	Static Verification	Mistakes made in software creation
2	Bad Software Design		Poor choices made in software creation
3	Compromised Software	TPM-like + Chain-of-trust	Adversary tampers with software in development
4	Compromised Hardware	“Exotic” only	Adversary tampers with hardware in development
5	Hardware Bug		Mistakes made in hardware development
6	Bad Hardware Design		Poor choices made in hardware development
7	Production Defect		Hardware production defects (due to natural phenomena)
8	Deterioration	Periodic inspection	Internal hardware fault at runtime due to natural phenomena

9	Environment damages hardware	Shielding, ECC	Externally caused hardware fault at runtime due to natural phenomena
10	Operator HW Mistake	Thoughtful UI, Authorization, Access Control	Operator makes a mistake while interacting with hardware
11	Operator HW Wrong Choice	Thoughtful UI, Re-training, Authorization, Access Control	Operator makes a poor choice while interacting with hardware
12	Adversary Accesses Hardware	Physical Security, “Exotic”	Adversary tampers with hardware at runtime
13	Adversary Accesses Software	Access Control (Network and Local), Physical Security, TPM-like + Chain-of-Trust	Adversary tampers with software at runtime
14	Operator SW Mistake	Thoughtful UI, Authorization,Access Control	Operator makes a mistake while interacting with software
15	Operator SW Wrong Choice	Thoughtful UI, Re-training, Authorization, Access Control	Operator makes a poor choice while interacting with software
16	Syntax Mismatch		The current element uses a different syntax than its predecessor
17	Rate Mismatch	QoS Specification + Enforcement	The current element expects input at a different rate than its predecessor outputs
18	Semantic Mismatch		The current element and its predecessor do not interpret a given value in the same way

2.1: Eliminating Classes of Internal Faults

Overview: While an analyst can consider each guideword individually, we also provide the following questions which can be used to eliminate entire classes of faults. Note that the default choice is italicized, and non-default answers should be justified in the "Faults Not Considered" cells, Row 32+, Columns A-B (Guideword), C-I (Justification)

1. Phase of Creation or Occurrence -- "Should faults from the element's development be considered?"
    - *Yes* -- Development and operational faults
    - No -- Operational faults only (Remove 1-7)
2. Dimension -- "Does the element involve hardware, software, or both?"
    - Hardware -- Hardware only (Remove 1-3,13-15)
    - Software -- Software only (Remove 4-12)
    - *Both* -- Both hardware and software
3. Phenomenological cause, pt 1 (unless Software dimension only) -- "Will the hardware elements be protected from natural phenomena?"
    - Yes -- No Natural faults (Remove 7-9)
    - *No* -- Natural faults included
4. Phenomenological cause, pt 2 -- "Does the element receive input from directly from a human operator?"
    - *Yes* -- Human-made operational faults included
    - No -- Human-made operational faults excluded (Remove 10-11,14-15)
5. Objective -- "Is it possible that an adversary could gain access to the element?"
    - *Yes* -- Malicious and Non-Malicious faults
    - No -- Non-Malicious faults only (Remove 3-4,12-13)
6. Interaction -- "Have the two components joined by this connection either worked together before or been developed together?"
    - Yes -- No interaction faults (Remove 15-18)
    - *No* -- Interaction faults

2.2 Examining the Internally Caused Dangers

Overview: Here the analyst explains how the successor dangers (identified in step 1.1) could be caused by faults internal to the element using the guideword table from above.

1. Select a guideword
    1. Guide
        - The first thing an analyst needs to do is to select one of the non-eliminated guidewords
        - Enter a reference to the guideword:
            1. Row: (38+)
            2. Column: Guideword (B)
        - Each guideword may have more than one row -- this signifies that the same guideword may cause multiple dangers in the successor component
    2. Examples

        - Component: Operator SW Mistake
        - Connection: Compromised Software

2. Select a Successor Danger that this guideword could cause
    1. Guide
        - Next, the analyst should pick one of the successor dangers the guideword from 2.2-1 could cause
        - Enter a reference to the danger:
            1. Row: (38+)
            2. Column: Successor Danger (A)
        - Each successor danger may have more than one row -- this signifies that different faults in the current element will cause the same danger in the successor component
    2. Examples
        - Component: IVLine.Overinfusion
        - Connection: H.PatientOverdose
3. Interpret the danger
    1. Guide
        - Since one guideword can be interpreted in different ways, the analyst should now provide a concrete interpretation that explains how the guideword in column B causes the successor danger in column A
        - Enter the interpretation of the danger
            1. Row: (38+)
            2. Column: (D-E)
    2. Example
        - Component: The PCA pump runs even though it's not commanded to
        - Connection: The connection drops the message
4. Identify any Co-occurring Dangers
    1. Guide
        - Sometimes dangers will only manifest in the presence of other dangers -- this may be reflected in the natural language of the interpretation, but should be made explicit here by referring to the other dangers (separated by commas) here.
            – Note that these can be other internal faults, or external dangers from Step 1
        - Each cause of the danger will need its own entry. So, if two faults (A and B) have to occur simultaneously for the danger to occur, four rows will need to be created:
            – Two which cause the successor danger:

        - Fault A will have one row in the table (with fault B as a co-occurring danger)
        - Fault B will have its own subsequent row (with fault A as a co-occurring danger)
      - Two which are not hazardous:
        - Fault A without a simultaneous fault B
        - Fault B without a simultaneous fault A
    - Enter the name of co-occurring dangers, or "None" if not required
      1. Row: (38+)
      2. Column: Co-occurring Dangers, (E)
    - Dangers are assumed to be combined via AND joins --- more complex relationships (OR, M-of-N, etc.) can be explained in the interpretation / global env. state columns.
  2. Example
    - Component: N/A
    - Connection: N/A
5. Design-time Detection
  1. Guide
    - This allows an analyst to specify a mechanism to detect the presence of a fault at the design-time of a system
    - Avizienis et-al. state that there are five verification approaches (see Fig. 19, page 18),
      - Static Analysis,
      - Theorem Proving,
      - Model Checking,
      - Symbolic Execution, or
      - Testing
    - Record the mechanism of detection (which may be one or more of the five verification methods or a domain-specific approach), or "None" if design-time compensation is impossible
      1. Row (38+)
      2. Column Run-time Handling (F)
  2. Example:
    - Component: Model Checking
    - Connection: Testing
6. Run-time Detection
  1. Guide
    - This allows an analyst to specify a mechanism to detect the occurrence of a danger at runtime

        - Avizienis et-al. state that there are two ways errors can be detected at runtime (see Fig. 16, page 16), either
            - Concurrently (as the element is performing its job), or
            - Preemptively (while the element is suspended for testing)
        - Record the mechanism (typically prefaced with either "Concurrent" or "Preemptive"), or "None" if run-time detection is impossible
            1. Row (38+)
            2. Column Run-time Detection (G)
    2. Example:
        - Component: None
        - Connection: Concurrent: Flow metering
7. Run-time Error Handling
    1. Guide
        - This allows an analyst to specify a mechanism to correct the occurrence of a danger at runtime
        - Avizienis et-al. state that there are three ways errors can be handled at runtime (see Fig. 16, page 16),
            - Rollback (restoring the system to a saved state),
            - Rollforward (moving to a state without errors, ie a known-safe state), or
            - Compensation (use redundancy to mask the error)
        - Record the mechanism of correction (typically prefaced with "Rollback", "Rollforward", or "Compensation"), or "None" if run-time compensation is impossible
            1. Row (38+)
            2. Column Run-time Handling (H)
    2. Example:
        - Component: None
        - Connection: Rollforward: Stop Analgesic Flow
8. Run-time Fault Handling
    1. Guide
        - This allows an analyst to specify a mechanism to correct the cause of a fault at runtime
        - Avizienis et-al. state that there are four ways faults can be handled at runtime (see Fig. 16, page 16),
            - Diagnosis (identifying and recording the causes of the issue),
            - Isolation (physically or logically excluding the faulty components from further participation in the system),

    - Reconfiguration (switching in spare components or reassigning tasks among non-failed components), or
    - Reinitialization (“rebooting” the system)
  - Record the mechanism of correction (typically prefaced with "Diagnosis", “Isolation”, “Reconfiguration”, “Reinitialization”), or "None" if run-time handling is impossible
    3. Row (38+)
    4. Column Run-time Handling (I)

2. Example:
   - Component: Reinitialization: Reboot the pump
   - Connection: None

Appendix B

SAFE Worksheets

System:	[Fill]								**System Boundary**	
	Fundamentals								System	Environment
	Name	Reference							[Fill]	[Fill]
Accident Levels	[Fill]	N / A								
Accidents:	[Fill]	[Fill]								
			Hazardous Factor	System Element	System Element State	Env. Element	Env. Element State			
Hazards:	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]			
Safety Constrai	[Fill]	[Fill]								
Explanations										
Reference	Explanation									

Activity 0: Fundamentals

Step 0.2

Element:	**Successor Link Name(s):**	**Predecessor Link Name(s)**	**Classification**	
[Fill]	[Fill]	[Fill]	**Architectural:**	[Fill]
	[...]	[...]		[...]

Activity 1: Unsafe Interactions

Step 1.1	**Step 1.2**						
Successor Dangers	**Manifestations**						
	Pred. Link	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
[Fill]	[Fill]	[Fill]		[Fill]	[Fill]	[Fill]	
[...]	[...]	[...]		[...]	[...]	[...]	

Process Variable Name	**Process Values**						**Unit**
[Fill]	[Fill]	[...]					[Fill]
[...]	[Fill]	[...]					[...]

Step 1.3

Externally Caused Dangers						**Proposed Mitigations**	
Successor Danger	Name	Process Var. Name	Process Var. Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]
[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
[Fill]	[Fill]
[...]	[...]

Step 2.2

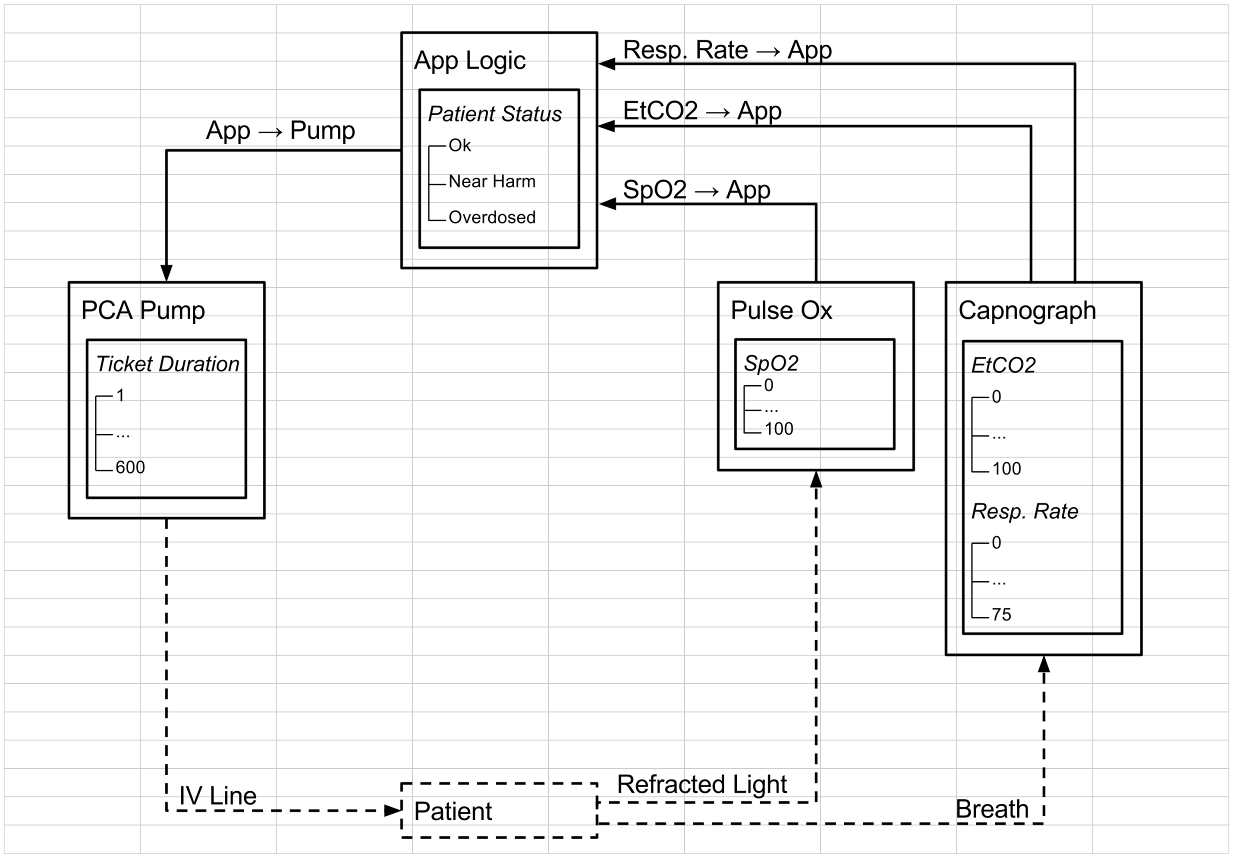
Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]	[Fill]
[...]	[...]	[...]	[...]	[...]	[...]	[...]	[...]

Appendix C

Full PCA Example

System:	PCA Interlock								**System Boundary**	
	Fundamentals								System	Environment
	Name	Reference							PCA Pump	Patient
									App Logic	
Accident Levels:	AL.DeathOrSeriousInjury	N / A							Pulse Oximeter	
									Capnograph	
Accidents:	Acc.PatientHarmed	AL.DeathOrSeriousInjury								
			Hazardous Factor	System Element	System Element State	Env. Element	Env. Element State			
Hazards:	H.TooMuchAnalgesic	Acc.PatientHarmed	Analgesic	PCA Pump	Pumping	Patient	NearHarm			
Safety Constraints:	SC.DontODPatient	H.TooMuchAnalgesic								
Explanations										
Reference	Explanation									
Acc.PatientHarmed	The patient is harmed or seriously injured as a result of the App's actions									
H.TooMuchAnalgesic	The patient is given more analgesic than they can safely tolerate									
Architecture	As modeled by Arney-etal in ICCPS10 (in section 4.3) with some modifications									
	A lot of possibly unmeetable assumptions (guaranteed timing of network and app)									
	Modified to include RR and EtCO2 physiological monitors (in addition to SpO2)									


App Logic
Patient Status
Ok
Near Harm
Overdosed
Resp. Rate → App
EtCO2 → App
SpO2 → App
App → Pump
PCA Pump
Ticket Duration
1
...
600
Pulse Ox
SpO2
0
...
100
Capnograph
EtCO2
0
...
100
Resp. Rate
0
...
75
IV Line
Patient
Refracted Light
Breath

Activity 0: Fundamentals

Step 0.2

Element:	**Successor Link Name:**	**Predecessor Link Name(s)**	**Classification**	
PCA Pump	PCA Pump -> IV Line	AppLogicCommands -> PCA Pump	**Architectural:**	Actuator

Activity 1: Unsafe Interactions

Step 1.1 / **Step 1.2**

Successor Dangers	**Manifestations**						
	Pred. Link	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
SC.DontODPatient	AppLogicCommands -> PCA Pump	PCAPump.TicketTooLong	Not Hazardous	Not Hazardous	PCAPump.ErraticTicket	PCAPump.EarlyTicket	PCAPump.LateTicket

Process Variable	**Process Values**						**Unit**
Ticket Duration	1	2	3	...	598	599	600	Seconds

Step 1.3

Externally Caused Dangers						**Proposed Mitigations**	
Successor Danger	Name	Process Var. Name	Process Var. Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
SC.DontODPatient	PCAPump.TicketTooLong	Ticket Duration	Higher than safe	The ticket has a time value that is too long	None	None	N / A
SC.DontODPatient	PCAPump.ErraticTicket	Ticket Duration	Any	The PCA Pump gets a ticket "out of the blue"	None	None	N / A
SC.DontODPatient	PCAPump.EarlyTicket	Ticket Duration	Any	The PCA Pump gets a ticket "too soon" -- before it has finished handling the previous ticket	None	Concurrent: Timeouts	Rollforward: Pump switches into permanent KVO (and notifies the clinician?)
SC.DontODPatient	PCAPump.LateTicket	Ticket Duration	Any	The PCA pump gets a ticket late, so it's valid past the time window it should be	None	Concurrent: Timestamped "tickets"	Rollforward: Pump switches into KVO

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" PCA Pump
Bad Software Design	
Compromised Software	
Compromised Hardware	
Hardware Bug	
Bad Hardware Design	
Production Defect	
Adversary Accesses Hardware	The hospital has physical security measures in place
Adversary Accesses Software	
Syntax Mismatch	The PCA pump isn't a connection between two components
Rate Mismatch	
Semantic Mismatch	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling

SC.DontODPatient	Deterioration	The pump is poorly maintained and fails open due to deterioration	None	Testing: Maintenance intervals should be established by the manufacturer and verified by regulators	Preemptive: Periodic pump examinations	None	None
SC.DontODPatient	Environment damages hardware	A cosmic ray flips a bit in the pump, making it run	None	Testing: Subject the pump to various environmental problems	Preemptive: Self-test	Compensation: ECC Memory	Isolation: Shielding
		The pump is poorly protected from the environment and fails open due to, eg, liquids			Preemptive: Periodic pump examinations	None	Isolation: Adequate sealing, N/A: careful use in the clinical environment
SC.DontODPatient	Operator HW Mistake	The operator accidentally presses a button she didn't mean to, giving either too much drug, too strong of a drug, or drug too quickly	None	Testing: Perform user studies on the interface	None	None	Diagnosis: Thoughtful UI (re)design
SC.DontODPatient	Operator HW Wrong Choice	The operator misunderstands the patient state and / or clinical process, giving either too much drug, too strong of a drug, or drug too quickly	None	Testing: Perform user studies on the interface	None	None	Diagnosis: Thoughtful UI (re)design, periodic retraining
SC.DontODPatient	Operator SW Mistake	The operator accidentally presses a button she didn't mean to, giving either too much drug, too strong of a drug, or drug too quickly	None	Testing: Perform user studies on the interface	None	None	Diagnosis: Thoughtful UI (re)design
SC.DontODPatient	Operator SW Wrong Choice	The operator misunderstands the patient state and / or clinical process, giving either too much drug, too strong of a drug, or drug too quickly	None	Testing: Perform user studies on the interface	None	None	Diagnosis: Thoughtful UI (re)design, periodic retraining

Activity 0: Fundamentals

Step 0.2

Element:	**Successor Link Name:**	**Predecessor Link Name(s)**	**Classification**	
AppToPumpCmds	AppLogicCommands -> PCA Pump	App Logic -> AppLogicCommands	**Architectural:**	Controller -> Actuator

Activity 1: Unsafe Interactions

Step 1.1	**Step 1.2**						
Successor Dangers	**Manifestations**						
	Pred. Link	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
PCAPump.TicketTooLong	App Logic -> AppLogicCommands	AppToPumpCmds.TicketTooLong	Not Hazardous	Not Hazardous	AppToPumpCmds.ErraticTicket	AppToPumpCmds.EarlyTicket	AppToPumpCmds.LateTicket
PCAPump.ErraticTicket							
PCAPump.EarlyTicket							
PCAPump.LateTicket							

Step 1.3

Externally Caused Dangers					**Proposed Mitigations**		
Successor Danger	Name	Global Env. State	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling	Design-time Mitigation
PCAPump.TicketTooLong	AppToPumpCmds.TicketTooLong	Patient.NearHarm	The ticket has a time value that is too long	None	None	N / A	N / A
PCAPump.ErraticTicket	AppToPumpCmds.ErraticTicket	Patient.NearHarm	The app->pump connection gets a ticket "out of the blue"	None	None	N / A	N / A
PCAPump.EarlyTicket	AppToPumpCmds.EarlyTicket	Patient.NearHarm	The app->pump connection gets a ticket "too soon" -- before it has finished handling the previous ticket	None	Concurrent: Timeouts	Rollforward: Network disables connection (and notifies the clinician?)	N / A
PCAPump.LateTicket	AppToPumpCmds.LateTicket	Patient.NearHarm	The app->pump gets a ticket late, so it's valid past the time window it should be	None	None	N / A	Timestamped tickets or tickets have a valid end-time (and the app needs a global clock)

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" network
Bad Software Design	
Compromised Software	
Compromised Hardware	
Bad Software Design	
Bad Hardware Design	
Production Defect	
Deterioration	Deterioration is not a significant source of concern over the life of the networking materials
Environment damages hardware	The app isn't responsible for network maintenance
Operator HW Mistake	The network doesn't interact directly with a human operator
Operator HW Error	
Hacked Hardware	The hospital has physical security measures in place
Hacked Software	
Operator SW Mistake	The network doesn't interact directly with a human operator
Operator SW Wrong Choice	

Step 2.2							
Internally Caused Dangers							
Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
PCAPump.TicketTooLong	Syntax Mismatch	A ticket is issued by the app in a different format than expected by the pump, so it runs for an unintended length of time	None	Model Checking: Verify syntax of sender and receiver	None	None	None
PCAPump.EarlyTicket	Rate Mismatch	Tickets are sent from the app too quickly for the pump to handle	None	Model Checking: Verify QoS of sender and receiver	Concurrent: Timeouts	Rollforward: Network disables connection (and notifies the clinician?)	None
PCAPump.LateTicket	Rate Mismatch	The app doesn't send tickets fast enough because it thinks the pump can't handle them	None	Model Checking: Verify QoS of sender and receiver	Concurrent: Expected arrival time	Rollforward: Network disables connection (and notifies the clinician?)	None
PCAPump.TicketTooLong	Semantic Mismatch	A ticket is issued by the app in a different format than expected by the pump, so it runs for an unintended length of time	None	Testing: Verify semantics of sender and receiver	Concurrent: Messages should use some sort of semantic tag, eg, 11073 nomenclature	Rollforward: Mismatched tags mean the app switches to a safe state and notifies the clinician	None

Activity 0: Fundamentals				
Element:	**Successor Link Name:**	**Predecessor Link Name(s)**	**Classification**	
App Logic	App Logic -> AppLogicCommands	SpO2ToApp -> App Logic	**Architectural:**	Controller
		EtCO2ToApp -> App Logic		
		RRToApp -> App Logic		

Activity 1: Unsafe Interactions							
Step 1.1	**Step 1.2**						
	Manifestations						
Successor Dangers	**Pred. Link**	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
AppToPumpCmds.TicketTooLong	SpO2ToApp -> App Logic	AppLogic.SpO2TooHigh	Not Hazardous	AppLogic.NoSpO2	Not Hazardous	AppLogic.SpO2Early	AppLogic.SpO2Late
AppToPumpCmds.ErraticTicket	EtCO2ToApp -> App Logic	Not Hazardous	AppLogic.EtCO2TooLow	AppLogic.NoEtCO2	Not Hazardous	AppLogic.EtCO2Early	AppLogic.EtCO2Late
AppToPumpCmds.EarlyTicket	RRToApp -> App Logic	AppLogic.RRTooHigh	Not Hazardous	AppLogic.NoRR	Not Hazardous	AppLogic.RREarly	AppLogic.RRLate
AppToPumpCmds.LateTicket							

Process Variable	**Process Values**						**Unit**
Patient Status	Very healthy	Quite healthy	Pretty healthy	...	A little healthy	Risk	Overdosed	N / A

Step 1.3							
Externally Caused Dangers					**Proposed Mitigations**		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
AppToPumpCmds.TicketTooLong	AppLogic.SpO2TooHigh	Patient.NearHarm	Patient Status >= Risk	The feedback from all three sensors is simultaneously incorrect leading the app to believe the patient is healthy	AppLogic.EtCO2TooLow AND AppLogic.RRTooHigh	None	N / A
None	AppLogic.SpO2TooHigh	N / A	Any	The feedback from either one or two of the sensors are incorrect, but due to redundancy harm is avoided	AppLogic.EtCO2TooLow OR AppLogic.RRTooHigh OR None	Concurrent: Assume best-case reading is valid	Compensation: Require healthy reading from all three sensors
AppToPumpCmds.TicketTooLong	AppLogic.EtCO2TooLow	Patient.NearHarm	Patient Status >= Risk	The feedback from all three sensors is simultaneously incorrect	AppLogic.SpO2TooHigh AND AppLogic.RRTooHigh	None	N / A
None	AppLogic.EtCO2TooLow	N / A	Any	The feedback from either one or two of the sensors are incorrect, but due to redundancy harm is avoided	AppLogic.SpO2TooHigh OR AppLogic.RRTooHigh OR None	Concurrent: Assume best-case reading is valid	Compensation: Require healthy reading from all three sensors
AppToPumpCmds.TicketTooLong	AppLogic.RRTooHigh	Patient.NearHarm	Patient Status >= Risk	The feedback from all three sensors is simultaneously incorrect	AppLogic.SpO2TooHigh AND AppLogic.EtCO2TooLow	None	N / A
None	AppLogic.RRTooHigh	N / A	Any	The feedback from either one or two of the sensors are incorrect, but due to redundancy harm is avoided	AppLogic.SpO2TooHigh OR AppLogic.EtCO2TooLow OR None	Concurrent: Assume best-case reading is valid	Compensation: Require healthy reading from all three sensors
None	AppLogic.NoSpO2	N / A	Any	The feedback from a sensor is missing, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket
None	AppLogic.NoEtCO2	N / A	Any	The feedback from a sensor is missing, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket
None	AppLogic.NoRR	N / A	Any	The feedback from a sensor is missing, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket

AppToPumpCmds.LateTicket	AppLogic.SpO2 Early	N / A	Any	The app's ticket is late because it is handling an (or a number of) unexpected SpO2 message(s)	Any	Concurrent: Timeouts	Compensation: Drop messages violating QoS settings
AppToPumpCmds.LateTicket	AppLogic.EtCO2 Early	N / A	Any	The app's ticket is late because it is handling an (or a number of) unexpected EtCO2 message(s)	Any	Concurrent: Timeouts	Compensation: Drop messages violating QoS settings
AppToPumpCmds.LateTicket	AppLogic.RREarly	N / A	Any	The app's ticket is late because it is handling an (or a number of) unexpected RR message(s)	Any	Concurrent: Timeouts	Compensation: Drop messages violating QoS settings
None	AppLogic.SpO2 Late	N / A	Any	The feedback from a sensor is delayed, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket
None	AppLogic.EtCO2 Late	N / A	Any	The feedback from a sensor is delayed, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket
None	AppLogic.RRLate	N / A	Any	The feedback from a sensor is delayed, but the app is built to not issue tickets if any information is missing	Any	Concurrent: Require signal from all three sensors	Rollforward: Issue zero-length ticket

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Syntax Mismatch	Element is a component, not a connection
Rate Mismatch	
Semantic Mismatch	
Compromised Hardware	We're using a previously-certified MAP implementation (ie, safety assessment of the MAP itself is not part of the safety assessment of the app)
Hardware Bug	
Bad Hardware Design	
Production Defect	
Deterioration	We're using an externally maintained MAP (ie, the protection of the MAP itself is not part of the safety assessment of the app)
Environment Damages Hardware	
Adversary Accesses Hardware	
Adversary Accesses Software	
Operator HW Mistake	The app logic doesn't interact with an operator.
Operator HW Wrong Choice	
Operator SW Mistake	
Operator SW Wrong Choice	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
AppToPumpCmds.TicketTooLong	Software Bug	A software bug leads to incorrect ticket calculations	None	Theorem proving: formally verify the behavior of the app logic.	None	None	None
AppToPumpCmds.ErraticTicket		A software bug leads to the app issuing tickets erratically					
AppToPumpCmds.EarlyTicket		A software bug leads to the app sending tickets earlier than it should					
AppToPumpCmds.LateTicket		A software bug leads to the app issuing tickets later than it should					
AppToPumpCmds.TicketTooLong	Bad Software Design	The app is designed for someone with a normal opioid tolerance (95% of the population) but the patient is an outlier	None	Testing and statistically-backed, "bootstrapping" certification	Concurrent: Physiological monitors	Rollforward: Use an adaptive algorithm and start with a very small dose	None

AppToPumpCmds.TicketTooLong		Other poor desgin choice leads to inappropriate-length or erratic tickets	None			None	None
AppToPumpCmds.ErraticTicket							
AppToPumpCmds.EarlyTicket							
AppToPumpCmds.LateTicket							
AppToPumpCmds.TicketTooLong	Compromised Software	An adversary gets access to the app while it's being developed	None	None	Concurrent: Some sort of TPM-like device on the MAP itself and a cryptographic chain-of-trust	None	Isolation: Chain-of-trust violations block app launch
AppToPumpCmds.ErraticTicket							
AppToPumpCmds.EarlyTicket							
AppToPumpCmds.LateTicket							

Activity 0: Fundamentals

Element:	Successor Link Name:	Predecessor Link Name(s)	Classification	
SpO2ToApp	SpO2ToApp -> App Logic	PulseOx -> SpO2ToApp	**Architectural:**	Sensor -> Controller

Activity 1: Unsafe Interactions

Step 1.1	Step 1.2						
	Manifestations						
Successor Dangers	Pred. Link	Content		Halted	Erratic	Timing	
		High	Low			Early	Late
AppLogic.SpO2TooHigh	PulseOx -> SpO2ToApp	SpO2ToApp.SpO2TooHigh	Not Hazardous	SpO2ToApp.NoSpO2	Not Hazardous	SpO2ToApp.SpO2Early	SpO2ToApp.SpO2Late
AppLogic.NoSpO2							
AppLogic.SpO2Early							
AppLogic.SpO2Late							

Step 1.3

Externally Caused Dangers					Proposed Mitigations		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
AppLogic.SpO2TooHigh	SpO2ToApp.SpO2TooHigh	Patient.NearHarm	Patient SpO2 > Actual Value	The feedback from the SpO2 sensor is higher than its actual value	None	None	None
AppLogic.NoSpO2	SpO2ToApp.NoSpO2	Any	None	There is no feedback from the SpO2 sensor	None	None	None
AppLogic.SpO2Early	SpO2ToApp.SpO2Early	Any	Any	The feedback from the SpO2 sensor arrives earlier than it should	None	Concurrent: Timeouts	Rollforward: Network disables connection (and notifies the clinician?)
AppLogic.SpO2Late	SpO2ToApp.SpO2Late	Any	Any	The feedback from the SpO2 sensor arrives later than it should	None	None	None

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" network
Bad Software Design	
Compromised Software	
Compromised Hardware	
Bad Software Design	
Bad Hardware Design	
Production Defect	
Deterioration	Deterioration is not a significant source of concern over the life of the networking materials
Environment damages hardware	The app isn't responsible for network maintenance
Operator HW Mistake	The network doesn't interact directly with a human operator
Operator HW Error	
Hacked Hardware	The hospital has physical security measures in place
Hacked Software	
Operator SW Mistake	The network doesn't interact directly with a human operator
Operator SW Wrong Choice	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling

AppLogic.SpO2 TooHigh	Syntax Mismatch	The SpO2 message is in a different syntactic format than what the app is expecting, so the app misinterprets it, leading to the app reading an inflated SpO2 value	None	Model Checking or Testing: Verify that syntax of SpO2 value used by Pulse Oximeter matches that used by app	None	N / A	None
AppLogic.NoSpO2		The SpO2 message is in a different syntactic format than what the app is expecting, so the app can't understand it, leading to the app having no SpO2 value					
AppLogic.SpO2 TooHigh	Semantic Mismatch	The underlying meaning of the SpO2 value produced by the puse oximeter isn't the same as the underlying meaning assigned to the value by the app, leading to the app interpreting an inflated SpO2 value	None	N/A: Standardize semantics at ecosphere level	Concurrent: Messages should use some sort of semantic tag, eg, 11073 nomenclature	Rollforward: Mismatched tags mean the app switches to a safe state and notifies the clinician	None
AppLogic.SpO2 Early	Rate Mismatch	The pulse oximeter sends SpO2 messages faster than the app is expecting / can handle them	None	Static Analysis: Verify that RT / QoS specifications cannot be violated	Concurrent: Specified RT / QoS Properties	If messages arrive faster than allowed the network drops them and the app switches into a safe state	None
AppLogic.SpO2 Late		The pulse oximeter doesn't send SpO2 messages as frequently as the app needs them				If messages don't arrive as frequently as specified the app switches into a safe state and notifies the clinician	

Activity 0: Fundamentals

Step 0.2

Element:	**Successor Link Name:**	**Predecessor Link Name(s)**	**Classification**	
Pulse Ox	PulseOx -> SpO2ToApp	PatientToPulseOx -> PulseOx	**Architectural:**	Sensor

Activity 1: Unsafe Interactions

Step 1.1	**Step 1.2**						
	Manifestations						
Successor Dangers	**Pred. Link**	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
SpO2ToApp.SpO2TooHigh	PatientToPulseOx -> PulseOx	PulseOx.HighReading	Not Hazardous	PulseOx.NoConnection	Not Hazardous	PulseOx.EarlyReading	PulseOx.LateReading
SpO2ToApp.NoSpO2							
SpO2ToApp.SpO2Early							
SpO2ToApp.SpO2Late							

Process Variable	**Process Values**						**Unit**
Patient SpO2	100%	99%	98%	...	2%	1%	0%	Percentage

Step 1.3

Externally Caused Dangers					**Proposed Mitigations**		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
SpO2ToApp.SpO2TooHigh	PulseOx.HighReading	Patient.Near Harm	Patient SpO2 > Read value	The pulse oximeter gets a bad reading from its patient-attachment (eg, finger clip)	None	Concurrent: Use a sensor with a data-quality reading	Rollforward: Drop readings without adequate quality (transforming this into NoSpO2)
SpO2ToApp.NoSpO2	PulseOx.NoConnection	Any	Any	The pulse oximeter's patient-attachment becomes disconnected or otherwise stops producing data	None	None	N / A
None	PulseOx.EarlyReading	Any	Any	The pulse oximeter's patient-attachment produces messages faster than the pulse-oximeter itself expects them	None	Concurrent: RT / QoS specifications	Rollforward: Drop readings that arrive too early
None	PulseOx.LateReading	Any	Any	The pulse oximeter's patient-attachment produces messages slower than the pulse-oximeter itself expects them	None	Concurrent: RT / QoS specifications	Rollforward: Notify clinician and stop producing data (transforming this into NoSpO2)

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" pulse oximeter
Bad Software Design	
Compromised Software	
Compromised Hardware	
Hardware Bug	
Bad Hardware Design	
Production Defect	
Adversary Accesses Hardware	The hospital has physical security measures in place
Adversary Accesses Software	

Operator HW Mistake		There are no user settings used for the pulse oximeter					
Operator HW Wrong Choice							
Operator SW Mistake							
Operator SW Mistake							
Syntax Mismatch		The pulse oximeter isn't a connection between two components					
Rate Mismatch							
Semantic Mismatch							
Step 2.2							
Internally Caused Dangers							
Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
SpO2ToApp.SpO2TooHigh	Environment damages hardware	A cosmic ray flips a bit in the PulseOx, breaking it in any possible way	None	None	Preemptive: Self-test	Compensation: ECC Memory	Isolation: Shielding
SpO2ToApp.SpO2TooHigh		The pulse oximeter is poorly protected from the environment and fails due to, eg, liquids		Testing: Subject the PulseOx to various environmental problems	Preemptive: Periodic pulseox examinations	Compensation: Additional physiological monitors should be used in case of errors with the pulse oximeter	Isolation: Adequate sealing, N/A: careful use in the clinical environment
SpO2ToApp.NoSpO2							
SpO2ToApp.SpO2Early							
SpO2ToApp.SpO2Late							
SpO2ToApp.NoSpO2	Deterioration	The pulse oximeter is poorly maintained and fails due to deterioration	None	Testing: Maintenance intervals should be established by the manufacturer and verified by regulators	Preemptive: Periodic examinations	None	None

Activity 0: Fundamentals

Element:	Successor Link Name:	Predecessor Link Name(s)	Classification	
EtCO2ToApp	EtCO2ToApp -> App Logic	Capnograph -> EtCO2ToApp	**Architectural:**	Sensor -> Controller

Activity 1: Unsafe Interactions

Step 1.1	Step 1.2						
	Manifestations						
Successor Dangers	**Pred. Link**	**Content**		**Halted**	**Erratic**	**Timing**	
		High	Low			Early	Late
AppLogic.EtCO2TooLow	Capnograph -> EtCO2ToApp	Not Hazardous	EtCO2ToApp.EtCO2TooLow	EtCO2ToApp.NoEtCO2	Not Hazardous	EtCO2ToApp.EtCO2Early	EtCO2ToApp.EtCO2Late
AppLogic.NoEtCO2							
AppLogic.EtCO2Early							
AppLogic.EtCO2Late							

Step 1.3

Externally Caused Dangers					Proposed Mitigations		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
AppLogic.EtCO2TooLow	EtCO2ToApp.EtCO2TooLow	Patient.NearHarm	Patient EtCO2 < Actual Value	The feedback from the EtCO2 sensor is lower than its actual value	None	None	None
AppLogic.NoEtCO2	EtCO2ToApp.NoEtCO2	Any	Any	There is no feedback from the EtCO2 sensor	None	None	None
AppLogic.EtCO2Early	EtCO2ToApp.EtCO2Early	Any	Any	The feedback from the EtCO2 sensor arrives earlier than it should	None	Concurrent: Timeouts	Rollforward: Network disables connection (and notifies the clinician?)
AppLogic.EtCO2Late	EtCO2ToApp.EtCO2Late	Any	Any	The feedback from the EtCO2 sensor arrives later than it should	None	None	None

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" network
Bad Software Design	
Compromised Software	
Compromised Hardware	
Bad Software Design	
Bad Hardware Design	
Production Defect	
Deterioration	Deterioration is not a significant source of concern over the life of the networking materials
Environment damages hardware	The app isn't responsible for network maintenance
Operator HW Mistake	The network doesn't interact directly with a human operator
Operator HW Error	
Hacked Hardware	The hospital has physical security measures in place
Hacked Software	
Operator SW Mistake	The network doesn't interact directly with a human operator
Operator SW Wrong Choice	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling

AppLogic.EtCO2 TooLow	Syntax Mismatch	The EtCO2 message is in a different syntactic format than what the app is expecting, so the app misinterprets it, leading to the app reading a deflated EtCO2 value	None	Model Checking or Testing: Verify that syntax of EtCO2 values used by Capnograph matches that used by app	None	N / A	None
AppLogic.NoEtCO2		The EtCO2 message is in a different syntactic format than what the app is expecting, so the app can't understand it, leading to the app having no EtCO2 value					
AppLogic.EtCO2 TooLow	Semantic Mismatch	The underlying meaning of the EtCO2 value produced by the puse oximeter isn't the same as the underlying meaning assigned to the value by the app, leading to the app interpreting a deflated EtCO2 value	None	N/A: Standardize semantics at ecosphere level	Concurrent: Messages should use some sort of semantic tag, eg, 11073 nomenclature	Rollforward: Mismatched tags mean the app switches to a safe state and notifies the clinician	None
AppLogic.EtCO2 Early	Rate Mismatch	The pulse oximeter sends EtCO2 messages faster than the app is expecting / can handle them	None	Static Analysis: Verify that RT / QoS specifications cannot be violated	Concurrent: Specified RT / QoS Properties	If messages arrive faster than allowed the network drops them and the app switches into a safe state	None
AppLogic.EtCO2 Late		The pulse oximeter doesn't send EtCO2 messages as frequently as the app needs them				If messages don't arrive as frequently as specified the app switches into a safe state and notifies the clinician	

Activity 0: Fundamentals

Element:	Successor Link Name:	Predecessor Link Name(s)	Classification	
RRToApp	RRToApp -> App Logic	PulseOx -> RRToApp	Architectural:	Sensor -> Controller

Activity 1: Unsafe Interactions

Step 1.1	Step 1.2						
	Manifestations						
Successor Dangers	Pred. Link	Content		Halted	Erratic	Timing	
		High	Low			Early	Late
AppLogic.RRTooHigh	PulseOx -> RRToApp	RRToApp.RRTooHigh	Not Hazardous	RRToApp.NoRR	Not Hazardous	RRToApp.RREarly	RRToApp.RRLate
AppLogic.NoRR							
AppLogic.RREarly							
AppLogic.RRLate							

Step 1.3

Externally Caused Dangers					Proposed Mitigations		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
AppLogic.RRTooHigh	RRToApp.RRTooHigh	Patient.NearHarm	Patient RR > Actual Value	The feedback from the RR sensor is higher than its actual value	None	None	None
AppLogic.NoRR	RRToApp.NoRR	Any	Any	There is no feedback from the RR sensor	None	None	None
AppLogic.RREarly	RRToApp.RREarly	Any	Any	The feedback from the RR sensor arrives earlier than it should	None	Concurrent: Timeouts	Rollforward: Network disables connection (and notifies the clinician?)
AppLogic.RRLate	RRToApp.RRLate	Any	Any	The feedback from the RR sensor arrives later than it should	None	None	None

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" network
Bad Software Design	
Compromised Software	
Compromised Hardware	
Bad Software Design	
Bad Hardware Design	
Production Defect	
Deterioration	Deterioration is not a significant source of concern over the life of the networking materials
Environment damages hardware	The app isn't responsible for network maintenance
Operator HW Mistake	The network doesn't interact directly with a human operator
Operator HW Error	
Hacked Hardware	The hospital has physical security measures in place
Hacked Software	
Operator SW Mistake	The network doesn't interact directly with a human operator
Operator SW Wrong Choice	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling

AppLogic.RRTooHigh	Syntax Mismatch	The RR message is in a different syntactic format than what the app is expecting, so the app misinterprets it, leading to the app reading an inflated RR value	None	Model Checking or Testing: Verify that syntax of RR values used by Capnograph matches that used by app	None	N / A	None
AppLogic.NoRR		The RR message is in a different syntactic format than what the app is expecting, so the app can't understand it, leading to the app having no RR value					
AppLogic.RRTooHigh	Semantic Mismatch	The underlying meaning of the RR value produced by the puse oximeter isn't the same as the underlying meaning assigned to the value by the app, leading to the app interpreting an inflated RR value	None	N/A: Standardize semantics at ecosphere level	Concurrent: Messages should use some sort of semantic tag, eg, 11073 nomenclature	Rollforward: Mismatched tags mean the app switches to a safe state and notifies the clinician	None
AppLogic.RREarly	Rate Mismatch	The pulse oximeter sends RR messages faster than the app is expecting / can handle them	None	Static Analysis: Verify that RT / QoS specifications cannot be violated	Concurrent: Specified RT / QoS Properties	If messages arrive faster than allowed the network drops them and the app switches into a safe state	None
AppLogic.RRLate		The pulse oximeter doesn't send RR messages as frequently as the app needs them				If messages don't arrive as frequently as specified the app switches into a safe state and notifies the clinician	

Activity 0: Fundamentals

Step 0.2

Element:	**Successor Link Name(s):**	**Predecessor Link Name(s)**	**Classification**	
Capnograph	Capnograph -> EtCOToApp	PatientToCapnograph -> Capnograph	**Architectural:**	Sensor
	Capnograph -> RRToApp			

Activity 1: Unsafe Interactions

Step 1.1	**Step 1.2**					
	Manifestations					
Successor Dangers	**Pred. Link**	**Content**	**Halted**	**Erratic**	**Timing**	
					Early	Late
EtCO2ToApp.EtCO2TooLow	PatientToCapnograph -> Capnograph	PatientToCapnograph.BadReading	PatientToCapnograph.NoData	Not Hazardous	PatientToCapnograph.EarlyData	PatientToCapnograph.LateData
EtCO2ToApp.NoEtCO2						
EtCO2ToApp.EtCO2Early						
EtCO2ToApp.EtCO2Late						
RRToApp.RRTooHigh						
RRToApp.NoRR						
RRToApp.RREarly						
RRToApp.RRLate						

Process Variable	**Process Values**							**Unit**
Patient EtCO2	100%	99%	98%	...	3%	2%	1%	Percent
Patient RR	75	74	73	...	2	1	0	Breaths per Minute

Step 1.3

Externally Caused Dangers					**Proposed Mitigations**		
Successor Danger	Name	Ctrld Process State	Process Var. Name and Value	Interpretation	Co-occurring Dangers	Run-time Detection	Run-time Handling
EtCO2ToApp.EtCO2TooLow	PatientToCapnograph.BadReading	Patient.NearHarm	EtCO2 < Actual Value	The sensor itself malfunctions, providing an over-optimistic reading of the patient's health	None	None	N / A
RRToApp.RRTooHigh			RR > Actual Value				
EtCO2ToApp.NoEtCO2	PatientToCapnograph.NoData	Any	None	The sensor stops providing any information at all, so the capnograph also can't produce any	None	None	N / A
RRToApp.NoRR							
None	PatientToCapnograph.EarlyData	Any	Any	The capnograph's patient-attachment produces messages faster than the capnograph itself expects them	None	Concurrent: RT / QoS specifications	Rollforward: Drop readings that arrive too early
None	PatientToCapnograph.LateData	Any	Any	The capnograph's patient-attachment produces messages slower than the capnograph itself need them	None	Concurrent: RT / QoS specifications	Rollforward: Notify clinician and stop producing data (transforming this into NoSpO2)

Activity 2: Internal Faults

Step 2.1

Faults Not Considered

Guideword	Justification
Software Bug	We're using a "proven in use" capnograph
Bad Software Design	
Compromised Software	
Compromised Hardware	

Hardware Bug	
Bad Hardware Design	
Production Defect	
Adversary Accesses Hardware	The hospital has physical security measures in place
Adversary Accesses Software	
Operator HW Mistake	There are no user settings used for the capnograph
Operator HW Wrong Choice	
Operator SW Mistake	
Operator SW Mistake	
Syntax Mismatch	The capnograph isn't a connection between two components
Rate Mismatch	
Semantic Mismatch	

Step 2.2

Internally Caused Dangers

Successor Danger	Guideword	Interpretation	Co-occurring Dangers	Design-time Detection	Run-time Detection	Run-time Error Handling	Run-time Fault Handling
EtCO2ToApp.EtCO2TooLow	Environment damages hardware	A cosmic ray flips a bit in the Capnograph, breaking it in any possible way	None	None	Preemptive: Self-test	Compensation: ECC Memory	Isolation: Shielding
RRToApp.RRTooHigh							
EtCO2ToApp.EtCO2TooLow		The capnograph is poorly protected from the environment and fails due to, eg, liquids		Testing: Subject the capnograph to various environmental problems	Preemptive: Periodic pump examinations	Compensation: Additional physiological monitors should be used in case of errors with the pulse oximeter	Isolation: Adequate sealing, N/A: careful use in the clinical environment
EtCO2ToApp.NoEtCO2							
EtCO2ToApp.EtCO2Early							
EtCO2ToApp.EtCO2Late							
RRToApp.RRTooHigh							
RRToApp.NoRR							
RRToApp.RREarly							
RRToApp.RRLate							
EtCO2ToApp.NoEtCO2	Deterioration	The capnograph is poorly maintained and fails due to deterioration	None	Testing: Maintenance intervals should	Preemptive: Routine Maintenance	None	None
RRToApp.NoRR							