Error Type Refinement for Assurance of Families of Platform-Based Systems (Extended Version)*

Sam Procter[1], John Hatcliff[1], Sandy Weininger[2], and Anura Fernando[3]

[1] Kansas State University, Manhattan, Kansas, USA
{samprocter, hatcliff}@ksu.edu
[2] United States Food and Drug Administration, Silver Spring, Maryland, USA
sandy.weininger@fda.hhs.gov
[3] Underwriters Laboratories, Chicago, Illinois, USA
anura.s.fernando@ul.com

Abstract. Medical Application Platforms (MAPs) are an emerging paradigm for developing interoperable medical systems. Existing assurance-related concepts for conventional medical devices including hazard analyses, risk management processes, and assurance cases need to be enhanced and reworked to deal with notions of interoperability, reuse, and compositionality in MAPs.
In this paper, we present the motivation for a framework for defining and refining error types associated with interoperable systems and its relevance to safety standards development activities in this domain. This framework forms the starting point for the analysis and documentation of faults, propagations of errors related to those faults, and their associated hazards and mitigation strategies—all of which need to be addressed in risk management activities and documented by assurance cases for these systems. We ground these concepts by describing how such a framework could be used in the AAMI/UL 2800 family of standards being developed for interoperable medical systems, and how error type refinement aligns with the envisioned 2800 refinement structures for different interoperability architectures and clinical applications. We show how this notion of refinement can potentially be supported in the AADL Error Modeling error type system, which would provide a basis for tool-supported risk management methodologies for platform-based interoperable systems.

Keywords: Interoperable medical systems, hazard analyses, faults, errors, reusable components and assurance

1 Introduction

Modern medical devices are increasingly network-aware, and this offers the potential to use middleware infrastructure to form systems of cooperating components. Initial integration efforts in industry are focused on streaming device

* This work is supported in part by the US National Science Foundation (NSF) (#1239543), the NSF US Food and Drug Administration Scholar-in-Residence Program (#1355778,#1446544) and the NIH / NIBIB Quantum Program.

data into electronic health records and integrating information from multiple devices into single customizable displays. However, there are numerous clinical motivations for moving beyond this to consider frameworks that, for example, can coordinate the actions of cooperating devices to automate clinical workflows, provide clinical "dashboards" that fuse multiple physiological data streams to provide composite health scores, generate alarms / alerts derived from multiple physiological parameters, provide automated clinical decision support, realize "closed loop" sensing and actuating scenarios, or even automatically construct and execute patient treatments.

1.1 Emerging Computational Paradigms and Dependable Architectures

In previous work, we have introduced the notion of a *medical application platform*. As defined in [7] a MAP is "a safety- and security-critical real-time computing platform for: (a) integrating heterogeneous devices, medical IT systems, and information displays via a communication infrastructure, and (b) hosting application programs (i.e., *apps*) that provide medical utility [beyond that provided by the individual devices] via the ability to both acquire information from and control integrated devices. . . and displays."

Platform-based approaches to integrated systems have a number of benefits, but they also introduce a number of safety and security challenges not addressed by current medical safety standards. While conventional approaches to development and deployment of safety-critical systems typically involve assessment and certification of complete systems, with a platform approach there is a need for (a) reuse of risk management artifacts, supporting hazard analyses, and assurance cases for both infrastructure implementations and components, and (b) compositional approaches to risk management, assurance, and certification.

Reliability Analysis and the Assurance Case Paradigm In development and deployment of medical devices and other safety-critical systems, hazard analyses play a key role in designing to achieve safety (i.e., the avoidance of harm to the patient, operation or the clinical environment) and in assessing the residual risk in a completed system. A hazard is often defined as "a source of harm," and system hazard analyses focus on identifying how hazards may arise in the context of system development and execution. The results of hazard analyses are typically reflected in an assurance case for a safety critical system. For example, an assurance case will often argue that appropriate hazards have been identified and that each hazard has been designed out, its risks controlled, or it has been otherwise dealt with in a manner that will result in an acceptable level of residual risk. A hazard analysis may proceed in a "bottom-up" fashion as in a Failure Modes and Effects Analysis (FMEA) which considers how each component may fail and how effects of component failure may propagate forward and outward to the system boundary, giving rise to hazards; alternatively, a hazard analysis may proceed in a "top-down" fashion as in a Fault Tree Analysis (FTA)

which starts from a hazardous state or event at the system boundary and reasons in a backward fashion to determine events and failures within the system that could cause the top-level unwanted event [5]. Concepts that cross-cut most hazard analyses are the notions of *fault*: the root cause of a component's failure to satisfy its specification and *error*: the deviation from a component's specified behavior [16][4]. In a bottom-up analysis, consideration of possible faults initiates the analysis and leads to an enumeration of the ways in which a component may produce errors (e.g., corrupted values, inappropriate timing of message transmittal, inability to perform a requested service, etc.) that may end up propagating outward to the system boundary and exposing hazards. In a top-down analysis, the analyst works backward through causality chains, considering how different types of errors could flow through the system, until faults that correspond to root causes are identified.

Certification, Standards, and Regulation To support the development, assurance, and certification of integrated medical systems, including systems built using platform concepts, the Association for the Advancement of Medical Instrumentation (AAMI) and Underwriters Laboratories (UL) are developing the 2800 family of standards for safe and secure interoperable medical systems. It has been proposed that AAMI / UL 2800 will provide a framework for specifying system and component-level safety and security requirements and guiding vendors in constructing objective evidence and assurance cases that demonstrate that their components, architectures, and integrated clinical systems comply with those requirements. 2800 is proposed to be organized as (a) a base "general" standard that provides architecture- and application-independent requirements and (b) "particular" standards that introduce application and architecture specific requirements by inheriting and refining the standard. It has been proposed that particular standards will specify how the generic risk management process and notions of faults, errors, failures, hazards, etc., in the base standard are specialized and allocated to the associated architectures, component kinds, and clinical applications. Vendor assurance cases that are used to demonstrate compliance with particular standards must provide evidence that their implementations account for, mitigate, or otherwise achieve an acceptable level of residual risk for the error types inherited through the standard hierarchy.

Reuse and Modularization In this standards-based approach for reasoning about the safety of interoperable systems, there is a significant need for a flexible nomenclature framework for faults / errors. Interoperable systems include components produced by different vendors. When risk management and assurance case artifacts are referenced and reused among vendors as systems are composed from components, component vendors need to be able to disclose what types of errors may propagate out of their components and what types of errors their

[4] Though these definitions are sourced from the AADL EM standard document, we note that they align well with, e.g., the taxonomy in [4]

components mitigate. There needs to be a standard vocabulary with a consistent semantic interpretation for faults and errors to ensure proper composition. Some errors are relevant to some types of components but not others (e.g., those associated with failure to achieve message transmittal in accordance with declared real-time and quality of service constraints are relevant to middleware but not to medical device components). Taxonomy mechanisms are needed to organize errors into categories according to kinds of components found in interoperable medical systems. Safety is ultimately expressed in terms of the notions of harm associated with a particular clinical application. Accordingly, there is a need to extend and specialize generic errors to specific clinical applications while providing a mechanism to facilitate traceability back to generic errors to support standard requirements that guide vendors to consider all appropriate generic error categories.

1.2 Our Contributions

The contributions of this paper are as follows:

- we identify overall goals for organizing and standardizing error types in the context of hierarchically organized standards for platform-based interoperable medical systems,
- we illustrate how the SAE standard Architecture and Analysis Design Language (AADL) Error Modeling framework, its open error type hierarchy, and its built-in error library can potentially support the desired notions of organization, extensibility, and refinement described above, and
- we describe how this open error type hierarchy would be used in the context of broader risk management, assurance case development, and certification regimes for platform-based interoperable medical systems.

2 Background

2.1 AAMI / UL 2800

AAMI / UL 2800 aims to define safety and security requirements to support the paradigm of constructing integrated systems from heterogeneous interoperable components. These requirements address component interfaces, implementations of components, middleware and networking infrastructure, and architectures that constrain the interactions between components as they are integrated to achieve system safety objectives. The standard is not anticipated to prescribe specific technologies or interface specifications for achieving integration and interoperability. Instead, it is expected to provide a framework for specifying system and component-level safety requirements and guiding vendors in constructing objective evidence and assurance cases that demonstrate that their components, architectures, and integrated clinical systems comply with those requirements.

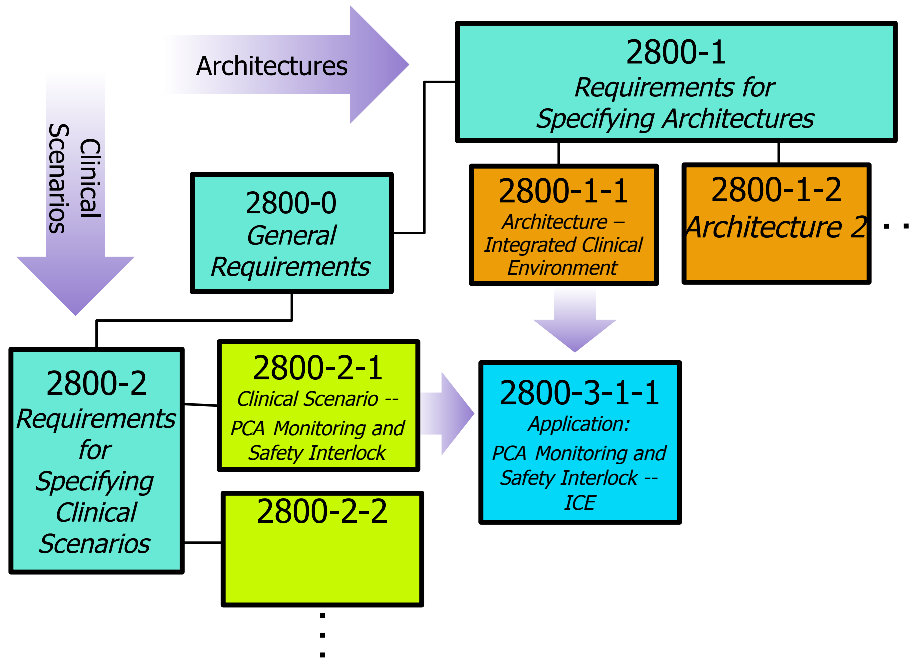


Fig. 1. The AAMI / UL 2800 Family of Standards

The structure for the 2800 family of standards aims to accommodate the following (sometimes conflicting) goals[5]:

- *Generality:* 2800 aims to provide safety requirements that are applicable to multiple architectures and a variety of clinical systems and applications.
- *Application Specificity:* Hazards and top-level system safety constraints, which typically drive the risk management and safety assurance processes, are application specific. Thus, AAMI/UL 2800 is expected to provide a framework for introducing specific standards that address particular systems and applications as well as stating requirements on how vendors develop and assure specific systems.
- *Architecture Specificity:* Plug-and-play interoperability and other ways of reusing system components and their assurance cannot be achieved without defining the architecture within which components interoperate. Thus, 2800 is expected to provide a framework for documenting architectures and the role that a specific architecture plays in (a) controlling potentially hazardous emergent properties by constraining interactions between components, and (b) providing safety-related services used to mitigate common errors.

[5] Some text in this section has been excerpted from unpublished communications as part of ongoing standardization efforts within the 2800 committee.

To reconcile these potentially conflicting goals and to enable reuse of application- and architecture-independent requirements, 2800 is proposed to be organized into a collection of linked standards (see Figure 1). The organization strategy is similar to that of IEC 60601 where a core set of requirements is refined along multiple dimensions to create requirements that are specialized to particular applications or implementation aspects. Specification-, application- and architecture-independent requirements are presented in the core *2800-0 General Requirements* standard while additional standards refine and extend core requirements for particular architectures (the *2800-1-x* series) or particular applications (the *2800-2-y* series). The *2800-3-x-y* series proposes to define application-specific requirements that are specialized for a particular architecture's approach to interoperability. The 2800 family's open, refinement-based approach allows for extension to address additional architectures and applications as new interoperability technologies and clinical needs arise. This enables manufacturers to specify an interoperable system's behavior but does not constrain how it should be implemented.

2.2 The Integrated Clinical Environment Architecture

The Integrated Clinical Environment (ICE) standard (ASTM F2761-2009 [3]) defines one particular architecture for MAPs. The boxes with dashed lines in Figure 2 present the ICE architecture. ASTM F2761 identifies an abstract "functional model" that includes components such the Supervisor, Network Controller, etc. with brief high-level descriptions of the role of these components within the architecture. Future implementation standards are envisioned that provide detailed implementation requirements and interface specifications for these components. The ICE *Network Controller* provides a high-assurance network communication capability, establishing virtual "information pipes" between heterogenous devices (often from different vendors) and apps running in the Supervisor.

The interface of a device is described in a domain specific language called the Device Model (DM) language. An ICE DM is a "representation of the capabilities of [a medical device] that includes information needed to qualitatively and quantitatively describe, control, and monitor its operation" and the Network Controller "shall provide association to and communication with each attached [device] by interpreting the device model." That is, the Network Controller exposes the ICE Interfaces of attached devices specified using the ICE DM to Supervisor apps. ASTM F2761 states that the *Supervisor* "provides a platform for functional integration between ICE compliant equipment via the network controller and can provide application logic and an operator interface" [3]. 2800-1-1, currently being drafted, complements and provides guidance for the planned ASTM F2761 implementation standards by defining safety and security requirements for the ICE architecture.

The Medical Device Coordination Framework (MDCF) is a prototype implementation of ICE jointly developed by researchers at Kansas State University and the University of Pennsylvania [10]. Components added by the MDCF are

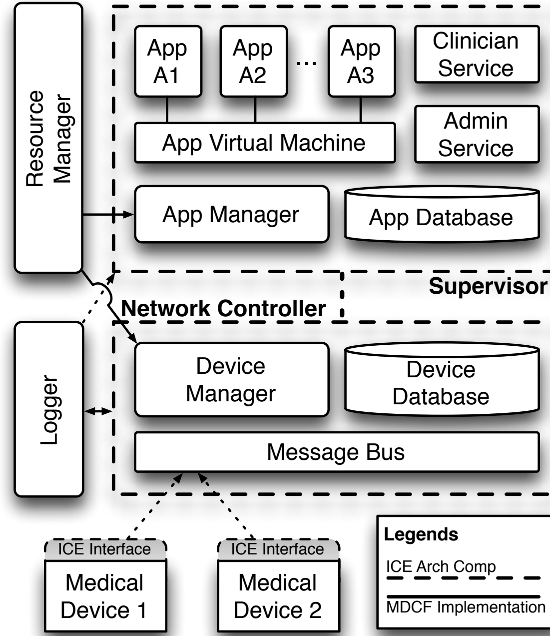


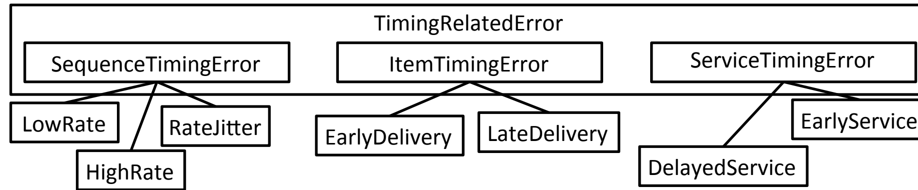
Fig. 2. The ICE Architecture and MDCF

presented in solid-lined boxes in Figure 2. In addition to enhancing the concept of *Apps*, The MDCF provides a middleware substrate and associated services [9], tools for authoring apps, generating executable APIs [14], and performing risk management activities [13].

2.3 PCA Safety Interlock Scenario

We describe one example of the MAP approach—a PCA safety interlock—here as a motivating example. After major trauma, hospital patients are often provided pain relief via patient-controlled analgesia (PCA) pumps. These allow a patient to press a button and request an analgesic (often an opioid narcotic) to manage pain. Standard safety mechanisms such as button timeouts fail to account for potential problems (e.g., opioid tolerance or human error), so various ways exist for an overdose to occur [8]. This can lead to respiratory depression and even death.

An ICE app can be used to implement a safety interlock that sets the PCA pump to a known safe state (i.e., infusion disabled) if—according to monitoring devices typically used in critical care situations—the patient shows signs of respiratory distress. While the exact set of monitored physiological parameters can vary, our example implementation uses the patient's blood-oxygen saturation (SpO_2), ratio of exhaled carbon-dioxide ($EtCO_2$) and respiratory rate (RR).



(a) Graphical view

```
1  --ErrorLibrary.aadl
2  TimingRelatedError: type set {ItemTimingError, SequenceTimingError,
       ServiceTimingError};
3  ItemTimingError: type;
4  EarlyDelivery: type extends ItemTimingError;
5  LateDelivery: type extends ItemTimingError;
6  SequenceTimingError: type;
7  HighRate: type extends SequenceTimingError;
8  LowRate: type extends SequenceTimingError;
9  RateJitter: type extends SequenceTimingError;
10 ServiceTimingError: type;
11 DelayedService: type extends ServiceTimingError;
12 EarlyService: type extends ServiceTimingError;
```

(b) Textual view

Fig. 3. The AADL Error Model Error Type Hierarchy for Timing Errors

After determining the respiratory health of the patient using some physiological model, the app can issue *enable* or *disable* commands to the pump. This app has been studied extensively in prior work, e.g., [2].

2.4 AADL's Error Model's Error Types

The Architecture Analysis and Design Language (AADL) enables the design of a system's architectural aspects: its hardware (e.g., processors, buses, memory, etc.), software (e.g., ports, processes, threads, etc.) and the bindings between the two [17]. In addition to this core functionality, there are a number of language annexes that extend the modeling of AADL to architecturally-related domains, such as the behavior annex, which enables the specification of component behavior, or the error modeling annex, which enables the modeling of failure-related aspects of their systems [15,16]. One useful aspect of this error modeling annex is its error definition and propagation mechanisms, which are modeled after Wallace's Fault Propagation and Transformation Calculus [19].

In the AADL error model, both faults and errors are represented as error types, instances of which can be propagated between components over their existing ports and channels (i.e., those specified in the core AADL language). The error model comes with a pre-built type hierarchy—the *error library*—that is composed of five "root" types (`ServiceError`, `TimingRelatedError`, `ValueRelatedError`, `ReplicationError`, and `ConcurrencyError`) that can be refined (through a full type lattice, created via extension, renaming, and

aggregation) down to more specific errors. Consider Figure 3, for example, which shows the hierarchy of the error library's `TimingRelatedError` (full hierarchies for the other types are available in [16]). The root type `TimingRelatedError` is an aggregation of three types, including `ItemTimingError`, which is refined (through type extension) to both `EarlyDelivery` and `LateDelivery`; i.e., if a single item (e.g., a message) has incorrect timing, it must be either early or late—it cannot be both, nor can it be neither. Finally, if the given root error types are insufficient for some purpose, completely new ones can be created.

The exact semantics of these errors are described in the the EMV2 standard in both natural and set-theoretic language [16]. Two terms defined in the standard are "A service S is defined as a sequence of n service items s_i with $n > 0$" and "A service item [is] characterized by a pair (v_i, δ_i) where v_i is the value or content of service item s_i and δ_i is the delivery time of service item s_i." Using these definitions, an `ItemTimingError` "represents errors where a service item [is] delivered outside its expected time range D_i of service item s_i" or "$\exists s_i \in S | \delta_i \notin D_i$." These definitions are refined along with the error types, so for example `EarlyDelivery` "represents errors where a service item is delivered before the expected time range..." or "$\exists s_i \in S | \delta_i < D_i$."

A significant strength of the AADL error modeling approach is its extensibility. This is particularly evident in situations where product safety is dependent on the reliability of safety-related control loops within the overall control structure. In such situations, the error modeling annotations can be extended to reflect reliability metrics that may be embodied in standards such as MIL 217 (Reliability Prediction of Electronic Equipment [12]) and UL 991 (Tests for Safety-Related Controls Employing Solid State Devices [18]) for supervisory control. Thus, when an appropriately modeled error is introduced into the control model of the system, the sensitivity of the safety control to variations in these reliability-related parameters can be better understood.

3 Error Refinement

3.1 Supporting 2800 Goals

Developing a framework for error types within 2800 addresses multiple assurance-related needs: (a) libraries of error types to guide hazard analyses, risk management processes, and aspects of assurance case construction, (b) appropriate coverage and document traceability targets (embedded in error libraries) that vendors can trace to as part of their compliance obligations, (c) a common interpretation/semantics for errors across vendors in order to support interoperability, (d) machine-readable specification of error types for automation of hazard analyses, and (e) systematic specification of error types within formal architecture descriptions to provide the basis for fault-injection testing.

Addressing (d) and (e) are beyond the scope of this paper; we propose goals for addressing (a-c) across the 2800 hierarchy below.

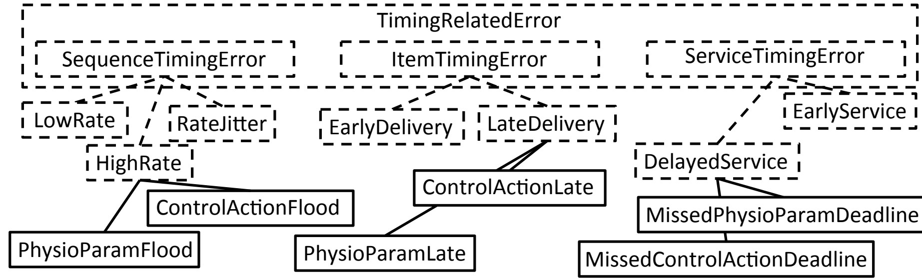
Identifying common error types: 2800-0 would provide a library of error types in an Informative Annex that supports the 2800 risk management process. The 2800-0 Risk Management requirements would specify that these error types should be considered in the initiating activities for bottom-up hazard analyses such as FMEA and would form the leaf nodes for top-down analyses such as FTA. Compliance requirements would specify that vendors should state how error types are accounted for in their analyses (e.g., they must use each error type or document why any that were left out were not applicable). Authors of standards that refine 2800-0 would be required to trace, via refinement mechanisms, newly introduced error types to those provided in the Informative Annex. The use of these error types in hazard analyses is discussed in more detail in Section 4.

Allocation of error types to common component categories found in interoperable systems: 2800-0 would also identify Interoperability Component Categories—common categories found in interoperable medical systems, e.g., medical devices (which may be further subdivided by role, e.g., into sensors and actuators), communication infrastructure, application hosting components, health IT systems, network gateways, etc. In the 2800-1-x series, 2800-1-x authors would indicate how their architectural components align with 2800 Interoperability Component Categories. Based on this association, they would be required to specify how each component in the architecture accounts for the error types associated with that category. This "accounting" may involving refining the errors into more specific categories for the particular architecture.

Allocation of error types to application components and hazards: In the 2800-2-x series, 2800-2-x authors would associate error types with specific devices or systems used in a particular application context. This would provide vendors seeking to comply with 2800-2-x a more precisely contextualized collection of errors, and a more accurate basis of accounting for appropriate "coverage" of errors associated with a particular context.

3.2 Refinement by Component Category

How can we leverage the concept of error refinement (via extension, renaming, or aggregation) from Section 2.4 given our goals from Section 3.1? We should focus on the "leaf" error types—i.e., the fully refined error model types. For example, authors of a 2800-1-x standard can decide whether a error type applies to a particular component role in the system architecture. If it does, they can extend it to one or more subtypes that better describe how the error might occur in a generic version of the component. If it does not apply, the standard should include justification for its exclusion it so that users of the architecture-specific error type library can understand the rationale. Consider Figure 4, which shows our timing errors from Figure 3 after their refinement to apps (different refinements would exist for other architectural elements, e.g., devices, networking components, supervisor components, etc.). Specifically, consider line 2 of Figure



(a) Graphical view

```
1  --AppErrorLibrary.aadl
2  -- Ignore EarlyDelivery, since the network never holds messages
3  PhysioParamLate : type extends ErrorLibrary::LateDelivery;
4  ControlActionLate : type extends ErrorLibrary::LateDelivery;
5  PhysioParamFlood : type extends ErrorLibrary::HighRate;
6  ControlActionFlood : type extends ErrorLibrary::HighRate;
7  -- Ignore LowRate, since it's just an accumulation of delayed messages
8  -- Ignore RateJitter, since it's either EarlyDelivery (which we don't have) or
      LateDelivery
9  MissedPhysioParamDeadline : type extends ErrorLibrary::DelayedService;
10 MissedControlActionDeadline : type extends ErrorLibrary::DelayedService;
11 -- Ignore EarlyService since it's impossible
```

(b) Textual view

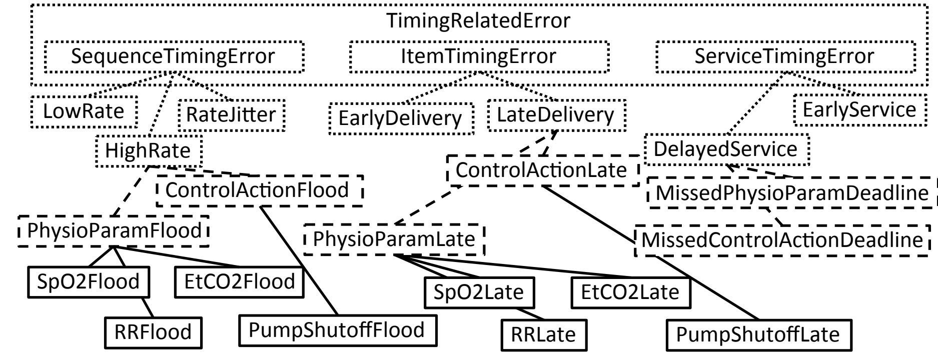
Fig. 4. The AADL EM Timing Hierarchy, refined for Apps

4b: since early delivery of messages is impossible, we eliminate it from consideration by simply not extending it. We expect that MAP apps will receive two types of input: physiological data from patient monitoring devices and commands to the application from other apps or clinicians. Lines 3 and 4 show that these two message types could both be late, and should be considered separately.

Of course, other types of components will have their own refinements. For example, the networking middleware (i.e., the Network Controller in ASTM F2761) is agnostic to message types, so its refinements to, e.g., the `HighRate` error type would be generic to the types of messages being transmitted. We expect some real-time network controllers (such as MIDAS [9]) to provide guarantees against any particular component saturating the network, so errors refined to these network controllers would reflect this.

3.3 Refinement by Component Implementation

We do not expect that all error types can be fully refined solely according to a component's architectural kind. While some component implementations, such as network controllers or supervisor components, may be largely interchangeable, others will not be. The behavior of medical devices and apps will vary considerably based on actual component implementation. For these components, the error types should be further refined. The process specified in the previous sec-



(a) Graphical view

```
--PCAInterlockErrors.aadl
SpO2Late : type extends AppErrorLibrary::PhysioParamLate;
EtCO2Late : type extends AppErrorLibrary::PhysioParamLate;
RRLate : type extends AppErrorLibrary::PhysioParamLate;
PumpShutoffLate : type extends AppErrorLibrary::ControlActionLate;
SpO2Flood : type extends AppErrorLibrary::PhysioParamFlood;
EtCO2Flood : type extends AppErrorLibrary::PhysioParamFlood;
RRFlood : type extends AppErrorLibrary::PhysioParamFlood;
PumpShutoffFlood : type extends AppErrorLibrary::ControlActionFlood;
-- Ignore MissedPhysioParamDeadline because we are just a subscriber
-- Ignore MissedControlActionDeadline because we are just a publisher
```

(b) Textual view

Fig. 5. The App Timing Error Hierarchy, refined for the PCA Interlock App

tion can be continued with our new architectural information, i.e., the actual architecture of a given component. Consider the error types from Figure 4 (associated with *2800-2-1* and *2800-3-1-x*) as they might be applied to the PCA interlock app from Section 2.3. As the app uses three physiological parameters (SpO_2, $EtCO_2$, and RR) and one control action (PumpShutoff), the generic app error types can be refined to be specific to these parameters, as in Figure 5. These fully refined error types are application specific and traceable to both the component's category (i.e., app) and the root AADL EM library types. The app's developer had a starting point for deriving hazards (i.e., Figure 4) rather than the much more ambiguous starting position of the status quo.

3.4 Using Error Types in Hazard Analysis and Testing

Hazard analyses include reasoning about where errors originate, what failures may result, and how errors propagate through the system. While the error type framework can aid in a more consistent presentation of these concepts, when combined with formal architectural descriptions of systems, it can also enable automation of some hazard analysis steps. The AADL EM error propagation mechanisms (see Figure 6) enable developers to specify how their components

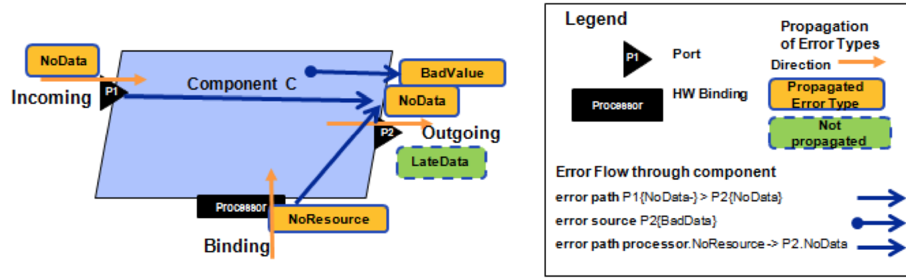


Fig. 6. An example of AADL's Error propagation, from [16]

create, propagate, transform and consume errors. In Figure 6, for example, the outgoing `NoData` error type can result either from an incoming `NoResource` error (i.e., `NoResource` is *transformed* by the component into `NoData`) or it can simply be *propagated* from a predecessor component. The component is a *source* of the `BadValue` error type, meaning that it can produce the error even if its input is error-free. Note that both top-down and bottom-up analyses leverage AADL's error propagation mechanisms.

There are two benefits to using the error types we have identified work with the EM error propagation mechanism. First, component developers will know what kind of errors they'll receive simply as a function of declaring what kind of component they are creating (e.g., device, app, network controller, etc.). Second, those component developers will also know the types of errors they are allowed to propagate. In the assurance case arguing for a component's safety-related properties, they can explain how their component handles (or fails to handle) each incoming error, and under what conditions their component propagates particular errors. This explanation, unlike in the status quo, will not be narrative in form, but rather can be written in the much more precise, machine-readable format of the AADL EM error types. Tooling can leverage these precise specifications of error creation, propagation and compensation for a range of purposes, e.g., the hazard analysis report from [13] or even more advanced techniques like fault-injection testing [1].

3.5 Allocation of Related Concepts to 2800 Standard Documents

Table 1 provides examples of how the error type framework might be used in 2800. Table entry names that appear in square brackets represent standards content that complies with requirements in standards higher in the hierarchy, whereas names in parentheses represent requirements that are refined (made more specific to a particular application or architecture). The table is not exhaustive; other requirements may compel, for example, vendors to capture error-related propagation or mitigation properties on component boundaries (following the concepts but not necessarily the AADL tooling in Section 3.4), specify how

2800-0: General Requirements	
Error Type Framework	Common categories of system and clinical process errors and semantics
System Topology	Common interoperability components kinds and allocation of 2800 errors
Risk Management	Requirements that vendors address 2800 error types in risk management
Testing	Requirements for fault injection testing to test for effectiveness of mitigation strategies
Assurance Cases	Requirements for arguing for safety in the presence of error handling and mitigation strategies
2800-1: Process/Requirements for Specifying Interoperability Architectures	
Traceability	Requirements that refining 2800-1-X standards map interoperability component kinds and errors to specified architectures
Arch. Specification	Requirements that refining 2800-1-X standards associate error types to standardized architectural viewpoints
2800-1-1: Safety and Security Requirements for ICE Interoperability Architecture	
[Traceability]	Map interoperability component kinds and refine error types to ICE Architecture components
[Arch. Specification]	Associate error types to standardized architectural viewpoints for ICE Architecture
(Risk Management)	Requirements that vendors address refined 2800-1-1 error types in risk management
(Testing)	Requirements for fault injection testing to test for effectiveness of mitigation strategies for refined 2800-1-1 error types
(Assurance Cases)	Requirements for arguing for safety in the presence of error handling and mitigation strategies for refined 2800-1-1 error types
2800-2: Process/Requirements for Specifying Clinical Scenarios	
Appl. Proc. Spec.	Requirements that refining 2800-2-X standards associate clinical error types to instantiations of common processes in the clinical application's context
Appl. Sys. Spec.	Requirements that refining 2800-2-X standards refine 2800-0 system error types to kinds of system components relevant to application
Appl. Proc. Mitigation	Informative Annex of common design / mitigation strategies for common clinical process error types.
2800-2-1: Safety Requirements for PCA Infusion Monitoring / Interlock	
[Appl. Proc. Spec.]	Associate clinical error types to instantiations of common processes in the clinical application's context.
[Appl. Sys. Spec.]	Refine 2800-0 system error types to kinds of system components relevant to application
(Testing)	Requirements for fault injection testing to test for effectiveness of mitigation strategies for refined 2800-2-1 error types
(Assurance Cases)	Requirements for arguing for safety in the presence of error handling and mitigation strategies for refined 2800-2-1 error types

Table 1. Examples of 2800 contents related to error type framework

errors at lower levels of abstraction (e.g., at the network or middleware layer) are manifested in terms of errors in application layers, and assign occurrence likelihood rankings to errors at particular points in the architecture.

4 Example

In [6], Feng et al. present a high-level safety argument for the PCA interlock scenario (reprinted as Figure 7). Briefly, this argument relies on a risk-benefit analysis, and in analyzing the risks, they employ the strategy of "Argue over all hazards" (**S2.1** in Figure 7). Though there is a slight mismatch in both terminology (what we term errors Feng et al. term hazards) and app structure (their model uses a "ticket-based" approach, where the PCA pump is enabled for some amount of time by a ticket, rather than the enable / disable approach described in this work), it is clear that a central challenge to arguing over *all* potential errors is simply *enumerating* all potential errors. We believe that the technique outlined in this work can greatly help reduce (though not completely eliminate) this epistemic challenge, regardless of if the underlying hazard analysis used to produce the list of errors / assurance case is bottom-up or top-down in nature.

4.1 Bottom-Up: Moving from Refined Types to Top-Level Errors

In a bottom-up analysis like FMEA, a developer would use the set of fully refined error types by analyzing the impact of each type's propagation into her component. Consider Table 2, which shows an example FMEA worksheet for the PCA Interlock example. Currently, the fourth column ("Causal Factors") is largely up to the system knowledge and hazard analysis expertise of the system analyst. With the list of fully refined fault types from this work, though, this column can be seeded *before* the analysis begins in earnest. For example, the set of error types for a developer of a new PCA Interlock app implementation would include types from Figure 5. A particularly poorly implemented app might only use SpO_2, so the impact of the `SpO2Late` error type, might, depending on app implementation, be transformed by the app into `PumpShutoffLate` or a value related error (i.e., the app uses stale data and issues an incorrect command). The resulting error(s) would be propagated out by the app into the PCA Pump. Once each error from the set of fully refined types has been analyzed in this manner, the developer will have an increased level of confidence in the completeness of the analysis.

4.2 Top-Down: Verifying Top-Level Errors with Refined Types

In a top-down analysis like FTA or System-Theoretic Process Analysis (STPA, [11]), the set of fully refined error types would be used *after* the hazard analysis has been completed, as a double-check of the completeness of the analysis. That is, the result of a top-down analysis is a list of root errors (i.e., those errors

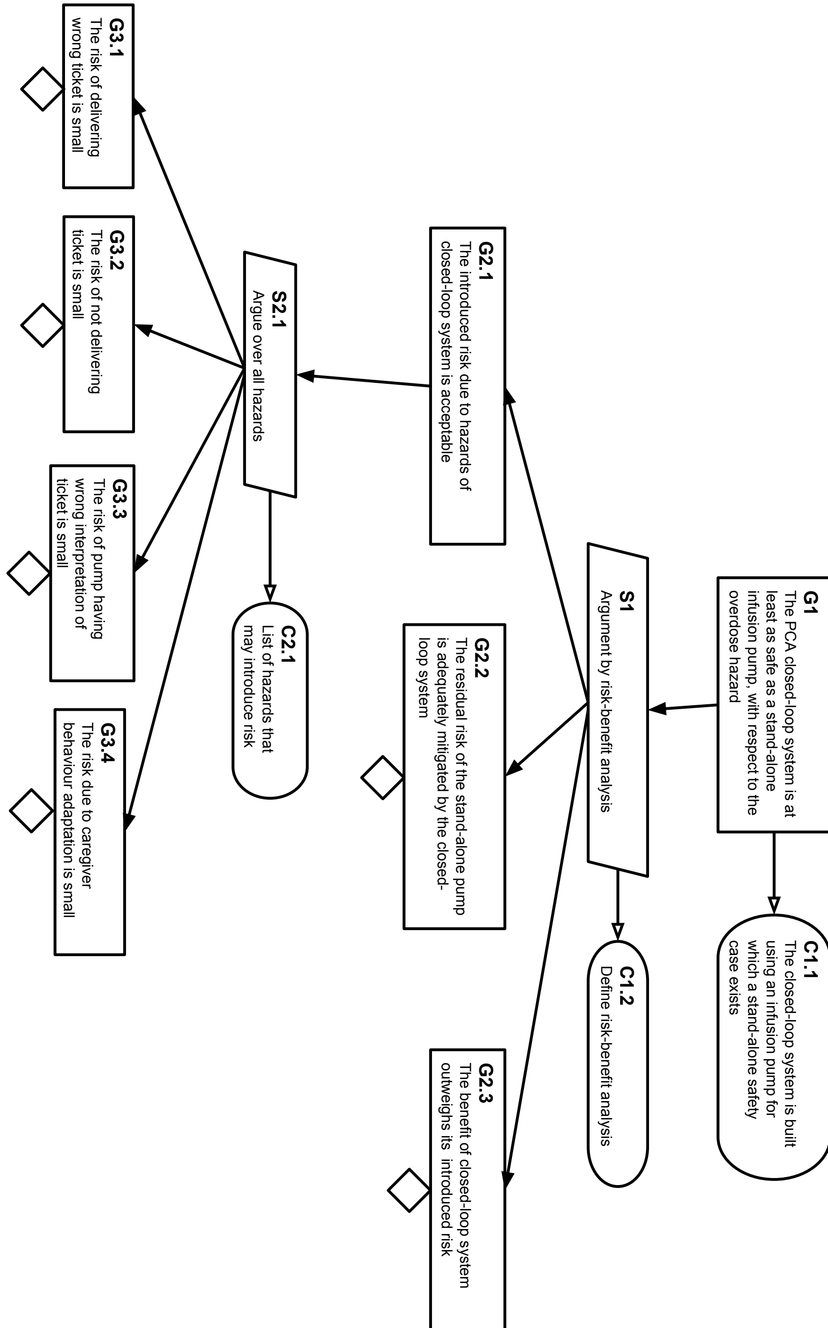


Fig. 7. An example assurance case for a PCA Interlock system, reproduced from [6]

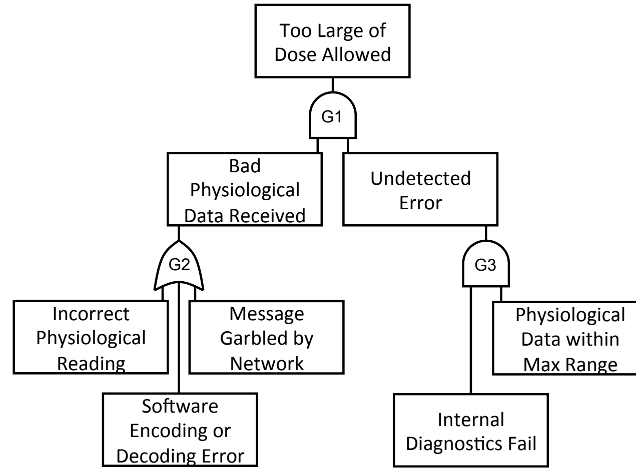


Fig. 8. An example of a FTA for the PCA Interlock App, reproduced from [13]

which either originate within components of the system, or enter through the system boundary) and the ways in which they are combined, propagated, or transformed by various system components to result in specified undesirable events (i.e., the top level nodes of the analysis tree in FTA or the unsafe control actions in STPA). This list of root errors would then be compared to the list of fully refined error types for the component, and any mismatches would indicate a problem: unaddressed errors from the set of refined types would reflect a gap in the current analysis while a fault discovered in analysis with no analogue in the set of refined error types would mean that the error library, or its refinements, are incomplete.

For example, consider Figure 8. While this is an overly simplified example, it is useful to show how the fully refined fault types might fit into a larger analysis process. When evaluating this fault tree, a reviewer or analyst should note that because it does not include errors from Figure 5 like “late physiological reading” (or a more refined error, like “late SpO_2 reading”), there is a good chance that one or more faults have been missed.

5 Conclusion

Safety-critical medical systems are being developed using platform-based architectures that emphasize multi-vendor component reuse. Work is needed to adapt existing risk management and assurance case techniques to support this paradigm of system development. In this paper, we have argued that there is a need for a refinement-based framework that enables defining error types to support safety standards for interoperable systems.

We are working with the 2800 standards committee to align these concepts with the 2800 risk management processes. We are integrating the framework

with our AADL-based risk management tooling environment for ICE apps [13] and collaborating with US Food and Drug Administration (FDA) engineers as part of the US National Science Foundation FDA Scholar-in-Residence program to ensure that the concepts are oriented to support regulatory submissions for MAP infrastructure implementations and apps. Although we have focused on the medical domain, the same motivation and solution strategy is relevant to other domains including avionics (e.g., the Open Group's Future Airborne Capability Environment[6]) and the industrial internet.

References

1. Arlat, J., Aguera, M., Amat, L., Crouzet, Y., Fabre, J.C., Laprie, J.C., Martins, E., Powell, D.: Fault injection for dependability validation: A methodology and some applications. Software Engineering, IEEE Transactions on 16(2), 166–182 (1990)
2. Arney, D., Pajic, M., Goldman, J.M., Lee, I., Mangharam, R., Sokolsky, O.: Toward patient safety in closed-loop medical device systems. In: Proceedings of the 1st ACM/IEEE International Conference on Cyber-Physical Systems. pp. 139–148. ACM (2010)
3. ASTM International: ASTM F2761 - Medical Devices and Medical Systems - Essential safety requirements for equipment comprising the patient-centric integrated clinical environment (ICE) (2009), www.astm.org
4. Avižienis, A., Laprie, J.C., Randell, B., Landwehr, C.: Basic Concepts and Taxonomy of Dependable and Secure Computing. Dependable and Secure Computing, IEEE Transactions on 1(1), 11–33 (2004)
5. Ericson II, C.A.: Hazard analysis techniques for system safety. John Wiley & Sons (2005)
6. Feng, L., King, A.L., Chen, S., Ayoub, A., Park, J., Bezzo, N., Sokolsky, O., Lee, I.: 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. pp. 94–99 (2014), http://dx.doi.org/10.4230/OASIcs.MCPS.2014.94
7. Hatcliff, J., King, A., Lee, I., MacDonald, A., Fernando, A., Robkin, M., Vasserman, E., Weininger, S., Goldman, J.M.: Rationale and architecture principles for medical application platforms. In: 2012 IEEE/ACM Third International Conference on Cyber-Physical Systems (ICCPS). pp. 3–12. IEEE (2012)
8. Hicks, R.W., Sikirica, V., Nelson, W., Schein, J.R., Cousins, D.D.: Medication errors involving patient-controlled analgesia. American Journal of Health-System Pharmacy 65(5), 429–440 (2008)
9. King, A., Chen, S., Lee, I.: The middleware assurance substrate: Enabling strong real-time guarantees in open systems with openflow. In: Object/component/service-oriented realtime distributed computing (ISORC), 17th IEEE Computer Society symposium on. IEEE (2014)
10. King, A., Procter, S., Andresen, D., Hatcliff, J., Warren, S., Spees, W., Jetley, R., Jones, P., Weininger, S.: An open test bed for medical device integration and coordination. In: Proceedings of the 31st International Conference on Software Engineering (2009)

[6] https://www.opengroup.us/face/

11. Leveson, N.: Engineering a Safer World: Systems Thinking Applied to Safety. MIT Press (2011)
12. Mil-Hdbk, U.: 217. Reliability Prediction of Electronic Equipment, version F, DOD, USA (1991)
13. Procter, S., Hatcliff, J.: 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. pp. 124–133. IEEE (2014)
14. Procter, S., Hatcliff, J., Robby: Towards an AADL-Based Definition of App Architectures for Medical Application Platforms. In: Proceedings of the International Workshop on Software Engineering in Healthcare. Washington, DC (July 2014)
15. SAE AS-2C Architecture Description Language Subcommittee: SAE Architecture Analysis and Design Language (AADL) Annex Volume 2: Annex B: Behavior Annex. Tech. rep., SAE Aerospace (April 2011)
16. SAE AS-2C Architecture Description Language Subcommittee: SAE Architecture Analysis and Design Language (AADL) Annex Volume 3: Annex E: Error Model Language. Tech. rep., SAE Aerospace (June 2014)
17. SAE AS5506B: Architecture Analysis and Design Language (AADL). AS-5506B, SAE International (2004)
18. UL: UL 991: Tests for Safety-Related Controls Employing Solid-State Devices (1995), `www.ul.com`
19. Wallace, M.: Modular architectural representation and analysis of fault propagation and transformation. Electronic Notes in Theoretical Computer Science 141(3), 53–71 (2005)

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	Current Controls	Hazard	Risk	Recommended Action
Provide SpO_2	Fails to provide	N/A	Network failure, device failure	SpO_2 not reported	Unknown patient state	App		Potential for overdose	3D	Default to KVO command
	Provides late	N/A	Network congestion, transient device failure	SpO_2 not reported	Unknown patient state	App		Potential for overdose	3C	Default to KVO command until new data arrive
	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. An Example FMEA for the PCA Interlock App, reproduced from [13]