



# Architecture-Supported Audit Processor: Interactive, Query-Driven Assurance

Sam Procter, Jerome Hugues

## ► To cite this version:

Sam Procter, Jerome Hugues. Architecture-Supported Audit Processor: Interactive, Query-Driven Assurance. 11th European Congress on Embedded Real-Time Systems (ERTS 2022), Jun 2022, Toulouse, France. hal-03701277

**HAL Id: hal-03701277**

**<https://hal.science/hal-03701277>**

Submitted on 5 Jul 2022

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Copyright

# Architecture-Supported Audit Processor: Interactive, Query-Driven Assurance

Sam Procter

sprocter@sei.edu.edu

Software Engineering Institute, Carnegie Mellon  
University  
Pittsburgh, Pennsylvania

Jerome Hugues

jjhugues@sei.edu.edu

Software Engineering Institute, Carnegie Mellon  
University  
Pittsburgh, Pennsylvania

## ABSTRACT

Establishing that safety-critical systems are actually safe requires a large effort and involves a range of tasks, from conducting preliminary hazard analyses to creating detailed assurance cases. This paper introduces the Architecture-Supported Audit Processor, or ASAP, which generates a number of safety-specific system views that deeply integrate a system's architecture and arguments about its safety. These views are generated interactively and automatically using safety-specific extensions to the Architecture Analysis and Design Language (AADL). Though use of the tooling and views do not require the use of any particular process, they align well with a system-theoretic approach. This paper discusses the background and use of ASAP on a demonstrative example.

## KEYWORDS

Model-Based Engineering, System Safety, Hazard Analysis, Architecture Analysis and Design Language (AADL)

## 1 INTRODUCTION

Safety-critical systems, i.e., those systems whose failure would result in death, injury, or unacceptable financial losses, are becoming increasingly sophisticated and software reliant [10]. Developing confidence in the safe and reliable behavior of these systems requires efforts to assure them. The cost and time required for these assurance efforts is significant, and it has been argued that they are a bottleneck that prevents the rapid deployment of new and improved versions [5]. Additionally, many assurance practices were not designed for modern systems and do not consider the impact of software on safety [10], nor do they take advantage of model-based engineering techniques for the traceability of safety artifacts to system elements.

In this paper, we report on a tool-supported approach that addresses these shortcomings. We propose contextualizing assurance evidence in an environment that supports modern system development practices, and that explicitly links assurance evidence to safety argumentation. We present

the *Architecture-Supported Audit Processor* (ASAP), an interactive tool which follows this strategy and supports common assurance activities in an architecture-centric, model-based system development environment. The tool provides an assurance-specific view of a system that builds on previous work on merging modern safety analysis and system architecture [15].

- (1) **Research Context** We identify three research ideas from the safety and argumentation communities and discuss how they illuminate a path forward for next-generation hazard analysis.
- (2) **SAFE Improvements** We propose an approach for performing safety analysis that builds on the *Systematic Analysis of Faults and Errors* (SAFE) hazard analysis technique, which supports model-based, compositional safety argumentation.
- (3) **Tool** We describe the inputs, use cases, and outputs of the ASAP tool.

In order to better explain the use of ASAP, we also orient its inputs and outputs in the process of a popular hazard analysis technique:

- (4) **Mapping to STPA** We establish an example mapping to STPA [13], a popular system-theoretic hazard analysis. Our mapping leverages system development tasks that are already performed in a typical development and presents safety information in an interactive, navigable format that can be queried.

The remainder of the paper is organized as follows: Section 2 surveys three research topics which informed the design of ASAP. Section 3 describes background concepts, including languages, tooling, and theory this work directly builds on. Section 4 describes ASAP and its application to a small system. Section 5 presents related work. We discuss future work in Section 6 and conclude in Section 7.

## 2 SURVEY: THREE CHALLENGES TO ASSURANCE

In addition to the technical background, covered in the next section, this work has been informed by three challenges to system assurance, which we describe here.

## 2.1 Assurance evidence should be contextualized with explicit safety arguments

One of the challenges that makes assurance difficult is that the evidence produced by some assurance strategies may not be what is most relevant for actually determining the safety of a system. Take for example hazard analysis – a common way of assessing the safety of a system [4]. Many hazard analyses consist of a series of steps to be performed at one or more stages in the system development lifecycle (e.g., preliminary design, detailed design, system test, etc.). The outcome of the analysis’s steps is evidence that the system is either free from the types of safety issues that the technique is designed to detect or those issues have been mitigated to a point where residual risk is acceptable in light of the system’s benefits. Some safety standards rely on this evidence for certification of a system’s worthiness for a particular mission.

Rushby, however, has argued that the claims and arguments which rely on that evidence—and upon which those standards are built—are sometimes based on reasoning that is left implicit [18]. He continues by noting that while approaches based on standards as well as structured-argumentation like *safety cases* (also referred to more generally as *assurance cases*) both have their strengths and weaknesses, standards-based approaches are “slow-moving and conservative.” This aligns, to some extent, with arguments made by Leveson, who writes that many hazard analyses and system safety standards are fundamentally outdated in their approach [12]. In Leveson’s view, a common error is to evaluate a system’s *reliability* and use that as a stand-in for the system’s *safety*. This approach, she continues, was more valid when safety-critical systems were largely hardware based: individual component failures were much more likely to be the ultimate cause of a system failure before the addition of software significantly increased the variety of component configurations [10]. As a consequence, we undertake the following:

**Our Goal:** Assurance evidence should be contextualized, and that context should be explicit safety argumentation.

## 2.2 Assurance argumentation should be hierarchical

A second challenge to system assurance efforts stems from the competing goals of assurance documentation: it should be both easily understood yet completely accurate. System descriptions which are brief and abstract may be easily understood, but lack the technical precision and depth necessary to convey a complete understanding of a system. Complete and precise system descriptions, on the other hand, can take a significant amount of time to understand. This problem can be addressed, to an extent, through standardization: when

assurance documentation is packaged in an expected format, familiarity with the standard can aide in understanding. But this leads to institutional inertia; i.e., Rushby’s criticism of standards-based approaches as slow-moving [18] or Espinoza et al.’s more pointed criticism that standards can be a barrier to innovation [5].

We note that there is an interesting parallel here to another field where arguments are designed to convince human experts of their correctness: that of mathematical proof. In that domain, Lamport has argued for the utility of hierarchically structured, hypertext-enabled proofs [9]. An initial, high-level argument can be presented at an abstract level, but the reader can expand portions that are unclear as necessary; in this way technology and argument structure can be used to address the competing goals simultaneously. Therefore, we also undertake the following:

**Our Goal:** Assurance evidence should be initially presented at a high level, but a viewer should be able to expand argumentation as desired.

## 2.3 Assurance evidence should be modular and composable

A third challenge stems from the discrepancy between the way systems are built (compositionally, as aggregates of components) and how safety argumentation is structured (monolithically). That is, because critical systems are designed for operation in particular environment, safety argumentation rarely composes as easily as software or hardware elements. This mismatch can lead to inconsistency in guidance for assuring seemingly related systems [7] or system updates that are either postponed or avoided altogether to avoid the costs of (re)certification [5]. However, fully compositional safety is an enduring challenge because it requires successfully pursuing one of two very challenging strategies:

- (1) *Anticipating the environment and role of a component:* This was the approach taken by ISO 26262’s concept of a “Safety Element out of Context” [8], ISO 14971’s concept of “Intended Use” [1]), and SAFE’s concept of a component’s role [15].
- (2) *Completely describing all aspects of the component:* This description would have to analyze all possible uses in all possible contexts.

Recognizing that this goal may not be achievable, we nonetheless advocate its pursuit because progress towards it will still reduce the burden of creating assurance argumentation and speed the development of critical systems. Thus, our final objective is:

**Our Goal:** Assurance argumentation should be compositional.

### 3 TECHNICAL BACKGROUND

This section introduces the background of the modeling technologies, example, and safety methodologies used in the paper.

#### 3.1 AADL and OSATE

The *Architecture Analysis and Design Language* (AADL) is an internationally standardized architecture description language [19]. AADL, which has both a textual and graphical syntax, is supported by the *Open Source Architecture Tool Environment*<sup>1</sup> (OSATE), which is a development environment based on the Eclipse IDE. System designers can use OSATE and AADL to model their system’s software elements (e.g., thread, process, subprogram), hardware elements (e.g., processor, memory, bus), and the connections and bindings between them [6]. Annexes extend the core language to address different, non-architectural aspects of system design, such as behavior or data modeling.

One such extension that this work relies heavily upon is the error modeling annex [20], which includes mechanisms for specifying *error types*, which are errors that can be instantiated and propagated as tokens between components (similar to Wallace’s Fault Propagation and Transformation Calculus [26]). For example, a sensor that produces a reading that may be higher than the actual value would be modeled as being a source propagation for tokens of the Value High error type. *Error propagations* represent the broadcast or reception of error tokens from/into components, typically via ports. This is used to represent a component producing erroneous output or receiving erroneous input, and can be used to trace the path that errors take through a system, e.g., a flawed sensor value is transformed by a software controller into an inappropriate command, which is transformed into a potentially unsafe actuation by a servo. The error modeling annex comes with a user-extensible library of error types, which is organized hierarchically into broad categories of error [16]. The type system is quite flexible, and can be used by a modeler to represent arbitrary error conditions, potentially including, e.g., the state of the system’s environment.

#### 3.2 An Example System: PulseOx Forwarding

To illustrate the features of ASAP, we use an illustrative, open-source<sup>2</sup> example from the medical domain. In addition to a control loop with multiple inputs and outputs, it has a single safety concern, which is the failure to issue a necessary alert. It consists of the following elements:

- **Hardware Devices** – Represented as AADL devices

<sup>1</sup><https://osate.org/>

<sup>2</sup><https://github.com/osate/osate2-asap/tree/main/org.osate.asap.examples>

---

```

1package PulseOx_Forwarding_Logic
2public
3  -- Import statements elided for space
4
5  process PulseOx_Logic_Process
6    features
7      LogicSp02 : in data port
6        ⇐ PulseOx_Forwarding_Types::Sp02;
8      LogicDerivedAlarm : out event port
9      {MAP_Properties::Output_Rate => 200 ms .. 400
6        ⇐ ms;};
10   properties
11     MAP_Properties::Process_Type => logic;
12     MAP_Properties::Component_Type => controller;
13   annex EMV2 {**
14     use types PulseOx_Forwarding_Errors;
15     error propagations
16       LogicSp02: in propagation {Sp02ValueHigh,
6        ⇐ Sp02ValueLow, EarlySp02, LateSp02,
6        ⇐ NoSp02, ErraticSp02};
17       LogicDerivedAlarm: out propagation
6        ⇐ {MissedAlarm, BogusAlarm};
18     end propagations;
19   **};
20 end PulseOx_Logic_Process;
21 process implementation PulseOx_Logic_Process.imp
22 subcomponents
23   CheckSp02Thread : thread CheckSp02Thread.imp;
24   Sp02Val : data PulseOx_Forwarding_Types::Sp02
25   {MAP_Error_Properties::Process_Variable => true;};
26 connections
27   outgoing_alarm : port CheckSp02Thread.Alarm ->
6     ⇐ LogicDerivedAlarm;
28 end PulseOx_Logic_Process.imp;
29
30 thread CheckSp02Thread
31 features
32   Alarm : out event port;
33 properties
34   Thread_Properties::Dispatch_Protocol => Periodic;
35 end CheckSp02Thread;
36 thread implementation CheckSp02Thread.imp
37 end CheckSp02Thread.imp;
38
39end PulseOx_Forwarding_Logic;

```

---

**Listing 1: An example of AADL’s textual syntax, showing the specification of a software controller.**

- *pulseOx*: A pulse oximeter device, which measures the blood oxygen saturation (SpO<sub>2</sub>) of a patient via a non-invasive fingerclip.

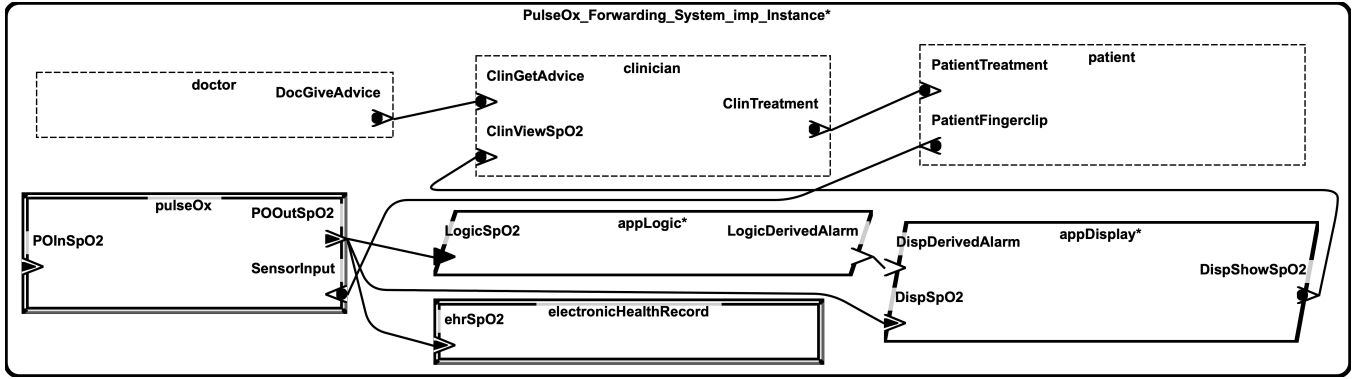


Figure 1: The PulseOx Forwarding System, in AADL's graphical syntax

- *electronicHealthRecord*<sup>3</sup>: An adapter for the electronic health record, which records the patient's SpO<sub>2</sub>.
- **Software Processes** – Represented as AADL processes
  - *appLogic*: Simple application logic that triggers an alert if the patient's SpO<sub>2</sub> is too low.
  - *appDisplay*: Simple logic to display both the alert (if present) and the patient's SpO<sub>2</sub>.
- **Humans** – Represented as AADL abstracts
  - *clinician*: A clinician who monitors the display and treats the patient.
  - *doctor*: A doctor who advises the clinician but does not directly treat the patient.
  - *patient*: The patient who provides SpO<sub>2</sub> readings (via the PulseOx) and receives treatment.

Figure 1 shows the overall system in AADL's graphical syntax. This system is not complex, but has two aspects which make it ideal for demonstrating ASAP's features: it shows multiple *control loops*, i.e., circular paths through the system which involve both sensing and actuating; and some components (i.e., the processes) decompose cleanly into subcomponents (in this case a collection of communicating threads). Some of these components would be already be modeled as part of a normal system engineering process, while others (i.e., the humans) would typically need abstractions created specifically for safety analysis, e.g., to represent particular execution or interaction scenarios. Exactly which ones would need to be created for ASAP is impossible to specify without knowing what other analyses are being run on the system: modelers are typically encouraged to add only as much detail as required by the analyses they want to perform.

<sup>3</sup>We recognize that this is not how electronic health records are typically used, here we use the term for a more generic data store.

### 3.3 STPA and SAFE

The *System Theoretic Process Analysis* (STPA) is a hazard analysis that is designed to address many of the criticisms found with more reliability-oriented, hardware-focused analysis techniques [11, 13]. It has been adapted to work with AADL models without significant modification to the process [17].

The *Systematic Analysis of Faults and Errors* (SAFE) is a hazard analysis technique that is heavily derivative of STPA, but contains a number of modifications, including a formal definition of hazard [15]. This, along with additional specificity available with low-level architectural specifications such as AADL models, enables the new automation discussed in below. We discuss it, STPA, and SAFE in more depth as we explore ASAP in Section 4.

## 4 THE ARCHITECTURE-SUPPORTED AUDIT PROCESSOR

The Architecture-Supported Audit Processor (ASAP) is a plugin to OSATE that enables three “viewpoints” of a system. These viewpoints are diagrams and tables which are dynamically generated (i.e., in response to user input) and designed to support activities performed by system safety auditors. ASAP's viewpoints are generated from the system architecture as modeled in OSATE and supplementary safety information, entered by the system designer or analyst.

The first viewpoint is somewhat abstract and presents high-level *fundamental* aspects of the system (or component's) safety. The second presents elements in their immediate context, and the third focuses narrowly on the causes (and potential compensations) of errors within a component. This progression from abstract to specific is typical of both model-based system design in AADL and hazard analysis techniques such as SAFE and STPA. This progression means that more shallow analyses can be performed using less time: a simpler model and less ASAP-specific annotations will produce less rich diagrams and information. Alternatively,

different portions of the system can be modeled to different depths: a subcomponent whose behavior should be more deeply analyzed can be richly annotated while others are left essentially unspecified as “black boxes.” This lets designers focus on the portions of the system that are relevant to particular stakeholders without getting bogged down in creating sophisticated models solely for the purpose of enabling tool functionality.


#### 4.1 Viewpoint 1: Fundamentals

In order to orient the analysis towards particular safety issues, STPA and derivative analyses such as SAFE make explicit the links between low-level faults and errors and high-level safety concerns such as death or injury to a human. This is done by creating a *fundamentals hierarchy*, a structure that relates safety problems, and their solutions, to specific and general notions of accidents / losses. At the top of the hierarchy are *accident levels*, which are broad categories of harm that can be prioritized. A typical system might have death or injury to a human as the highest-ranked accident level, followed by damage to or destruction of mission equipment. *Accidents* are losses that can be caused by the system; any number of accidents can be linked to a single accident level. That is, there might be multiple specific accidents that would each result in harm to a human. A diagrammatic view of a fundamentals hierarchy is shown in Figure 2(a).

Linking concrete losses resulting from system failure (accidents) to the specific ways they occur (hazards) is a significant open challenge in designing architecturally-integrated safety analysis techniques. The difficulty comes in concisely specifying which system elements could be involved in causing a particular accident, and how the failure of those elements would cause the loss. Specifying the links between accidents and the system elements involved in their causation is necessary as these links form the context required to both understand the impact of system design choices and to construct coherent argumentation.

*Hazards* have a two-part definition in SAFE (which formalizes STPA’s definition<sup>4</sup>) as a combination of one system and one environment state that will cause an accident; note that multiple hazards can lead to a single accident. Intuitively, this two-part definition results from the notion that certain system behaviors are rarely always unsafe, but rather only unsafe given a particular state of the environment. Leveson uses the example of a train: it is only unsafe for train’s doors to be open while the train is moving [11]. In ASAP, hazards are modeled using the condition that causes them<sup>5</sup>; i.e., as

the arrival of an error type at a component via its interface (e.g., a port), see Figure 2(b).

Figure 3 shows the *fundamentals* viewpoint in ASAP which supports STPA’s first step. As shown by the lower portion, hazards contain a number of references to model elements including: a) *Accident*: The accident the hazard’s occurrence would cause, b) *Environment Element*: The component whose state is the environmental “half” of the hazard, c) *System Element*: The component whose state is the system half of the hazard, d) *Error Type*: The AADL error type representing the system element’s deviation from intended or acceptable behavior, and e) *Hazardous Factor* A human-readable name of what is being transmitted from the system element to the environment element<sup>6</sup>. Note that five of the nine elements linked to from the hazard (i.e., all those with icons other than ) are semantic objects in either the AADL or ASAP model, as opposed to plain text. By semantic objects, we mean that these refer to actual elements in the model, represented in OSATE as rich data structures with links to other elements. This helps keep the documentation synchronized with the model, and enables the query-driven behavior of the other viewpoints.

#### 4.2 Viewpoint 2: Connected Neighbors

In hierarchically-organized system models of any useful size, it can be difficult to understand how a particular component fits into the larger system. Even the relatively simple PulseOx Forwarding system can be difficult to quickly understand: there are multiple cyclic control flow paths as well as multiple levels of abstraction; we use model slicing to reduce the complexity [24]. AADL is purpose-built for hierarchically specifying system details.

STPA uses scoped control flow diagrams to present components in context in its second step, however, so we developed the *Connected Neighbors* viewpoint to show a given component, its immediate neighbors (i.e., those components that either produce input for the component or use its output), their neighbors, and any connections between the displayed components. AADL models contain all of this information already, the ASAP tooling extracts it automatically rather than requiring the diagrams to be constructed manually. Figure 4 shows an example of this viewpoint centering on the app-Logic component of the PulseOx Forwarding system. Note that some elements, such as the electronicHealthRecord or doctor, are too distant<sup>7</sup> from the focused element, and thus are not displayed.

<sup>4</sup>STPA’s full definition is “A system state or set of conditions that, together with a particular set of worst-case environmental conditions, will lead to an accident (loss).” [11, pg. 184]

<sup>5</sup>Note that in STPA, hazards can be either states or conditions (Leveson has discussed their equivalence [11]) but in ASAP they must be conditions.

<sup>6</sup>see, e.g., Ericson’s text for the role of hazardous factors in accidents [4].

<sup>7</sup>Distance here is the number of “hops” from a given component, i.e., those which intransitively interfere according to van der Meyden’s definition [25].

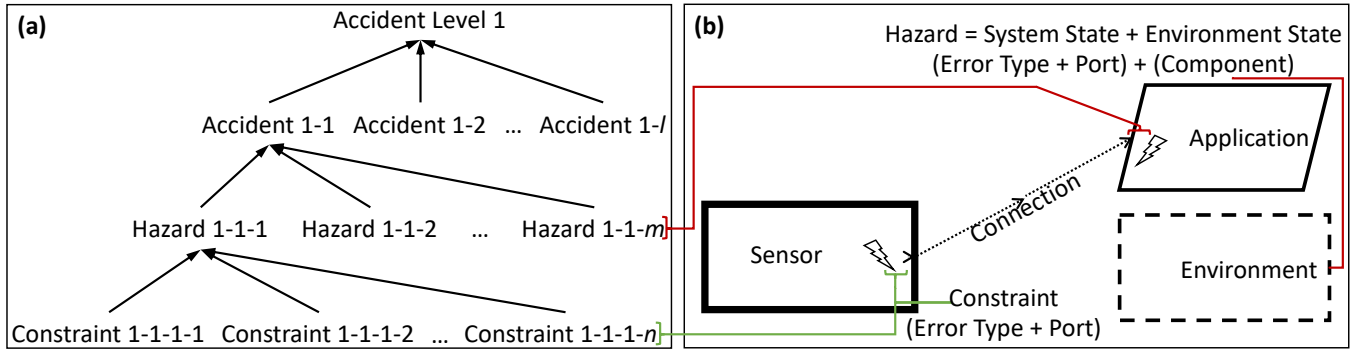


Figure 2: Part (a): A diagrammatic view of the fundamentals hierarchy. Part (b): How hazards and constraints are modeled in AADL.

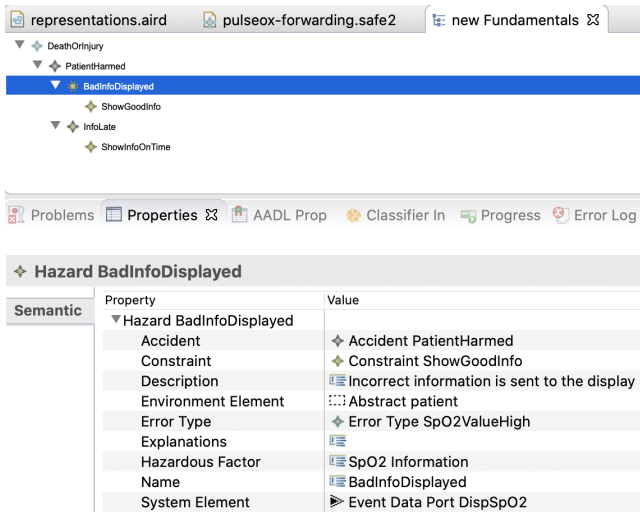


Figure 3: A fundamentals hierarchy (i.e., an instantiation of Figure 2a) in the ASAP tool.

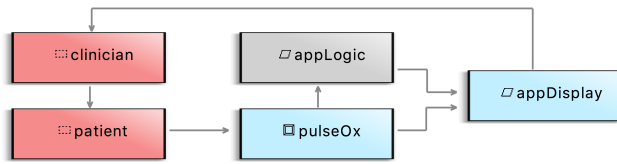


Figure 4: The Connected Neighbors view of the appLogic component in the ASAP tool. The primary element is shown in grey, immediate neighbors are blue, and the immediate neighbors of the neighbors are red. Connections represent the flow of data or commands. Note that this is a subgraph of Figure 1

Our goal with this view is not to replace system-level views like what AADL presents, but rather to support analyst intuition and rapid understanding of a particular component's

“point of view” of a system. For example, any input from the doctor that affects the application logic would first have to be understood by the clinician, which would affect the treatment administered to a patient, which would be detected by the PulseOx, at which point it would be received by the appLogic. Put another way, the doctor’s impact on the appLogic is mediated by three different system elements and thus may not be immediately relevant for gaining quick understanding, so the doctor does not appear as a neighbor of the appLogic in Figure 4.

### 4.3 Viewpoint 3: Unsafe Control Actions

ASAP’s third viewpoint displays information on how system hazards might come to occur, and how they could be prevented.

**4.3.1 Preliminaries: Causes, Compensations, and Guidewords.** In any non-trivial system, there are a large number of ways that things can go wrong; in a safety-critical system these are documented as violations of the system’s safety constraints. We refer to these violations as *causes*, i.e., ways that hazards (and associated losses) are caused. When thinking about a cause, the solution may or may not be apparent. If it is, a safety analyst should document it, we refer to these solutions (which may be partial, or conditioned on some other system behavior) as *compensations*. There is again a challenge in organizing and presenting a large amount of highly contextual information: a simple listing of causes and compensations is much harder to use than one organized, for example, around the system architecture or a taxonomy of system error types. These error types are equivalent to *guidewords* in hazard analysis techniques, which can be thought of as generic causes that are designed to prompt (i.e., guide) the thinking of analysts to consider various ways in which system elements might fail.

In addition to the manually-specified causes and compensations, however, a second form of loss scenario specification



emerges naturally from a fully specified EMV2 model of a system. Given a specification of how an error can come about (i.e., an error source), be transformed by various components (error transformations), and its final result (error sink), analysts can gain a fairly clear description of how a failure might occur. Because this form of scenario specification is machine-readable, tooling can also interpret these loss scenarios for various purposes such as building fault trees or calculating failure rates. This view is not present in ASAP, but it is available from other safety analyses in OSATE [2].

**4.3.2 A Hierarchical Table.** STPA's third step involves identifying control actions that could be unsafe. Typically, this identification is performed as analysts fill in a table where each row is a *control action* and there are four columns, one for each way STPA suggests a control action could be unsafe [13].

ASAP's version of this table is shown in Figure 5, note that there is an X for a connection (row) in the *ItemTimingError* (column). This denotes that documentation exists regarding the cause of a safety constraint violation involving the specified connection and the timing of messages being sent across it. A second table, from the same viewpoint can be generated that displays the same set of communication channels (i.e., rows) but the columns changed to show only the timing family of errors in the given component: here the full cause description (as well as optional compensation description) is shown. Identifying these scenarios where accidents / losses can occur is the fourth and final step of STPA.

The information required for these tables is pulled directly from both the system model (i.e., its error propagations), and the ASAP-specific fundamentals model, completing the deep integration of system model and hazard-analysis data. That is, there are two sources that are queried for each cell in the table: the "Fundamentals," which were created for ASAP's first viewpoint as well as the error propagations specified in the AADL model itself. If analyst-provided cause and / or compensation information is available, it is displayed; if only an error propagation indicates the presence of a problem, the text "Undocumented error propagation" is displayed instead.

Note that while the rows in the second, *refined* Unsafe Control Action table are the same as in the overview table, the columns can be errors from any error family, i.e., any of the abstract guidewords used in the model. While these column headings could be the guidewords from STPA, they could also be from AADL's Error Library (as in Figure 5) or any other set of guidewords / error collection. That is, while ASAP's *Unsafe Control Actions* viewpoint (i.e., the top-level table shown in Figure 5 and the refined version of the table) are usable for STPA's third and fourth steps, it has two enhancements.

First, the rows are not restricted to just control actions, but instead include all connections in the selected system / component. This change was made because some problems can be associated with non-control actions (sensor readings can be incorrect, electrical or hydraulic power can be over- or under-supplied, etc.) Recognizing and documenting these potential problems directly—instead of only when they manifest as unsafe control actions—is more concise. Additionally, distinguishing between a control action and sensor feedback is difficult to do consistently: it depends on the analyst's judgment and the component's role within the system.

Second, as ASAP is not directly tied to any particular hazard analysis process, the columns are generic: they are derived from the top-level error types used in the AADL model, rather than being unchangeable. An analyst could certainly choose to use STPA's set of guidewords, or they could use error types from the AADL Error Library, a custom set, or one derived from / aligned with a particular safety standard required by their company or the domain they are working in. Several such sets of system errors exist; Procter and Feiler have described AADL's Error Library and its relationship to guidewords used in hazard analyses [16].

**4.3.3 A Hierarchical Taxonomy.** Building a taxonomy of guidewords is rarely the primary goal of a research effort, however; typically guidewords are created as part of developing a new hazard analysis method. It is a challenge, though, to balance the goals of being both sufficiently expressive (so analysts do not miss potential causes) while also not being overly prescriptive (which can make the analysis unwieldy and verbose). In ASAP, we addressed this problem by defining two levels of tables which can be generated for every component, but this relies on a hierarchical organization of guidewords / error types.

The two-level approach taken by ASAP relies on a hierarchical specification of system error, i.e., a generic error type must be refinable into a set of more specific error types. AADL's EMV2 supports just such an approach [20]; using it, system modelers or safety analysts can define custom error types, and then refine those into more specific types. Alternatively, the EMV2 standard comes with a predefined Error Library, which contains a straightforward decomposition of standard system errors [16]. Typically, users combine the two approaches: they begin with the Error Library's set of error types and refine those to align with their domain or system. These custom error types are supported by the Unsafe Control Actions tables, so domain-specific extensions to the model will be fully incorporated. See Figure 6, which shows a graphical view of the EMV2 library's hierarchy of timing-related errors, extended with custom error types specific to the PulseOx Forwarding application described in Section 3.2.



Communication Channels (ie, control actions and sensor feedback)	Top-Level Errors (ie, abstract guidewords)			
	ItemValueError	ItemTimingError	ViolatedConstraint	ServiceError
patient.PatientFingerclip -> pulseOx.SensorInput	X	X		
pulseOx.POOutSpO2 -> electronicHealthRecord.ehrSpO2	X	X		
doctor.DocGiveAdvice -> clinician.ClinGetAdvice				
pulseOx.POOutSpO2 -> appLogic.StoreSpO2Thread.incoming_spo2				
appDisplay.DispShowSpO2 -> clinician.ClinViewSpO2				
clinician.ClinTreatment -> patient.PatientTreatment				
appLogic.CheckSpO2Thread.Alarm -> appDisplay.HandleAlarmThread.Ala...				
pulseOx.POOutSpO2 -> appDisplay.UpdateSpO2Thread.SpO2	X	X		

X means one or more errors in this family can propagate on this channel

Figure 5: Unsafe Control Actions table generated on the PulseOx Forwarding System.

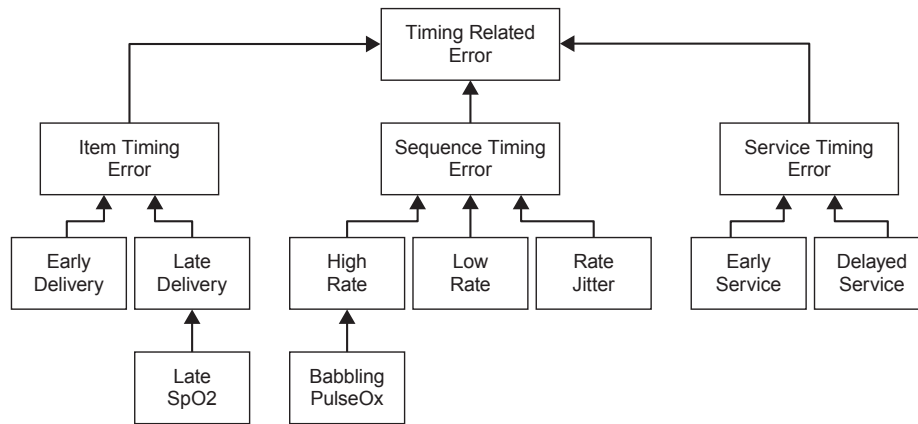


Figure 6: Hierarchy of timing errors, adapted from [16].

#### 4.4 Tying it together: Focus

A common feature in diagrams used for safety analysis is a way to highlight a particular component or fundamental as well as those related to it, i.e., some way to call attention to system elements related to a specific hazard, accident, component, error etc. For fundamentals, this task is straightforward: we simply highlight the higher-level fundamentals which contain the focus target (i.e., move up the tree from Figure 2(a)) as well as all the lower-level fundamentals it contains (i.e., the subtree rooted at the focus target). Handling a focused Hazard or Constraint requires additional effort, though: recall that those fundamentals contain references to the system model, i.e., to error propagations occurring at specific component ports. Thus we include predecessors and successors, which are the components that might cause or be affected by the associated error propagation.

Thus, focusing on an error propagation or system component is significantly more complex than focusing on an Accident or Accident Level. We note that there is a related challenge in static analysis of software: it is often necessary

to determine which program statements could have affected, or have been affected by, the program's state at a specific point in the source code. This issue is typically addressed through the use of program *slicing*, which Silva describes as “a technique for decomposing programs by analyzing their data and control flow” [23]. There are a number of program slicers available, we use one that has been built to run on AADL models [24].

A *backward slice* finds system elements (or error propagations) which could potentially affect the focused system element (or cause the focused error propagation). Correspondingly, a *forward slice* finds system elements (or error propagations) which could potentially be affected by the focused system element (or have been caused by the focused error propagation).

#### 4.5 Discussion

A safety analyst who wishes to use ASAP today, i.e., given existing regulatory regimes, will find it most useful to use

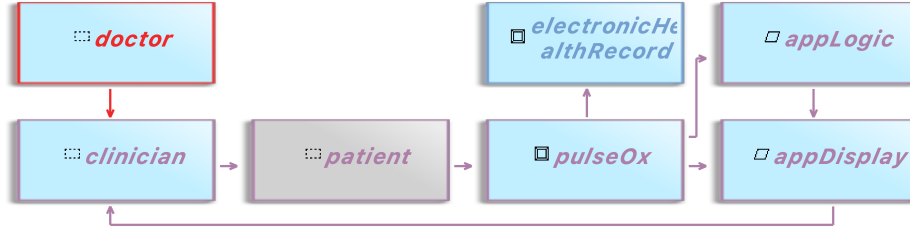


Figure 7: The Connected Neighbors view of the Patient, which has been selected as the current focus. Red elements are reachable from the focused element only via a backwards traversal. Blue elements are reachable only via a forwards traversal. Purple elements are reachable via both.

the tool while developing the argument for certification authorities. They might begin by specifying metadata about her system using Viewpoint 1, check the inputs and outputs of a high-criticality component using Viewpoint 2, and then explore undocumented causes and compensations using AADL’s EMV2 and Viewpoint 3’s tables. In the longer term, we envision safety processes that support the interactive, query- / view-driven approach explored in this work directly.

## 5 RELATED WORK

There is a large body of work that considers how the creation of systems and safety documentation should be automated. We differentiate the approaches into two groups, those whose automation is independent of a system’s hierarchical decomposition and those who integrate with it more deeply. We discuss these groups with representative technologies.

### 5.1 Automated Assurance Cases

AdvoCATE [3] is software developed by NASA that brings considerable automation to the creation of *assurance cases*, i.e., structured arguments that follow a defined, logical format. They incorporate both arguments and evidence, but are typically not as deeply integrated into a system’s architecture as the arguments in ASAP. The methods of argument traversal are, however, similar to ASAP in that AdvoCATE supports, e.g., hierarchical abstraction and queries / views. However, these are queries and views of the argument itself rather than the system under analysis as was our goal.

### 5.2 Hierarchical Safety Analysis

Hierarchically Performed Hazard Origin and Propagation Studies (HiP-HOPS) [22] takes into account system hierarchy and supports compositionality. Compared to ASAP, it uses a more traditional (i.e., not system-theoretic) model of safety and accident causality, and it operates on systems in MATLAB rather than AADL / OSATE. HiP-HOPS’ primary goal is the generation of hazard analysis reports, rather than

deeply integrating safety information and argumentation into a system’s architecture.

## 6 FUTURE WORK

### 6.1 Autogenerating Causes and Impacts

As discussed in Section 4.3 a fully-specified EMV2 description of a system encodes a machine-readable error path / event chain. In addition to the analyst-supplied narrative now visible in the unsafe control actions table, we are interested in transforming these event chains into something human readable. These causal event chains can be calculated using AWAS’s backward slicing functionality [24] (starting from either a constraint violation or an arbitrary error occurrence), though how these chains are best displayed to the user remains an open question.

AWAS’s forward slice calculates the errors and failures resulting from an error’s occurrence. These impacts could also be useful, though work would need to be done to align them with existing safety notions from, e.g., academic literature and / or safety standards. Once aligned, the best format for their presentation to the user would also need to be determined.

### 6.2 Alignment with Requirement Specifications

The goals and activities involved in safety engineering overlap to some extent with those involved in specifying a system’s requirements. We expect requirement specification may evolve, somewhat, with the use of ASAP and are interested in exploring ways of supporting and automating more rigorous requirement specifications. However, we recognize that specifying functional and safety requirements simultaneously is not a straightforward task. There is research in this area from both system theoretic safety [21] and architecture-centric perspectives [14]; we are interested

in seeing the extent to which ASAP's viewpoints can be extended or supplemented with additional requirement detail or traceability information.

## 7 CONCLUSION

In this paper we presented the Architecture-Supported Audit Processor, or ASAP, tool as well as its motivation and underlying theory. We applied it to a small example, demonstrated how it aligns with and improves upon a popular system-theoretic hazard analysis, and discussed possible avenues for future improvements.

## ACKNOWLEDGEMENTS

Copyright 2022 Carnegie Mellon University.

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

NO WARRANTY. THIS CARNEGIE MELLON UNIVERSITY AND SOFTWARE ENGINEERING INSTITUTE MATERIAL IS FURNISHED ON AN "AS-IS" BASIS. CARNEGIE MELLON UNIVERSITY MAKES NO WARRANTIES OF ANY KIND, EITHER EXPRESSED OR IMPLIED, AS TO ANY MATTER INCLUDING, BUT NOT LIMITED TO, WARRANTY OF FITNESS FOR PURPOSE OR MERCHANTABILITY, EXCLUSIVITY, OR RESULTS OBTAINED FROM USE OF THE MATERIAL. CARNEGIE MELLON UNIVERSITY DOES NOT MAKE ANY WARRANTY OF ANY KIND WITH RESPECT TO FREEDOM FROM PATENT, TRADEMARK, OR COPYRIGHT INFRINGEMENT.

[DISTRIBUTION STATEMENT A] This material has been approved for public release and unlimited distribution. Please see Copyright notice for non-US Government use and distribution. DM22-0095

## REFERENCES

- [1] Association for the Advancement of Medical Instrumentation. 2000. *ANSI/AAMI/ISO 14971: Medical devices—Application of risk management to medical devices*. Technical Report. ANSI/AAMI/ISO.
- [2] Julien Delange and Peter Feiler. 2014. Architecture Fault Modeling with the AADL Error-Model Annex. In *2014 40th EUROMICRO Conference on Software Engineering and Advanced Applications*. IEEE, Verona, Italy, 361–368.
- [3] Ewen Denney and Ganesh Pai. 2018. Tool support for assurance case development. *Automated Software Engineering* 25, 3 (2018), 435–499.
- [4] Clifton A. Ericson II. 2016. *Hazard Analysis Techniques for System Safety* (second ed.). John Wiley & Sons, Inc., Fredericksburg, Virginia, United States of America. 1–640 pages.
- [5] Huáscar Espinoza, Alejandra Ruiz, Mehrdad Sabetzadeh, and Paolo Panaroni. 2011. Challenges for an open and evolutionary approach to safety assurance and certification of safety-critical systems. In *Proceedings - WoSoCER 2011 - In Conjunction with ISSRE 2011*. IEEE, Hiroshima, Japan, 1–6.
- [6] Peter Feiler and David Gluch. 2012. *Model-Based Engineering with AADL* (1st ed.). Addison-Wesley Professional, Upper Saddle River, NJ. i–468 pages.
- [7] Brendan Hall, Jan Fiedor, and Yogananda Jeppu. 2020. Model Integrated Decomposition and Assisted Specification (MIDAS). *INCOSE International Symposium* 30, 1 (jul 2020), 821–841.
- [8] International Organization for Standardization. 2018. *ISO 26262-10: Road vehicles—Functional safety—Part 10: Guidelines on ISO 26262*. Technical Report.
- [9] Leslie Lamport. 2012. How to write a 21 st century proof. *Journal of Fixed Point Theory and Applications* 11, 1 (mar 2012), 43–63.
- [10] Nancy Leveson. 1995. *Safeware: System Safety and Computers*. Addison-Wesley.
- [11] Nancy Leveson. 2011. *Engineering a Safer World: Systems Thinking Applied to Safety*. MIT Press.
- [12] Nancy Leveson. 2020. Are you sure your software will not kill anyone? *Commun. ACM* 63, 2 (jan 2020), 25–28.
- [13] Nancy Leveson and John Thomas. 2018. *STPA Handbook*. Technical Report. 1–188 pages.
- [14] Anitha Murugesan, Sanjai Rayadurgam, and Mats Heimdahl. 2019. Requirements reference models revisited: Accommodating hierarchy in system design. In *Proceedings of the IEEE International Conference on Requirements Engineering*. 177–186.
- [15] Sam Procter. 2016. *A Development and Assurance Process for Medical Application Platform Apps*. Ph.D. Dissertation. Kansas State University.
- [16] Sam Procter and Peter Feiler. 2018. The AADL Error Library : An Operationalized Taxonomy of System Errors. In *HILT 2018*. Boston, MA.
- [17] S. Procter and J. Hatcliff. 2014. An architecturally-integrated, systems-based hazard analysis for medical applications. In *MEMOCODE 2014*.
- [18] John Rushby. 2010. Formalism in Safety Cases. In *Making Systems Safer: Proceedings of the Eighteenth Safety-Critical Systems Symposium*. Springer, London, Bristol, UK, 3–17.
- [19] SAE AS-2C Architecture Analysis and Design Language Issuing Committee. 2017. *Architecture Analysis & Design Language (AADL)*.
- [20] SAE AS-2C Architecture Description Language Subcommittee. 2015. *SAE Architecture Analysis and Design Language (AADL) Annex Volume 1: Annex E: Error Model Language*. Technical Report. SAE Aerospace.
- [21] Andrea Scarinci, Amanda Quilici, Danilo Ribeiro, Felipe Oliveira, Daniel Patrick, and Nancy Leveson. 2019. Requirement Generation for Highly Integrated Aircraft Systems Through STPA: An Application. *Journal of Aerospace Information Systems* 16, 1 (jan 2019), 9–21.
- [22] Septavera Sharvia and Yiannis Papadopoulos. 2015. Integrating model checking with HiP-HOPS in model-based safety analysis. *Reliability Engineering & System Safety* 135 (2015), 64–80.
- [23] Josep Silva. 2012. A Vocabulary of Program Slicing-Based Techniques. *Comput. Surveys* 44, 3, Article 12 (June 2012), 41 pages.
- [24] Hariharan Thiagarajan, John Hatcliff, and Robby. 2020. Awaz: AADL Information Flow and Error Propagation Analysis Framework. In *European Conference on Software Architecture (ECSA20)*. Springer, Cham, L'Aquila, Italy, 294–310.
- [25] Ron Van Der Meyden. 2007. What, Indeed, Is Intransitive Noninterference?. In *Proceedings of ESORICS 2007: 12th European Symposium On Research In Computer Security*. Springer Berlin Heidelberg, Dresden, Germany, 235–250.
- [26] Malcolm Wallace. 2005. Modular Architectural Representation and Analysis of Fault Propagation and Transformation. In *FESCA 2005*, Vol. 141. 53–71.