Chapter 2
Model-Based Design and Automated Validation of ARINC653 Architectures Using the AADL

Jérôme Hugues and Julien Delange

Abstract Safety-Critical Systems as used in avionics systems are now extremely software-reliant. As these systems are life- or mission-critical, software must be carefully designed and certified according to stringent standards. One typical pitfall of corresponding development project is the late detection of safety issues or bugs at integration time that impose to redo development steps. Model-Based Engineering aims at capturing system concerns with specific notations and use models to drive the development process through all its phases—design, validation, implementation and ultimately, certification. Through a single consistent notation, such an approach would avoid undefined assumptions and traditional hurdles due to informal, text-based, specifications. In this chapter, we present recent contributions we pushed forward in the AADL architecture description language for the design and validation of Integrated Modular Avionics systems. First, we review modeling patterns to support abstractions for Integrated Modular Avionics systems. We then introduce capabilities to check all ARINC653 patterns are enforced at model-level. In addition, we review error modeling and safety analysis capabilities towards the production of safety reports conforming to ARP4761 recommendations, along with code generation strategies to map model elements to code. All these contributions are integrated in one uniform modeling process based on the AADL.

Keywords AADL · EMV2 · Safety analysis · Code generation · ARINC653

J. Hugues
Institut Supérieur de l'Aéronautique et de l'Espace,
Universit de Toulouse, 31055 Toulouse, France
e-mail: jerome.hugues@isae.fr

J. Delange (✉)
Carnegie Mellon Software Engineering Institute, Pittsburgh, USA
e-mail: jdelange@sei.cmu.edu

S. Nakajima et al. (eds.), *Cyber-Physical System Design from an Architecture Analysis Viewpoint*, DOI 10.1007/978-981-10-4436-6_2

2.1 Introduction

Safety-Critical Systems (as the ones used in avionics, aerospace or automotive domains) are becoming extremely software-reliant. Boeing's new 787 Dreamliner contains more than 6.5 millions lines of software code [6]. Between 2006 and 2012, the software of the F-35 increased from 6800 K to 24000 KSLOCS [16].

While this trend brings many benefits such as ease of update or upgrade, re-use of software across different product lines; it also introduces new challenges.

As software is being updated and upgraded, it becomes more complex with many collocated functions on several processors that may interact (i.e. bus connections, interference between collocated tasks). In such environment, a single software error might have significant impacts: a report claims that 50% of car warranty costs are now related to electronics and embedded software [6].

In the avionics domain, a software error can have dramatic consequences. Such systems must be carefully engineered according to stringent standards such as DO178C [19], which mandates analysis, testing and certification activities. Unfortunately, ensuring compliance with this standard is labor-intensive and costly. As the development process is mostly manual and paper-driven, many errors are introduced. This adds significant rework efforts, cost and postpones product delivery.

During the last two decades, new standards have been defined to facilitate the development of safety-critical systems. In the avionics community, the ARINC653 standard [1] focuses on isolating software in partitions so that they meet higher safety requirements while reducing the number of CPUs. Such approaches help system designers structuring their architectures; but they still need to validate software isolation and deliver assurance of implementation correctness. As development activities are loosely coupled, many errors are introduced early on and impact other activities.

As a result, efforts made in early development phases, such as requirements and architecture, spread over the development process so that more than 60% of development efforts are focused on implementation and testing [7].

This motivates a Model-Based development approach for avionics software. Models use a formal representation of the system and constitute the core of the development workflow: they are processed across the development process to validate, implement and test the system. In order to implement such a development process, designers need an appropriate language to represent system architecture with their specific characteristics using an appropriate modeling language. Using a formal language reduces errors from textual specifications and undefined assumptions while automating the development process ensures that the system is correctly validated and implemented according to the specifications.

This chapter presents our recent contributions to the Architecture Analysis and Design Language (AADL) [21] for modeling and validating avionics architectures. Through a motivational example, we illustrate how model-based help detecting safety errors early while providing support for assurance cases.

We first illustrate how to represent Integrated Modular Avionics (IMA) architectures using the ARINC653 annex [22] so as to capture specific requirements (such as isolation properties). In addition, we present our validation rules that analyze models and ensure correctness of requirements enforcement. We then focus on safety analysis, and the production of various reports mandated by safety assessment authorities, as well as code generation strategies targeting avionics-grade operating systems.

These combined validations and automated report and code generation facilities are the foundations to validate the architecture and ultimately generate safety reports, paving the path towards system airworthiness at model-level.

2.2 Boeing 777 ADIRU Case Study

On 1 August 2005 a serious incident involving Malaysia Airlines Flight 124, occurred when a Boeing 777-2H6ER flying from Perth to Kuala Lumpur also involved an ADIRU fault resulting in uncommanded maneuvers by the aircraft acting on false indications [2]. The ATSB (Australian Safety Authority) found that the main probable cause of this incident was a latent software error that allowed the ADIRU to use data from a failed accelerometer. The ATSB report [18] indicates that a model of Software Health Management (SHM) for the Boeing 777 Air Data Inertial Reference Unit (ADIRU) is involved in the incident; and provides a full explanation of the incident.

The Architecture of the Boeing 777 ADIRU (Fig. 2.1) has multiple levels of redundancy. Two ADIRU units are used, primary and secondary. The primary ADIRU consists of 4 Fault Containment Areas (FCA). Each FCA contains multiple Fault Containment Modules (FCM). The ADIRU system can continue to work without maintenance if only one fault appears in each FCA. The system can still fly with 2 faults, but it needs maintenance before next flight. The Secondary Attitude Air Data Reference Unit (SAARU) also provides inertial data to flight computers. The flight computers use the middle value between the data provided by the ADIRU and SAARU.

In the report, it was revealed that accelerometer number 5 had failed in June 2001 and could still produce high acceleration values or voltages that were erroneous. This failure was identified and accelerometer number 5 was excluded from use in acceleration computation by ADIRU subsequently. On the day of the accident ADIRU went through a power cycle. Afterwards a second accelerometer failed (number 6). This failure was identified and accelerometer number 6 was excluded. But unexpectedly the software allowed the previous failed accelerometer number 5 to be used in acceleration computation, so the high value acceleration data was produced and output to the flight computer. Then the accident occurred. The reason why the failed accelerometer number 5 was reused in acceleration computation is the ADIRU software error in the algorithm that failed to recognize accelerometer number 5 as unserviceable after a power cycle.

Such an error results from incomplete system-level engineering that did not foresee such—apparently—basic situation. We claim these can be avoided by using

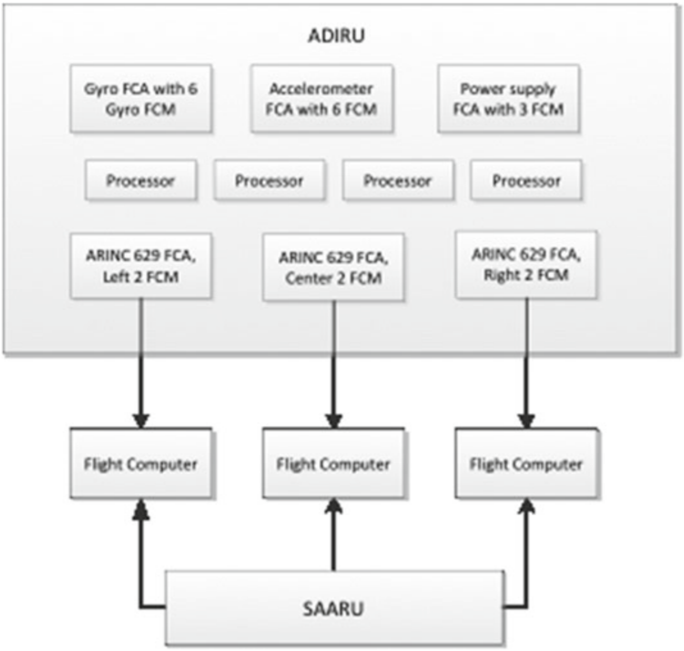

ADIRU
Gyro FCA with 6 Gyro FCM
Accelerometer FCA with 6 FCM
Power supply FCA with 3 FCM
Processor
Processor
Processor
Processor
ARINC 629 FCA, Left 2 FCM
ARINC 629 FCA, Center 2 FCM
ARINC 629 FCA, Right 2 FCM
Flight Computer
Flight Computer
Flight Computer
SAARU

Fig. 2.1 Architecture of the Boeing-777 ADIRU, from [2]

rigorous, tool-supported engineering process. In the following, we revisit this example, introduce the core AADL language and detail the specific modeling patterns from the ARINC653 annex [22] to represent avionics architectures. We then detail how we use and extend the Resolute [15] language to validate IMA requirements in AADL models and generate assurance case.

2.3 AADL and Patterns for IMA System

In this section, we review the AADL core language and extensions we proposed as part of the SAE AS2-C committee to model avionics systems based on the Integrated Modular Avionics paradigm.

2.3.1 The AADL Core Language

The Architecture Analysis and Design Language (AADL) [21] is a modeling language standardized by SAE International. It defines a notation to model software components, the deployment/configuration on a hardware platform, and its interaction with a physical system within a single and consistent architecture model. The core language specifies several categories of components with well-defined semantics. For each component, one defines both a component type to represent its external interface; along with one or more component implementations.

For example, the task and communication architecture of the embedded software is modeled with threads and processes that are interconnected with port connections, shared data access and remote service calls.

The hardware platform is modeled as an interconnected set of buses and memory components, with virtual processors representing hierarchical schedulers, and virtual buses representing virtual channels and protocol layers. A device component represents a physical subsystem with both logical and physical interfaces to the embedded software system and its hardware platform.

System components are used to organize the architecture into a multi-hierarchy. Users model the dynamics of the architecture in terms of operational modes and different runtime configurations through the mode concept.

Users can further characterize standardized component properties, e.g., by specifying the period, worst-case execution time for threads. The language is extensible; users may adapt it to their needs using two mechanisms:

1. **User-defined properties** New properties can be defined by users to extend the characteristics of the component. This is a convenient way to add specific parameters in the model (for example, criticality of a subprogram or task)
2. **Annex languages** Specialized languages can be attached to AADL components to augment the component description through additional characteristics or requirements (for example, specifying the component behavior [14] by attaching a state-machine). They are referred to as annex languages, meaning that they are extensions to a component.

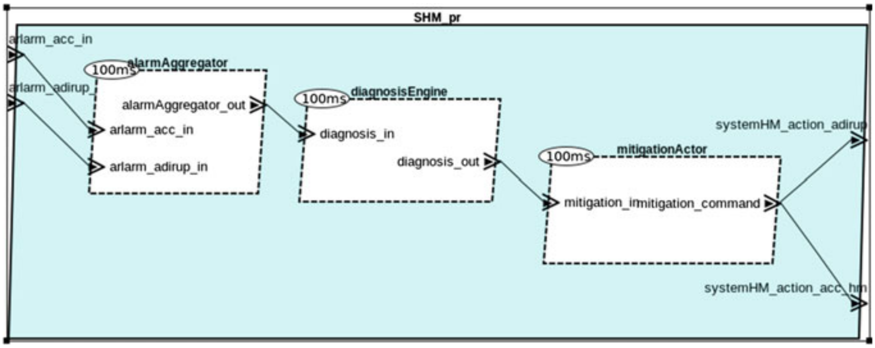


Fig. 2.2 ADIRU AADL model example - graphical representation

AADL provides two views to represent models:

1. **The graphical view** outlines components hierarchy and dependencies (bindings, connection, bus access, etc.). While it does not provide all details, this view is useful for communication and documentation purposes.
2. **The textual view** shows the description, with component interfaces, properties and annexes. It is appropriate for users to capture internal details and for tools to process and analyze the system architecture from models.

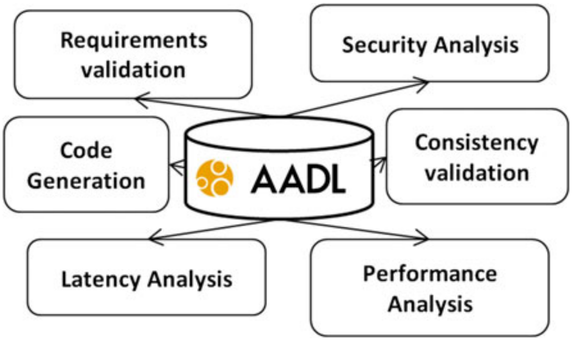
Figure 2.2 represents a simple AADL model (excerpt from the ADIRU case study) that contains three threads that communicate together. The corresponding textual notation of this model is shown in Listing 2.1.

```
process implementation systemHM_process.impl
subcomponents
  alarmAggregator: thread threads::alarmAggregator.impl;
  diagnosisEngine: thread threads::diagnosisEngine.impl;
  mitigationActor: thread threads::mitigationActor_th.impl;
connections
  C1: port arlarm_acc_in<->alarmAggregator.arlarm_acc_in;
  C2: port arlarm_adirup_in<->alarmAggregator.arlarm_adirup_in;
  -- [..]
end systemHM_process.impl;
```

Listing 2.1 ADIRU AADL example model - textual representation

The AADL model annotated with properties and annex language clauses is the basis for analysis of functional and non-functional properties along multiple dimensions from the same source, and for generating implementations, as shown in Fig. 2.3. We discuss existing analysis tools and methods to analyze and validate AADL models in Sect. 2.4.

Fig. 2.3 AADL ecosystem



2.3.2 Modeling Integrated Modular Architectures with AADL

The goal of the Integrated Modular Avionics (IMA) concept is to integrate software components in common hardware modules. The IMA approach aims at deploying the same portable software component on different execution hardware and to change their configuration or communication policy without impacting functional aspects (and the associated code). However, such design flexibility requires capability to analyze and verify the architecture to ensure that application requirements (such as deadline, memory or safety) are met. For example, relocating a software component from one processor to another affects the scheduling and latency. Having the ability to capture this change, analyze the architecture and validate these requirements will simplify the design of such architectures.

The AADL language can be leveraged to capture IMA architectures, its software aspects and its configuration and deployment policy. For description of software aspects, AADL provides the following components:

- **data** components represent types used on components interfaces (AADL `data` and `event data port`) to communicate values between functions.
- **subprogram** components capture logical software units executed by an ARINC653 process (AADL `thread` component). Such components can be realized using traditional languages (e.g. Ada or C) or functional models (e.g. Simulink or SCADE).
- **thread** components specify a task executing code (AADL `subprogram`). This concept is similar to a POSIX thread (also called process by some standard such as ARINC653). Timing requirements of AADL `thread` components (e.g. period, deadline, execution time) are specified using the core properties.
- **process** components provide a separated memory space hosting several tasks (AADL `thread` components); UNIX processes.

For configuration and deployment, AADL includes the following components:

- **bus** components represent the physical (e.g. wires) and logical (e.g. protocol) connections (i.e. ethernet bus) that transport data across physical nodes.
- **memory** components capture a physical memory (i.e. RAM) and its logical decomposition (i.e. memory segments).
- **processor** components represent a physical processor as well as the execution runtime (i.e. operating system such as Linux or, in the context of IMA systems, an ARINC653 module).

These components are integrated to capture the software platform and its execution runtime. The deployment policy can be easily modified in the model by changing component associations so that engineers can evaluate pros and cons of different configurations. Ultimately, AADL models are analyzed by tools to evaluate the architecture metrics (e.g. latency [11] or scheduling [8]). Let us note this approach is currently transitioning to the industry [17].

The core AADL language provides an accurate semantics to capture IMA principles (deployment of software components on different execution architectures, components reuse, etc.). However, as such, it does not support the representation of some specific characteristics of IMA architecture, in particular, the Operating System. IMA systems use an ARINC653-compliant Operating System, a time and space partitioning system that separates software into partitions. While AADL provides the capability to represent such Operating System, it still needs to be extended with specific properties and modeling patterns. Next sections introduce the AADL ARINC653 annex, a standardized document that specifies how to capture ARINC653-compliant systems with the AADL.

2.3.3 The AADL ARINC653 Annex

This section introduces the AADL ARINC653 annex [22] that provides the ability to model ARINC653-compliant architectures. Models that are compliant with this annex are conformant with the IMA principles but also specific ARINC653 constraints, such as time or space isolation.

The AADL ARINC653 annex [22] defines modeling patterns and specific properties to represent ARINC653 [1] platform requirements. ARINC653 defines specific concepts (e.g. time and space partitioning) that require additional AADL properties to be captured. Standardizing modeling patterns provides guidance to represent ARINC653 systems so that AADL users and tools use the same modeling patterns, making model analysis portable.

The next paragraphs introduce the modeling patterns and specific properties of the AADL ARINC63 annex and discuss the recent additions made to this standard document, especially regarding the specification of the Health Monitoring policy.

2.3.3.1 ARINC653 Module

An ARINC653 module is captured using a `processor` component that represents the physical processor and the isolation layer (e.g. the isolation kernel ensuring time and space separation between partitions). This component also contains `virtual processor` components, each one representing a partition.

The `processor` defines the time isolation policy of the `ARINC653` module: time slots durations and assignments to each partition. This requirement is captured by the `ARINC653::Module_Schedule` property. Partition should have at least one time slot to be sure it is executed at each module period.

2.3.3.2 ARINC653 Partition

An ARINC653 partition is defined by two main assets:

- **application software** it is associated with the partitions as ARINC653 processes, using AADL `thread` components. It is important to distinguish the difference between an AADL `process` component and an ARINC653 process, as the same word are used in both standards with different semantics. The former represents the ARINC653 application software partition while the latter is a task and mapped to an AADL `thread` component.
- **execution runtime** resources available to execute the application software partition. It is captured using an AADL `virtual processor` associated with the ARINC653 module (AADL `processor`).

The application software (AADL `process`) is associated with an execution runtime using the AADL processor bindings (property `Actual_Processor _Binding`). It is also associated with a memory segment (i.e. space isolation) using AADL memory association bindings (property `Actual_Memory_Binding`).

2.3.3.3 Memory Configuration

In an ARINC653 architecture, the main memory is separated in several disjoint segments. Each segment is associated with one partition, ensuring space isolation between collocated partitions.

The main memory component (e.g. RAM) is captured using a `memory` component. The policy is specified decomposing this AADL component using `memory` sub-components, each one representing one segment. Memory segment characteristics (e.g. segment size, base address) are specified by attaching AADL properties the corresponding AADL `memory` components.

Then, each segment is allocated to one partition using the AADL memory binding mechanism (property `Actual_Memory_Binding`). This is specified by associating the partition application software (e.g. AADL `process` component) to the memory segment (e.g. AADL `memory` component).

2.3.3.4 Scheduling Parameters

ARINC653 mandates a hierarchical scheduling approach with two levels:

1. The **module level** schedules partitions using a fixed, predictive time-line algorithm repeated at a given rate (called the Major Frame).
2. The **partition level** schedules ARINC653 processes (AADL `thread`) within partitions. This is policy is partition-dependent and relies on the mechanisms supported by the underlying partition execution runtime.

The scheduling policy at the *module level* is specified in AADL processor components, which represent ARINC653 modules. The ARINC653::Module_Schedule property defines a list of time slots for each partition, while the partitions' scheduling rates are defined by the ARINC653::Module_Major_Frame property.

The scheduling policy at the *partition level* is defined in AADL virtual processor components which represent the partition execution runtime. The scheduling policy is captured with the AADL Scheduling_Protocol property.

2.3.3.5 Intra-partition Communications

ARINC653 Intra-Partition communication are channels between ARINC653 processes (or AADL thread components) located within the partition. Such channels are confined within the partitions and do not require any specific capability from ARINC653 module. The standard distinguishes several intra-partition mechanisms: buffers, blackboards, events and semaphores which are translated in AADL using (respectively) event data ports, data ports, event ports and shared data components.

The associated AADL data classifier associated to a port represents the type of data used by the communication channels (ARINC653 buffers and blackboards). In addition, AADL properties (e.g. Queue_Size) are attached to AADL interfaces to represent specific requirements (e.g. number of data instances within a buffer, management of history, etc.).

2.3.3.6 Inter-partitions Communications

ARINC653 inter-partitions communications specify communication channels between partitions. This type of communication must be explicitly configured by the ARINC653 operating system. Only declared channels can be created and used at runtime. Correct configuration and implementation of this mechanism ensure space isolation across partitions by avoiding data leakage between partitions classified at different assurance level.

The ARINC653 standard distinguishes two types of inter-partitions communications: queuing and sampling ports, which are mapped in AADL using (respectively) event data ports and data ports. Ultimately, partitions ports are connected to tasks in order to represent data usage and explicitly capture what software component (thread or subprogram) processes and uses it.

2.3.3.7 Health-Monitoring Policies

ARINC653 health-monitoring policy configures error detection mechanisms and associated mitigation and recovery strategy to keep the system in a safe state.

Table 2.1 Mapping rules between ARINC653 and AADL concepts

ARINC653 concept	AADL concept
Module	Processor component
Partition	Process component bound to a virtual processor component and a memory component
Space isolation	Decomposition of physical memory components into logical memory components
Time isolation	Each partition (process component) is bound to a virtual processor component that is itself bound to a processor component (representing the module)
Process	Thread component
Queuing ports	Event data ports across process components
Sampling port	Data ports across process components
Buffer	Event data port across thread components
Blackboard	Data port across thread components

The general concept is that each potential error (e.g. a divide by zero exception, inconsistent memory access, etc.) is associated with a recovery action such as restarting the partition where the fault originated. The ARINC653 standard distinguishes three levels of health monitoring: module, partition and process.

The ARINC653 AADL annex defines a simple approach to map Health Monitoring policies with two properties:

- `ARINC653::HM_Error_ID_Levels` defines the level for each system error and at which level it is detected and eventually recovered;
- `ARINC653::HM_Error_ID_Actions` defines the set of recovery actions for each error that can be detected by the ARINC653 executive.

These properties replace the approach from previous versions of the ARINC653 AADL annex which used properties at each health monitoring level. Because the health monitoring policy was spread over the component hierarchy, this former method was confusing as it might introduce non-deterministic specifications (for example, having the same error handled at several levels which is not a legal ARINC653 specification).

2.3.3.8 Mapping Rules

Between AADLv2 and ARINC653 concepts are summarized in Table 2.1.

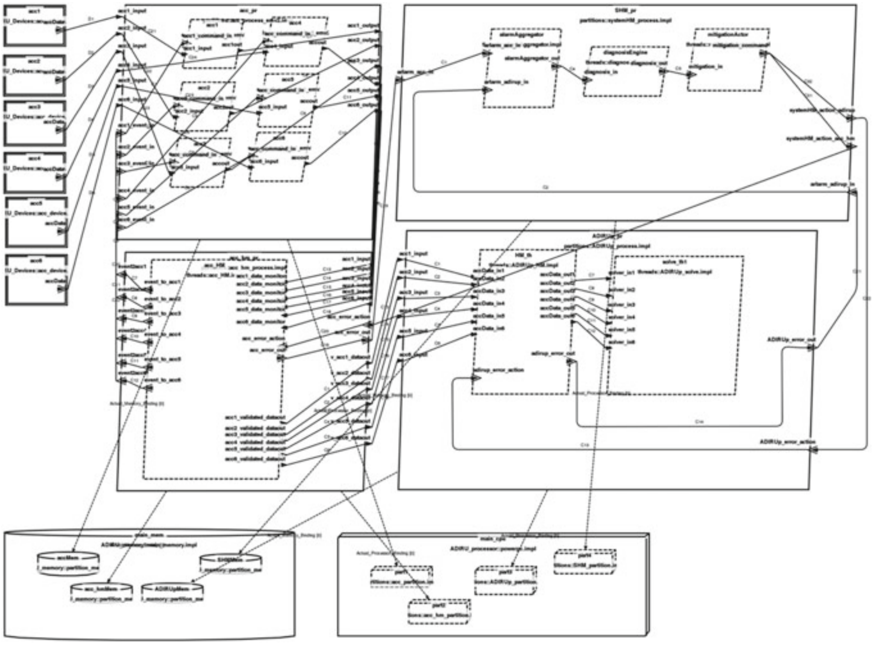


Fig. 2.4 ADIRU full model (graphical)

2.3.4 ADIRU Full Model

We captured in a set of AADL packages the full model.[1] It represents 1.5 KSLOCs of models and capture all facets of the functional architecture of the ADIRU, along with its scheduling and memory configuration parameters. A graphical representation is shown in Fig. 2.4.

Let us note this model has a high level of complexity, and captures all facets of the initial cas e study. Yet, it remains of tractable size thanks to the compact textual representation. From this model, several analyses can be performed. In the following, we focus on three of them: model-based assurance, safety analysis and code generation. For each, we explain how the model has been extended to address a particular concern.

2.4 Model-Based Assurance with AADL

The system architecture captured in AADL can be processed and validated against system requirements. Such analysis can be processed by specific tools that browse models components, extract properties and evaluate their correctness against the requirements.

[1] The model is available as part of the AADLib library of models: http://www.openaadl.org/aadlib.html.

This validation process is useful but is difficult to analyze, especially when there are a lot of inter-dependent results. Investigating analysis results and finding potential issues can be challenging, especially when architectures have inter-dependent requirements. In order to make the analysis review easier, we extend our analysis tool and auto-generate an assurance case from the validation results. This shows the inter-dependencies of each requirements using a hierarchical notation and details which ones are not enforced. The assurance case, associated with the validation results, constitutes an indicator of the system architecture quality (how many requirements are covered and validated in the architecture).

The next sections introduce the Resolute validation language, our extension for producing assurance case from validation results and how we apply this technique to produce assurance cases to check IMA requirements.

2.4.1 Validation of AADL Models

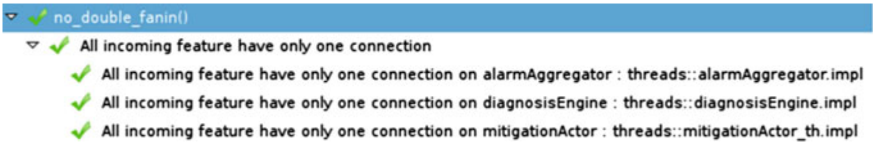
Analysis tools process AADL models and automatically check their correctness with regard to specific quality attributes (e.g. security, safety, performance). To date, AADL has been already successfully used to validate several quality attributes such security [12], performance, [11] or safety [9]. Analysis functions have been designed in the Open Source AADL Tool Environment (OSATE) [5], an Eclipse-based modeling framework. Analysis tools are implemented as Eclipse plug-ins that browse the components hierarchy, retrieve informations from the components (through AADL `properties` or `annex` languages) and produce an analysis report.

However, writing analysis methods as Eclipse-plugins require learning the internals of the modeling tool, study the Eclipse platform as well as the AADL meta-model. This makes the design of new analysis features difficult for non computer-science experts, which reduce the development of new analysis capabilities. Model analysis can be implemented using general constraints language (such as OCL [20]) but such approaches are often difficult to use, complicated to use and not user-friendly [4].

```
no_double_fanin() <=
  ** " All incoming feature have only one connection" **
  forall (c : component) . true => has_single_fanin (c)


has_single_fanin (comp : component) <=
  ** "All IN features have one connection on " comp **
  forall (f : features (comp)) . (direction(f) = "in")
                       => (length (connections (f)) = 1)
```

Listing 2.2 Example of a RESOLUTE theorem - check for single fanin



```
▿ ✓ no_double_fanin()
    ▿ ✓ All incoming feature have only one connection
          ✓ All incoming feature have only one connection on alarmAggregator : threads::alarmAggregator.impl
          ✓ All incoming feature have only one connection on diagnosisEngine : threads::diagnosisEngine.impl
          ✓ All incoming feature have only one connection on mitigationActor : threads::mitigationActor_th.impl
```

Fig. 2.5 RESOLUTE analysis result

To overcome this issue, the AADL community proposed a specific extension (through the annex mechanism of AADL), RESOLUTE [15], to process and analyze a model with a specific, user-friendly query language. It allows system designers to write new analysis methods within the modeling platform without having to learn the basics of Eclipse plug-ins development or any details of the AADL meta-model. When analyzing a model, RESOLUTE produces a hierarchical graphic representation of the execution results (as in Fig. 2.5).

Listing 2.2 shows an example of a resolute theorem that checks all incoming interfaces of all components are connected to a single source. Figure 2.5 shows the graphical representation of the analysis results when using this theorem on the AADL model introduced previously in Fig. 2.2 and Listing 2.1. In this example, the analysis passes: all incoming interface is connected to a single source The graphical representation of analysis results helps system designers to automatically check AADL models against specific requirements. A complete description of the language and analysis tools and more details can be found in [15].

2.4.2 Application to ARINC653 Requirements

We implemented ARINC653 validation rules using the RESOLUTE [15] language introduced in Sect. 2.4, and the new capability to generate assurance case [3, 13] from analysis results with a GSN notation. We designed a library of predefined validation theorems that validate ARINC653 requirements in AADL by checking that:

- Each partition AADL `process` is associated with exactly one memory segment (AADL `memory` component) and one partition execution runtime (AADL `virtual processor` component).
- Each ARINC653 module (AADL `processor` components) specify the partitions scheduling policy (property `ARINC653::Module_Schedule`) and execute each partition at least once during each scheduling period.
- All ARINC653 processes (AADL `thread`) define their scheduling characteristics (e.g. dispatch protocol, period, deadline).
- The ARINC653 Health-Monitoring Policy address all potential errors that are listed in the ARINC653 standard (such as divide by zero, application error, module error, etc.).

- Each memory segment (AADL `memory` component) is associated with at most one partition (AADL `process` component).
- All queuing ports or buffers (represented with AADL `event data ports`) specify the maximum number of data instances they can store (property `Queue_Size`).

These rules have been written in a RESOLUTE theorem library and integrated in OSATE. System designers can then use them directly without having to write any additional code. Note that these rules do not directly address safety issues related to the incident from Sect. 2.2 but detect any deployment or configuration issue that can led to such issue. This is part of a model-development process that can catch several type of errors. The next paragraph presents how safety analysis can be performed from the same model and detect safety issues as the one from Sect. 2.2.

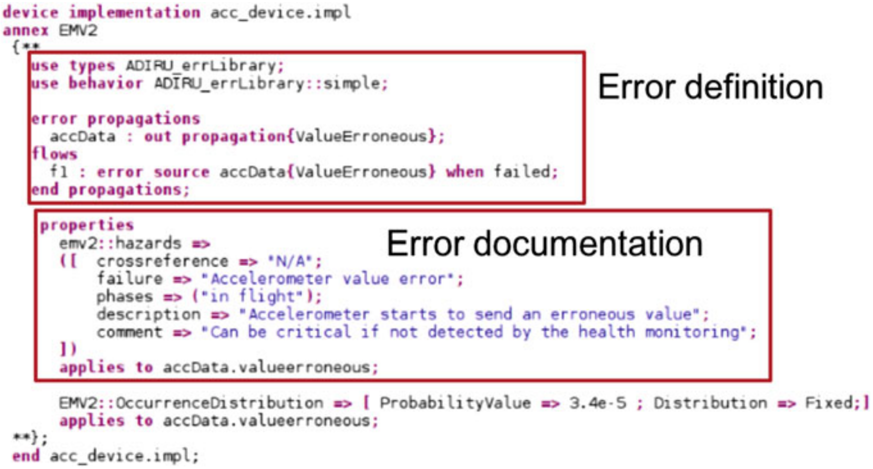
2.5 Safety Analysis

Aerospace Recommended Practice (ARP) 4761 from Society of Automotive Engineers (SAE) defines a process for using common modeling techniques, and analysis methods such as Functional Hazard Assessment (FHA), Fault Tree Analysis (FTA) or Fault Impact Analysis (FIA).

As part of the AS5506/A1 standard document [22], AADL has been enhanced with capabilities to support capture of erroneous behavior and to analyse their impact on the global architecture through the Error Modeling Annex v2 (EMV2 thereafter). EMV2 supports architecture fault modeling at three levels:

- Error propagation between components and their environment: Modeling of fault sources in a system, their impact on other components or the operational environment through propagation.
  It allows for safety analysis in the form of hazard identification, fault impact analysis, and stochastic fault analysis.
- Component faults, failure modes, and fault handling: Modeling of fault occurrences within a component, resulting fault behavior in terms of failure modes, effects on other components, the effect of incoming propagations on the component, and the ability of the component to recover or be repaired.
  It allows for modeling of system degradation and fail-stop behavior, specification of redundancy and recovery strategies providing an abstract error behavior specification of a system without requiring the presence of subsystem specifications.
- Focus on compositional abstraction of system error behavior in terms of its subsystems. It allows for scalable compositional safety analysis.

In addition, EMV2 introduces the concept of error type to characterize faults, failures, and propagations. It includes a set of predefined error types as starting point for systematic identification of different types of error propagations providing an error propagation ontology. Users can adapt and extend this ontology to specific domains. See [10] for a more detailed presentation of EMV2 features.



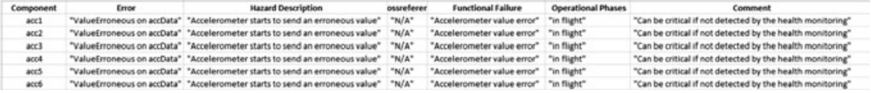
```
device implementation acc_device.impl
annex EMV2
 {**
 use types ADIRU_errLibrary;
 use behavior ADIRU_errLibrary::simple;

 error propagations
   accData : out propagation{ValueErroneous};
 flows
   f1 : error source accData{ValueErroneous} when failed;
 end propagations;

  properties
    emv2::hazards =>
    ([  crossreference => "N/A";
        failure => "Accelerometer value error";
        phases => ("in flight");
        description => "Accelerometer starts to send an erroneous value";
        comment => "Can be critical if not detected by the health monitoring";
    ])
    applies to accData.valueerroneous;

    EMV2::OccurrenceDistribution => [ ProbabilityValue => 3.4e-5 ; Distribution => Fixed;]
    applies to accData.valueerroneous;
 **};
 end acc_device.impl;
```

Fig. 2.6 Extension for safety concerns



Component	Error	Hazard Description	ossreferer	Functional Failure	Operational Phases	Comment
acc1	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"
acc2	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"
acc3	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"
acc4	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"
acc5	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"
acc6	"ValueErroneous on accData"	"Accelerometer starts to send an erroneous value"	"N/A"	"Accelerometer value error"	"in flight"	"Can be critical if not detected by the health monitoring"

Fig. 2.7 FHA report generated

We extended the previous model to add EMV2 concepts. Thanks to AADL encapsulation and inheritance mechanism, we can separate regular interfaces and properties from safety concerns (see Fig. 2.6).

By defining specific error types, propagations across components and error occurence probabilities, we can generate directly through OSATE2 analysis plug-ins both Fault Hazard Analysis (FHA) that captures the list of hazards that are derived from each function (Fig. 2.7) and Fault Trees (Fig. 2.8, note the figure is symetrical for the 6 accelerometers, we cropped it to make it legible) that represent the combinations of errors that could lead to a system-level failure.

Through Fault Impact and Faul Tree analyses, we could replay all scenarios depicted in the original case study, demonstrating the expressive power of EMV2 to capture in a concise way error propagation.

2.6 From Model to Code

The complete ADIRU system has been modeled in AADL thanks to the materials provided by the ATSB. To this model, we applied the validation scheme presented in Sect. 2.4.2 to the model, so as to validate the model against the ARINC653 requirements, and then perform safety analysis.

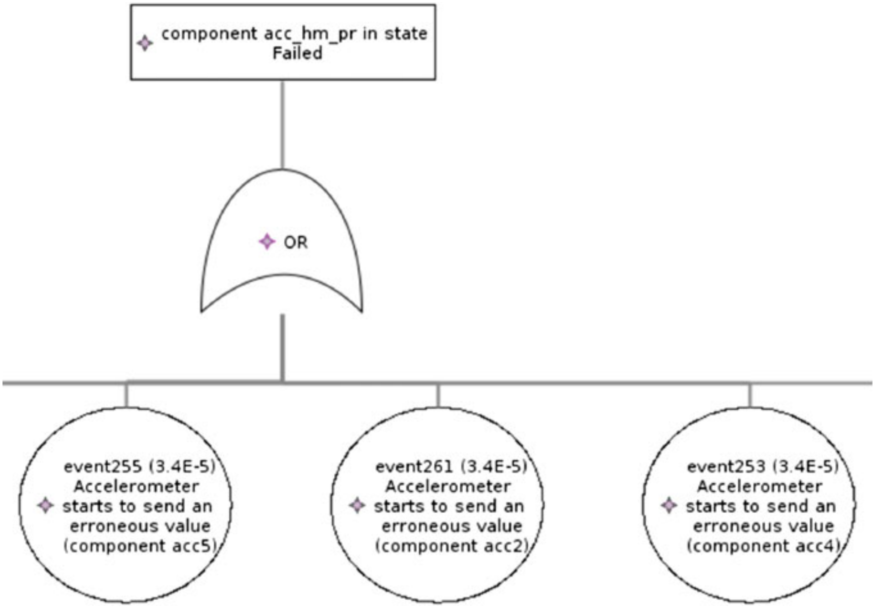

component acc_hm_pr in state Failed
OR
event255 (3.4E-5) Accelerometer starts to send an erroneous value (component acc5)
event261 (3.4E-5) Accelerometer starts to send an erroneous value (component acc2)
event253 (3.4E-5) Accelerometer starts to send an erroneous value (component acc4)

Fig. 2.8 FTA report generated

Let us note the AADL model captures all configuration parameters: partitions, buffers, link to memory segments, etc. We leveraged this information to validate the model is sound in the previous section. From this description, we can actually go further and also generate the corresponding source code.

To do so, we enhanced the Ocarina code generator [23] to target ARINC653 Application Executive (APEX), and more specifically DDC-I DeOS and WindRiver VxWorks653. These are extensions to previous work we did when targetting the FLOOS POK partitioned kernel. We rely on the Code generation annex that has been published in [22]. This annex provides guidelines to connect user-provided source code to a full distribution middleware generated from an AADL model.

Code generation covers two facets of the ARINC653 process:

1. Generation of XML descriptors: these configuration files describe all the resources required by the APEX to run: name and configuration parameters. These are derived from the definition of AADLv2 memory components that define the memory layouts, virtual processors capturing partitions and regular AADL threads, processes and subprograms;
2. Generation of code skeletons to be populated by the user source code. This codes provides regular patterns for periodic or sporadic activations, along with patterns for inter-/intra-partition communications using ARINC653 queueing and sampling ports.

Let us note that at this level, the initial AADLv2 model can be reused almost as-is. The only necessary addition is a property that indicates the APEX we want to target. This is captured in a property that is specific to our toolset.

Hence, from the AADL model, one can now generate a full prototype of the running application for both simulation and run-time validation purposes, combined with model-level validation. As such artefacts are usually produced manually by system engineers using (mostly) textual specification, generating them from a consistent semantics modeling language as AADL exhibits the following benefits:

1. **Costs reduction** as assurance cases can be automatically produced from models, it avoids labor-intensive costs to create them, especially considering the size of such documents on real systems. Also, because of this automation capability, the system's middleware code source can be updated and maintained as soon as the system is modified.
2. **Higher fidelity** using a semi-formal language (such as AADL) removes the errors related to informal specifications. It ensures the quality of the generated artifact and removes any indeterminism related to undefined assumptions made while reading non-formal, textual system specifications.

From the code generated, and its execution in a simulated environment, one can demonstrate all the scenarios that were initially tested as part of the safety analysis: simulation of the failure of a sensor, correct re-configuration of the system in the corrected revision and complete the performance analysis and evaluate the worst-case response time before complete recovery.

2.7 Conclusion and Future Work

Safety-critical software, such as the one used in avionics, must be carefully designed, validated and implemented. Such systems use dedicated execution platform (the IMA and its related ARINC653 operating system) and must comply with stringent certification standards (DO178C). Because of these requirements designing and maintaining such software is cost-intensive, and actual development methods no longer scale. Over the years, Model-Based approaches have shown interesting benefits to reduce development costs while maintaining (or even increasing) system quality.

In this chapter, we introduce our Model-Based Engineering approach to design and validate ARINC653 systems. We leverage the AADL language to capture IMA architecture and their requirements and propose specific modeling patterns for modeling ARINC653 operating systems.

In addition, we design a validation library for ARINC653 systems using RESOLUTE, an AADL-specific constraint language to analyze and validate software architectures. Using this tool, we are able to automate some system validation activities and auto-generate assurance cases, usually created manually. Such an automation might reduce manual efforts and keep certification documents up-to-date with the actual specifications.

Finally, we performed both safety analysis using AADLv2 EMV2 so that engineers can analyze fault impacts and generate safety documents (e.g. Fault-Tree Analysis, Failure Mode and Effects Analysis); and extended actual AADL code generation capabilities from the Ocarina toolset to auto-produce an implementation code that targets an ARINC653-compliant operating system.

Hence, we propose a large palette of tools to support all major activities for the engineering of safety-critical avionics system: full architecture capture, covering both functional and dysfunctional facets; analysis of system-level requirements on architecture artifacts, safety analysis and finally code generation. In our view, code generation in a simulated environment serves both for rapid prototyping and validation of many design choices.

All tools presented in this chapter are available through OSATE2 and Ocarina, and are available as free software. The ADIRU model is also fully public, to help the community better understand the close relationships between all those models. As we stated the model supports multiple verification and validation activities while remaining of modest size.

Future work will consider closer interaction with certification processes so as to align our contributions with current practices, but also challenge the benefits of model-based to reduce certification costs.

Acknowledgements Copyright 2016 Carnegie Mellon University. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8721-05-C-0003 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.

DM-0003495.

References

1. Airlines Electronic Engineering, Avionics application software standard interface—ARINC653. Technical Report (ARINC—Aeronautical Radio, Inc., 1997)
2. ATSB Transport Safety Investigation Report, In-flight upset event 240 km north-west of Perth, WA Boeing Company 777-200, 9M-MRG. Technical Report Aviation Occurrence Report 200503722 (ATSB, 2005)
3. R. Bloomfield, P. Bishop, Safety and assurance cases: past, present and possible future an adelard perspective, in *Making Systems Safer*, ed. by C. Dale, T. Anderson (Springer, London, 2010), pp. 51–67
4. J. Cabot, R. Clarisó, UML/OCL verification in practice, in *ChaMDE 2008 Workshop Proceedings: International Workshop on Challenges in Model-Driven Software Engineering* (2008), pp. 31–35

5. Carnegie Mellon Software Engineering Institute: OSATE—Open Source AADL Tool Environment. Technical report (2016), http://www.aadl.info
6. R.N. Charette, This car runs on code, in *IEEE Spectrum*, Feb 2009
7. B. Clark, R. Madachy, *Software Cost Estimation Metrics Manual for Defense Systems* (Software Metrics Inc., Haymarket, 2015)
8. J. Craveiro, J. Rufino, F. Singhoff, Architecture, mechanisms and scheduling analysis tool for multicore time-and space-partitioned systems. ACM SIGBED Rev. **8**(3), 23–27 (2011)
9. J. Delange, P. Feiler, D. Gluch, J.J. Hudak, AADL fault modeling and analysis within an ARP4761 safety assessment. Technical Report (2014)
10. J. Delange, P.H. Feiler, Architecture fault modeling with the AADL error-model annex, in *40th EUROMICRO Conference on Software Engineering and Advanced Applications, EUROMICRO-SEAA 2014*, Verona, Italy, 27–29 Aug 2014 (2014), pp. 361–368
11. J. Delange, P.H. Feiler, Incremental latency analysis of heterogeneous cyber-physical systems, in *Proceedings of 3rd IEEE International Workshop on Real-Time and Distributed Computing in Emerging Applications, REACTION 2014*, Rome, Italy, 2 Dec 2014 (2014)
12. J. Delange, L. Pautet, F. Kordon, Design, implementation and verification of MILS systems. Softw. Pract. Exper. **42**(7), 799–816 (2012)
13. E. Denney, G. Pai, J. Pohl., Advocate: an assurance case automation toolset, in *Proceedings of the 2012 International Conference on Computer Safety, Reliability, and Security, SAFECOMP 2012* (Springer, Berlin, Heidelberg, 2012), pp. 8–21
14. R. Frana, J.-P. Bodeveix, M. Filali, J.-F. Rolland., The AADL behaviour annex – experiments and roadmap, in *Engineering Complex Computer Systems* (2007), pp. 377–382
15. A. Gacek, J. Backes, D. Cofer, K. Slind, M. Whalen, Resolute: an assurance case language for architecture models, in *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology* (ACM, 2014), pp. 19–28
16. C. Hagen, J. Sorensen, Delivering military software affordably, in *Defense AT&L* (2013), pp. 30–34
17. A.V. Khoroshilov, I. Koverninskiy, A. Petrenko, A. Ugnenko, Integrating AADL-based tool chain into existing industrial processes, in *ICECCS* (2011), pp. 367–371
18. N. Mahadevan, A. Dubey, G. Karsai, A case study on the application of software health management techniques. ISIS-11-101, Jan 2011 (2011)
19. Military Aerospace, DO-178C nears finish line with credit for modern tools and technologies, May 2010
20. OMG, *UML 2.0 Specification* (Object Management Group, Final Adopted Specification, 2005)
21. SAE International, *AS5506B—Architecture Analysis and Design Language (AADL)*, Sept 2012
22. SAE International, *AS55061/A—SAE Architecture Analysis and Design Language (AADL) Annex Volume 1*, Oct 2015
23. B. Zalila, I. Hamid, J. Hugues, L. Pautet, Generating distributed high integrity applications from their architectural description

Springer

http://www.springer.com/978-981-10-4435-9

Cyber-Physical System Design from an Architecture
Analysis Viewpoint
Communications of NII Shonan Meetings
Nakajima, S.; Talpin, J.-P.; Toyoshima, M.; Yu, H. (Eds.)
2017, XIV, 159 p. 52 illus., 32 illus. in color., Hardcover
ISBN: 978-981-10-4435-9