

Model-Based Engineering for the Development of ARINC653 Architectures

Julien Delange, Olivier Gilles, Jérôme Hugues and Laurent Pautet

TELECOM ParisTech

Copyright © 2009 SAE International

ABSTRACT

The concept of partitioned kernel, introduced by the Integrated Modular Avionics (IMA) architecture comes with new challenges (isolation enforcement, partitioning trade-off, etc.) that must be addressed during the design and the implementation of partitioned architectures. However, the development process frequently consists in handwriting code, which makes difficult the analysis of the system. Such a development process does not ease the design of high-integrity systems.

Model Based Engineering describes architecture and application requirements with models. Models can be then used to ensure requirements enforcement or produce code, ensuring that requirements are enforced inside the implementation.

In this paper, we claim the Architecture Analysis and Design Language (AADL) as a valuable candidate to support a Model-Based method for the design and the implementation of ARINC653 systems. Using the AADL as a backbone language, we model such architectures using first-class citizen AADL constructs (virtual processors and buses) and detail their validation. We present our code generator that automatically assembles, configures and deploys application code from AADL models. We also discuss the efficiency of a model-based approach in the design of certifiable systems and explain how it helps the system integrator to certify its system.

INTRODUCTION

CONTEXT

Safety-critical (avionics, aerospace, etc.) systems come with a wide range of challenges (determinism, resource dimensioning) and inherent complexity (field buses, RTOS, etc.). In addition, these systems must be validated, simulated and certified according to a rigorous process. In the avionics domain, the ARINC653 [3, 4] standard introduces the concept of partitioning. It consists in isolating applications in space and time so each partition has an address space and a time slice to execute their code. Using a partitioning architecture, several applications at different criticality or security levels can be collocated on the same processor. Despite their criticality, the development of safety-critical systems is based on loosely coupled steps (manual validation, hand-written code, etc.). It makes difficult the design, implementation or certification due to the lack of consistency between each step of the development.

PROBLEM STATEMENT

Safety-critical systems are built according to a manual development process. However, it remains error-prone by nature and makes difficult the analysis of the architecture and the evaluation of potential issues. It's the state of the practice for developers manually write code after reading ARINC653 specifications. Obviously, no ones can guarantee specification enforcement from implementation code (this issue is addressed through

validation tests). Finally, certification of such systems is also issued manually, which is very long and expensive.

All these problems make the development of safety-critical systems very costly in terms of money, time and resources [5]. All steps of the design process are made manually and each remains error-prone, due to a lack of consistency in the whole development process (there is no clue that code follows specification, there is no proof that validation meets expected requirement, etc.)

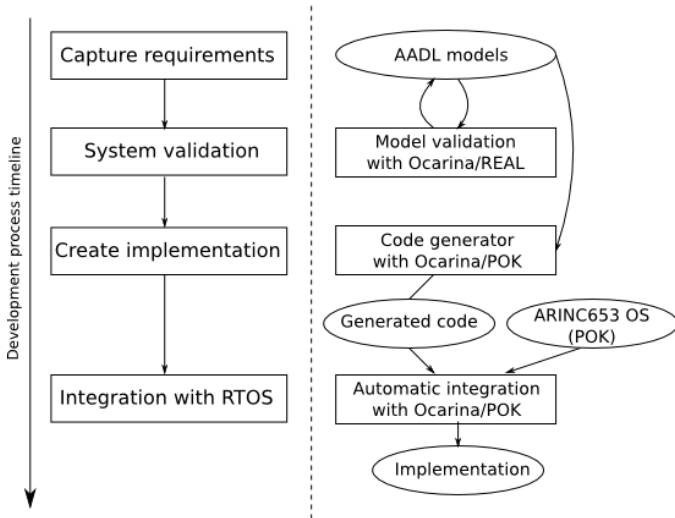


Figure 1 - Integration of our approach with a traditional development process

PROPOSED APPROACH

Model-Based Engineering provides a conceptual framework to use models as the central design. Architecture and application concerns are described with their properties. Model analysis offers a way to check system architecture with its requirements. Moreover, the implementation code can be derived from validated models. Indeed, the validation or verification effort would be useless if the implementation code is not derived from previously validated artifacts.

In our context, the modeling language must provide an appropriate semantics for modeling ARINC653 systems. We claim that the AADL [1, 10] is suitable for the modeling and the analysis of such architectures.

In this article, we propose a Model-Based development process for ARINC653 systems with the AADL as a backbone language. As illustrated in Figure 1, the use of a single modeling language for the whole development process improves its consistency and coherence. Our approach proposes a design and validation method and an automatic implementation tool.

At first, we propose a modeling approach to represent ARINC653 systems with their requirements with AADL. Our design method is composed of modeling patterns (how can we represent an ARINC653 partition using the AADL?) and dedicated properties to describe architecture requirements (what is the address space of this partition? What is its criticality level?). This design task is actually made by the system designer.

Then, we validate ARINC653 architectures using our dedicated AADL validation framework [6]. Such analysis tools check system requirements and resources allocation (enough space was allocated for a partition? has a partition enough time to execute its threads?). It also analyzes partitioning trade-off (can a partition affect another partition that is more critical?). Currently, this validation process is usually made manually by the system designer from hand-written specifications.

Finally, Ocarina, our AADL-to-C code generator, automatically creates the implementation code. The code follows the declarations of the AADL model and thus, contains appropriate constructions to enforce system requirements. Our code generator outputs ARINC653-compliant code and automatically creates ARINC XML deployment files. Thus, code can be integrated with any ARINC653-compliant OS. This task is manually done by developers and system integrators today.

The next section presents the AADL and describes modeling patterns and dedicated properties for the modeling of ARINC653 architectures. Then, we detail our validation rules to check ARINC653 requirements and partitioning trade-offs. Ocarina is also presented and we explain how we perform code generation from AADL models to concrete implementation code. Finally, we discuss the efficiency of our approach and its benefits regarding certification requirements.

ARINC653 SYSTEM MODELING

AADL is a component-centric language which allows the modeling of both software and hardware components. It focuses on the definition of component interfaces, separating them from implementations. The language defines both a graphical and a textual representation.

An AADL description is made out of components. The AADL standard defines basic component types that must be extended to add the features of a specific system. It defines software components (data, thread, subprogram, and process), execution platform components (memory, bus, processor, device, virtual processor, and virtual bus) and one hybrid component type (system).

Components describe well-identified elements of the actual architecture. The subprogram type models an application code fragment. Since a code fragment is not

at the level of an ARINC653 architectural element, it is used to a reference to another external model of piece of code (for example, a Simulink or Scade model). The `thread` type models the active part of an application. In our case, it corresponds to ARINC653 process. The `AADL process` type models an address space that contains `AADL thread` components. It corresponds to the address space of an ARINC653 partition (the runtime of the partition is represented by the `virtual processor` component). The `AADL processor` type models aspects of both the processor and the operating system. In our context, it corresponds to the ARINC653 module that enforces time and space partitioning. The `virtual processor` type models ARINC653 partition runtime (partition scheduler and associated resources) and is contained in `AADL processor` components to model the containment of partitions inside the ARINC653 module. The `memory` type models any kind of memory. In our context, this component models address spaces associated with partitions (`AADL process` components). The `bus` type models all kinds of networks and hardware connections. The `device` type models sensors, actuators or network interface and their role is to provide the functional interface to the hardware. The `virtual bus` type models layers in a protocol stack. Finally, the `system` type models composite components that are made up of hardware and software components.

Components are organized in a hierarchy, i.e.: components contain other components (called subcomponents in this case). AADL models contain a topmost `system` component that contains hardware and software subcomponents and their deployment.

The interface specification of a component is called its type. It provides features (e.g. communication ports). Components communicate one with another by connecting their features (the connections section). We map AADL ports to the ARINC653 ports. AADL ports that communicate across partitions are considered as ARINC653 inter-partition ports (queuing or sampling ports) while ports that communicate across threads are considered as ARINC653 intra-partition ports (blackboards, buffers).

Each component describes their internals: subcomponents, connections, etc in an implementation. An implementation of a thread or a subprogram can specify call sequences to other subprograms, thus describing the execution flows in the architecture. Since there can be different implementations of a given component type, it is possible to select the actual components to be put into the architecture, without having to change the other components, thus providing a convenient approach to application configuration.

AADL associates properties to AADL model elements. Properties are typed and represent name/value pairs that represent characteristics and constraints. Examples are the period and execution time of threads, the implementation language of a subprograms, etc.

Table 1 summarizes our AADL/ARINC653 mapping. In addition, we can describe other requirements of ARINC653. An AADL dedicated property set was written for this purpose and proposes a modeling approach for scheduling requirements or health monitoring. For a complete list of our mapping rules, readers would refer to the ARINC653 modeling annex of the AADL [2].

ARINC653 concept	AADL component
Process	Thread
Application Partition	Process, virtual processor and associated memory component
Module	Processor
Health Monitoring	Properties on processor, virtual processor and threads.

Table 1 - Summary of AADL/ARINC653 mapping

Figure 2 shows an ARINC653 system in AADL. It defines two partitions through different components (processes `partition1_process` and `partition2_process`, memories `part1memory` and `part2memory` and virtual processors `part1` and `part2`). These two partitions are collocated in the same ARINC653 module (the `cpu` processor component). We model two inter-partition communications (`queuing_conn` and `sampling_conn`). For that, we add one sampling port and one queuing port to each process that models a partition (`partition1_process` and `partition2_process`).

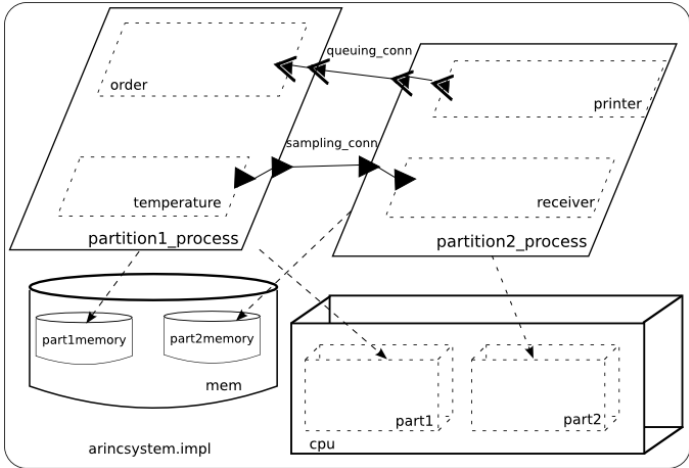


Figure 2 - An ARINC653 model in AADL

Using all these information, we are able to capture all requirements of ARINC653, including fine-grained specificities. The next section present validation pattern we perform to check requirements correctness.