

# Model-Based Engineering for the Development of ARINC653 Architectures

Julien Delange, Olivier Gilles, Jérôme Hugues, Laurent Pautet

TELECOM ParisTech  
LTCI UMR 5141  
46 rue Barrault  
F-75634 Paris, France  
lastname@telecom-paristech.fr

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

---

The Engineering Meetings Board has approved this paper for publication. It has successfully completed SAE's peer review process under the supervision of the session organizer. This process requires a minimum of three (3) reviews by industry experts.

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

ISSN 0148-7191

Positions and opinions advanced in this paper are those of the author(s) and not necessarily those of SAE. The author is solely responsible for the content of the paper.

**SAE Customer Service:** Tel: 877-606-7323 (inside USA and Canada)  
Tel: 724-776-4970 (outside USA)  
Fax: 724-776-0790  
Email: [CustomerService@sae.org](mailto:CustomerService@sae.org)

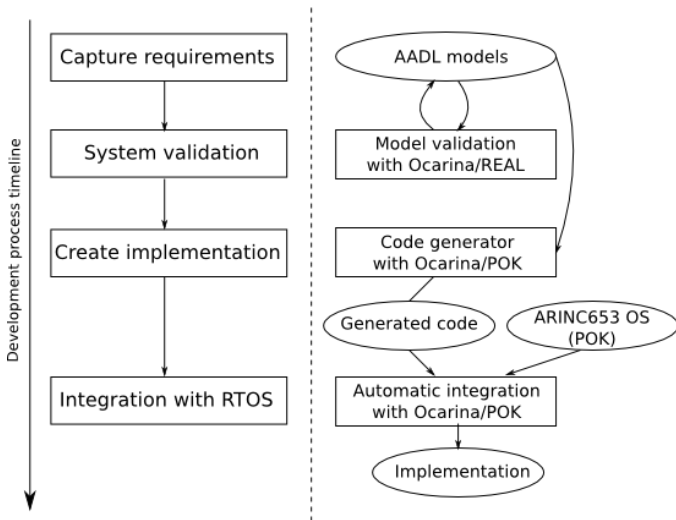
**SAE Web Address:** <http://www.sae.org>

Printed in USA

**SAE**International<sup>™</sup>

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 (queuing or sampling ports) while 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.

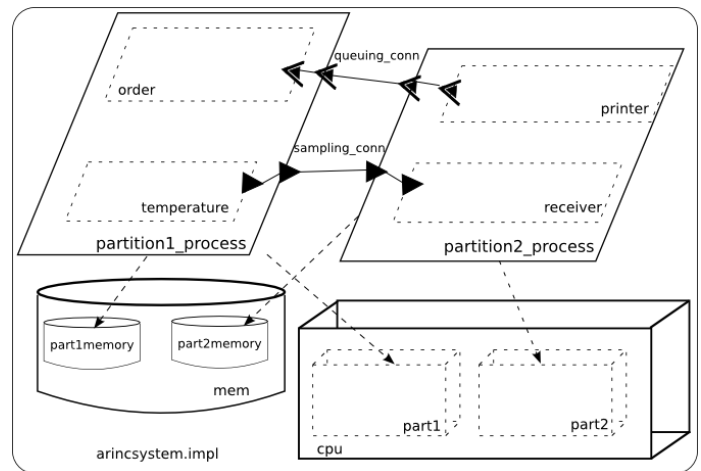
AADL associates properties to AADL model elements. Properties are typed and represent name/value pairs that represent characteristics and constraints. Examples are the period and execution time of threads, the implementation language of a subprograms, etc.

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.

## ARINC653 ARCHITECTURES VALIDATION USING REAL

Once the modeling of the entire system is done, we need to validate the model, checking that it actually represents a well-formed partitioned architecture and ensure runtime resources requirements enforcement (for example, a partition has enough memory for its threads).

To check model structure and requirements enforcement, we perform an analysis step on AADL models during our development process, as illustrated in Figure 1. This analysis is achieved using a dedicated language, REAL [9]. We propose an overview of the language and an illustration of its use for validating ARINC653 requirements.

REAL (Requirement Enforcement Analysis Language) aims at checking constraints enforcement on architectural descriptions at the specification step, saving significant time over verification at execution time. In this section, we describe the main features of this language, and then provide an example of a REAL theorem verifying security at model-level.

### LANGUAGE BASIC CONCEPTS

REAL is based on set theory. It allows one to build sets whose elements are AADL entities (connections, components or subprogram calls). Verification can then be performed on either a set or its elements by stating Boolean expressions. The basic unit of REAL is a *theorem*. A theorem verifies an expression over all the elements of a set that is called the *range set*.

In order to write complex expressions, one can use *predefined sets*, which contain the instances of the AADL model of a given type, or build intermediary sets, using *relations* between elements of sets (e.g. returns the elements of the set A which are subcomponents of any elements of the set B).

Finally, *subtheorems calls* can be used to build local or global variables, or to check pre-required constraints on the model. Callee theorems inherit at run-time from the caller environment (the *local\_set*), and the user can pass parameters. Thus, it is possible to design a library of theorems that will be used by higher-level, user-defined theorems. Such work has been done for schedulability analysis, response-time analysis and software-hardware adequacy.

## ENFORCING SECURITY RULES WITH REAL

In this section, we propose a REAL theorem to check time partitioning correctness of ARINC653 modules. In a partitioned architecture, partitions scheduling is made with a static scheduling approach repeated according to a period (the major time frame). The module (AADL processor) allocates time slots for partitions execution (AADL virtual processor). To check the major time frame and module correctness, we want to check that:

1. *Sum of partitions (AADL virtual processors) time slots is equal to the major time frame of their containing module (AADL processor)*
2. *Each module (AADL processor) contains at least one partition (AADL virtual processor).*

The following theorem checks these rules:

```
theorem module_compliance
  foreach cpu in processor_set do
    VP := {x in Virtual_Processor_Set |
           is_subcomponent_of (x, cpu)};

    check
      ((float(property (cpu, "ARINC653::Major_Frame")) =
        sum (property (cpu, "ARINC653::Time_Slots")))
        and
        (Cardinal (VP) > 0);
  end module_compliance;
```

**Figure 3 - REAL theorems to check major time frame consistency**

In this theorem, the *range set* is the predefined *processor set*, which contains all processor instances of the instantiated AADL model.

For each of those instances, we build an intermediary set called *VP* that contains all the elements of the predefined *virtual processor set* that verifies the *is\_subcomponent\_of* relation with it.

In the verification expression, we extract the AADL property "POK::Major\_Frame" (ie. processor available time) from the current *range variable* *cpu*. This value is compared to the sum of the value of each element of the "POK::Slots" AADL property list (i.e. processor time allocated to each slot). Finally, we check that the *VP set* contains at least one element.

In order to express dependency to this theorem, we can issue the following instruction in a caller theorem:

```
requires (module_compliance);
```

Then, all ARINC653 modules of the model will be automatically validated against this rule.

## AADL TO ARINC653 IMPLEMENTATION

Once system designer has specified and validated its architecture, implementation code is written. Actually, this step done manually, several developers write code according to their own interpretation of the specifications. This error-prone process is replaced by an automatic code generation process [7], which produces ARINC653-compliant C code, compiles it with an ARINC653 runtime and automatically creates applications compliant with ARINC653 requirements.

The code generation process is shown in Figure 4. First, the code generator takes AADL models as inputs and outputs ARINC653-compliant code. It configures the ARINC653 modules (kernels) and creates the code of each partition using a specific API (in our case, the ARINC653 APEX). The generated code is then compiled with application-level code and integrated with a dedicated runtime that provides low-level services. Finally, once this assembly step is performed, the process produces one or several binaries that contain all the necessary code to run the system. This is an automatic process, does not require any manual translation and thus, ensure production reliability.

In this section, we first detail our code generation strategies and the architecture of the generated code produced by our code generator, Ocarina. Then, we present our AADL/ARINC653 partitioned runtime, POK.

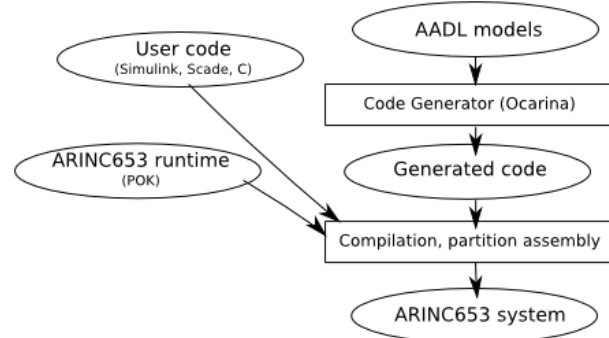


Figure 4 - Code generation process

### CODE GENERATION STRATEGIES

Our code generation process is based on AADL models and requires valid models as inputs. In particular, we assume that models contain the following information:

- Partition requirements in terms of memory (description of each address space with memory components) and resources (containment of ARINC653 processes into partitions)
- Partition communications, including inter-partition and intra-partition communication.

- Health monitoring, each layer declare handled faults and recovering strategies
- Scheduling, ARINC653 modules, partitions and processes must describe their scheduling requirements (period, execution time and so on)
- Connection requirements: in case of connected modules, we need to know which device is connected to the bus

Then, using this information, our AADL-to-ARINC653 compiler analyses the model and outputs code. It generates code for each module and their associated partitions, configuring each runtime entity (module, partitions) and ensuring requirements of the AADL model.

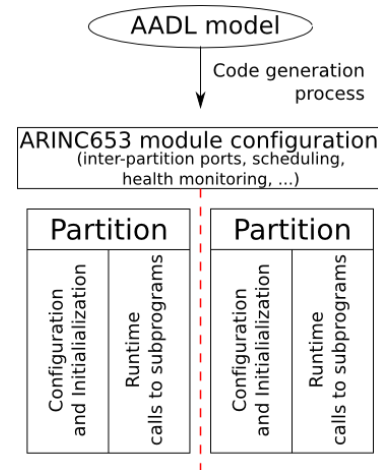


Figure 5 - Architecture of the generated code

### ARCHITECTURE OF GENERATED CODE

This code generation process generates one directory per ARINC653 module; each directory contains necessary resources for its execution. Inside each module, we generate two kind of code: **module configuration code** and **partition code**. The architecture of the generated code is shown in Figure 5.

**Configuration code** configures the ARINC653 module. It defines resources dimensioning, partitions scheduling and security policies (routing between ARINC653 sampling/queuing ports). This part of the code is especially critical since a mistake in this section can have significant impact (for example, an error in scheduling could cause a leak of time for one partition).

**Partition code** configures and defines the resources for each partition. This code is also divided into two main parts: the **initialization** and the **runtime**. The **initialization** part of the code is responsible of starting the partition. It creates inter/intra-partition communication

ports, instantiates ARINC653 processes of the partition, starts the health monitor service and puts the partition in the normal mode.

On the other hand, the **runtime** part of the code contains code executed by the threads. This code defines periodic functions executed by the ARINC653 processes and the sending/receiving on inter/intra-partition communication ports. It also calls application calls, such as Simulink or Scade models.

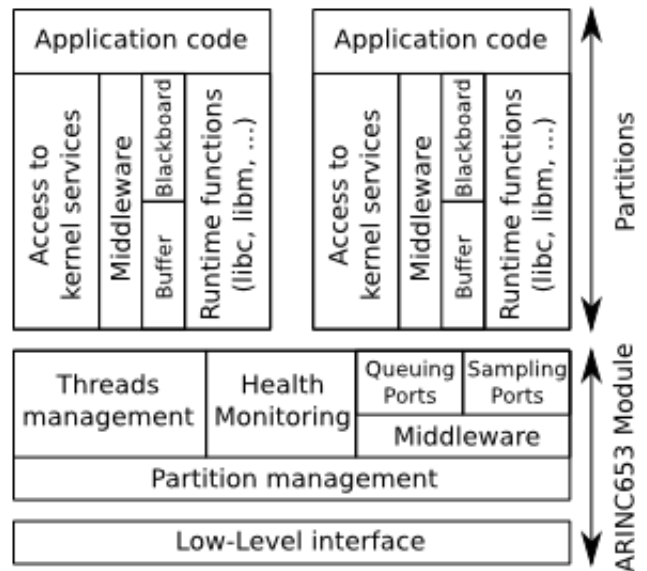
### ARINC653/AADL RUNTIME

Then, the generated code is compiled against a runtime that provides runtime services. For this purpose, we designed an AADL/ARINC653 compliant runtime, POK<sup>1</sup>. It provides time and space partitioning, supports the APEX (Application Executive) of the ARINC653 standard and is available for several architectures (x86 and PowerPC, we plan to port it to the LEON3 architecture).

The overall architecture of POK is shown in Figure 6. Each layer (module, partition) is divided in different sections that are potentially inter-dependent.

In the module/kernel part, the partition management is responsible for partition scheduling. It is the core functionality of the runtime. The threads management service handles partitions threads and schedules them according to their timing requirements. The Health Monitoring service is responsible for fault recovering at the module and partition level. When a fault is raised, the module performs an appropriate function to recover the error. The middleware layer supports inter-partitions communication services (queuing and sampling ports of the ARINC653 APEX). The queuing ports can store different values whereas sampling ports are a shared memory space where values can be read or written without a dedicated locking policy.

In the partition part, the runtime functions provide common functionalities (such as the C standard library). The middleware layer is divided into several parts and supports the different intra-partition communication functionalities of the APEX: blackboards, buffers, events and semaphores. Finally, it also contains an interface to the module/kernel to manage privileged services (thread management and so on).



**Figure 6 - Architecture of our AADL/ARINC653 runtime, POK**

The different parts of the runtime (kernel and partitions) are automatically configured by the code derived from AADL models. It keeps only the necessary code for each service and includes only required services. For example, if the AADL model does not describe communication between partitions, the code that implements inter-partitions communications is automatically removed from the kernel. This reduces the size of the module/partition, and is prone to an easier certification. In addition, it reduces the memory overhead, keeping only necessary functionalities.

### BENEFITS OF CODE GENERATION

Our code-generation approach provides several benefits for the implementation of ARINC653 systems. At first, this automatic process avoids errors by manual code production methods. It avoids errors introduced by developers and strongly enforces requirements described by the designer during early steps of the development process.

In addition, this automatic process generates only necessary code and thus, removes useless functionalities in each layer (module and partition). Required services can be automatically deduced from the AADL models (are the communication services used? what is the scheduling policy of each partition?). In consequence, the code generator creates appropriate code constructs to remove useless services and instantiates only needed resources.

<sup>1</sup> <http://pok.gunm.org>

## EXPERIMENTS

We carried out some experiments with our development process to illustrate its efficiency for the design of ARINC653 architectures. This experiment consists in implementing the model shown in Figure 2. We explain the modeling of the system, its validation and its automatic implementation with our code generation and AADL/ARINC653 runtime.

### SYSTEM DESCRIPTION

The system contains one ARINC653 module with two partitions. They communicate through inter-partitions communication using queuing and sampling ports. Each partition is executed during 1 second and communications are flushed periodically (period = 2s). Both partitions have an address space of 140Kbytes.

Both partitions contain two ARINC653 processes (AADL `thread` components). One ARINC653 process in the first partition sends data to a process located in the other partition using sampling ports (AADL `data ports`). One ARINC653 process in the second partition sends data to a process located in the first partition using queuing ports (AADL `event data ports`). These processes are scheduled periodically.

### MODELING WITH AADL

First, we model the ARINC653 module. We add an AADL `processor` component and add one time slice of one second for each partition. We also specify the major time frame (2s) in this component.

Then, we model memory segments for the partition. We model the segments with an AADL `memory` component and specify their size with an appropriate property (`Word_Count`). Then, we model the main memory with an AADL `memory` component and add two memory segments components as subcomponents.

We also model partitions. For that, we add two AADL `process` components to model the address space and two AADL `virtual processor` components to model partitions runtime. We specify the inter-partitions communication ports as AADL features with `data` and `event data ports`. The AADL `process` components are bound to `memory` component to specify the requirements of the address spaces. The `virtual processor` components that model partitions runtime are added into the AADL `processor` component that models the ARINC653 module. With that, we model the association between partitions and the module where they run.

We model ARINC653 processes of each partition. For that, we add AADL `thread` components in each partition (AADL `process` component). We model the scheduling (period, execution time, etc.) requirements by adding appropriate properties to the `thread` components. In addition, we describe the call sequence of the `thread` by specifying executed subprograms.

Finally, we add a main AADL `system` component that instantiates hardware and software components. This system contain one processor, several partitions and associates software with hardware components (what is the deployment strategy? what partitions are executed on which module?). This component also connects ports between the partitions, creating inter-partitions channels between the different inter-partitions ports.

The textual and graphical representation of this model is available in our AADL portal<sup>2</sup>. This is also available in the examples of the POK AADL/ARINC653 runtime.

### MEASURES

To illustrate that our Model-Based approach does not introduce a large overhead in the code, we carried out some measures on the memory footprint. To do that, we compile the code of both kernel and partitions with GCC and strip binaries. Our experiments are reproducible: all materials (models, runtime and toolchain) are available on our AADL portal<sup>2</sup>.

	Memory footprint
<b>ARINC653 module/kernel</b>	19 972 bytes
<b>Partition one</b>	10 270 bytes
<b>Partition two</b>	10 260 bytes

**Table 2 - Memory footprint of module/partitions**

Memory footprints are reported in Table 2. We can notice that these measures could be considered as small for a complete system that comprises high-level (partitioning support, real-time scheduling) and low-level (hardware management) functionalities. Most of the time, real-time operating systems have a high memory footprint due to embedded runtime (POSIX, libc ...), often more than 50 Kbytes. Here, the low memory footprint illustrates that a Model-Based process, with an appropriate language and efficient modeling patterns can drive the whole process without significant impact on implementation.

<sup>2</sup> <http://aadl.telecom-paristech.fr>

## CONCLUSION

In this article, we present an MDE-based development process for ARINC653 architectures. This process uses the AADL as a backbone language and addresses many issues actually identified in the design, validation and implementation of safety-critical systems.

In particular, the use of a single language in the whole process makes it more consistent and ensures that requirements defined by the system designer are enforced in the generated code and thus, in the final implementation.

This development process has already a full backed tool support through the Ocarina toolsuite<sup>3</sup> and the POK<sup>3</sup> runtime. We also provide a modeling framework for ARINC653 architectures, validation tools that uses REAL theorems, an AADL-to-ARINC653 code and an AADL/ARINC653 runtime.

In addition, we plan to improve this development process by integrating an automatic certification step. This certification step would analyze generated application and determine code coverage of the generated modules and partitions. Depending on the criticality level, this new step would determine if the code is covered according to certification requirements. In an avionics context, this would ease the certification of generated applications against the DO178B [8] standard.

## ACKNOWLEDGMENTS

This work has been partially funded by the ANR Flex-eWare project.

## REFERENCES

1. SAE. Architecture Analysis & Design Language v2.0 (AS5506), September 2008.
2. SAE. Architecture Analysis & Design Language, ARINC653 annex. To be published.
3. Avionics Application Software Standard Interface, Airlines Electronic Engineering, 1997
4. William Barnes. ARINC 653 and why is it important for a safety-critical RTOS, 2004
5. National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, 2002.
6. Julien Delange and Laurent Pautet and Peter H. Feiler. Validating safety and security requirements for partitioned architectures. In *Reliable Software Technologies -- Ada-Europe 2009*
7. Julien Delange, Laurent Pautet, and Fabrice Kordon. Code Generation Strategies for Partitioned Systems. In *29th IEEE Real-Time Systems Symposium (RTSS'08)*, pages 53–56, Barcelona, Spain, December 2008. IEEE Computer Society.
8. RTCA, Software considerations in airborne systems and equipment certification
9. Olivier Gilles and Jérôme Hugues. Validating requirements at model-level. *Proceedings of the 4th workshop on Model-Oriented Engineering (IDM'08)*, June 2008, Mulhouse, France.
10. Peter H. Feiler, David P. Gluch and John J. Hudak. *The Architecture Analysis and Design Language (AADL): An Introduction*. 2006.

## CONTACT

Julien Delange is a PhD student at TELECOM ParisTech and LIP6 under the direction of Laurent Pautet and Fabrice Kordon. His studies are focused on ARINC653 systems modeling and validation, code generation and partitioned runtime implementation. He is also currently designing a framework (the POK toolsuite) to implement ARINC653 systems from AADL models. This contains an AADL model validation tools, AADL code generation (Ocarina) and an ARINC653-compliant OS (the POK AADL runtime compliant with ARINC653 requirements).

Jerome Hugues is associate professor at the Networks and Computer Science department of ENST (PhD, ENST, 2005; graduate, ENST). He and his team focus on the use of architecture description language (AADL) to assist the construction, validation and configuration of Distributed, Real-Time and Embedded systems in the context of the Ocarina project. His main research interests include: distributed systems in particular middleware, real-time systems, programming languages.

Olivier Gilles is a PhD student at TELECOM ParisTech under the direction of Jérôme Hugues. He studies topics related to secure and efficient code generation from AADL, model-level optimizations and evaluation. He designed REAL, a constraint language for AADL models, which is used in the POK toolsuite, but also an optimization module for the Ocarina generator.

Laurent Pautet is full Professor at the Networks and Computer Science department of TELECOM ParisTech. His research work focuses on distributed real-time and embedded (DRE) middleware and also component-based frameworks and model-driven engineering tools to facilitate the automatic generation of application dedicated middleware. He has led the development of 3 free software technologies: GLADE (Distributed Systems Annex of GNAT/GCC), PolyORB (generic middleware) and Ocarina (AADL modeling framework).

---

<sup>3</sup> The Ocarina AADL toolsuite and the POK partitioned runtime are released under GPL or BSD license and are available for download at <http://aadl.telecom-paristech.fr>

