Validate, Simulate and Implement ARINC653 Systems using the AADL

Julien Delange
Laurent Pautet
TELECOM ParisTech
LTCI UMR5141
46, rue Barrault
75634 Paris Cedex 13, France
{delange,pautet}@enst.fr

Alain Plantec
Mickael Kerboeuf
Frank Singhoff
LISyC/University of Brest/UEB
20 av Le Gorgeu
29238 Brest Cedex 3, France
{plantec,kerboeuf,singhoff}
@univ-brest.fr

Fabrice Kordon
LIP6, Univ. P & M. Curie
4 place Jussieu
75252 Paris Cedex 05, France
fabrice.kordon@lip6.fr

ABSTRACT

Safety-critical systems are widely used in different domains and lead to an increasing complexity. Such systems rely on specific services such space and time isolation as in the ARINC653 avionics standard. Their criticality requires a carefully driven design based on an appropriate development process and dedicated tools to detect and avoid problems as early as possible.

Model Driven Engineering (MDE) approaches are now considered as valuable approach for building safety-critical systems. The Architecture Analysis and Design Language (AADL) proposes a component-based language suitable to operate MDE that fits with safety-critical systems needs.

This paper presents an approach for the modeling, verification and implementation of ARINC653 systems using AADL. It details a modeling approach exploiting the new features of AADL version 2 for the design of ARINC653 architectures. It also proposes modeling patterns to represent other safety mechanisms such as the use of Ravenscar for critical applications. This approach is fully backed by tools with Ocarina (AADL toolsuite), POK (AADL/ARINC653 runtime) and Cheddar (scheduling verification). Thus, it assists system engineers to simulate and validate non functional requirements such as scheduling or resources dimensioning.

Categories and Subject Descriptors

D.2.1 [**Software Engineering**]: Elicitation methods, Rapid System Prototyping; D.2.11 [**Software Engineering**]: Software Architectures—*Domain-specific architectures,Languages*

General Terms

Model-Based Engineering, AADL, Verification, Schedulability

Keywords

AADL, Model Based Engineering, Schedulability, ARINC653, Simulation, Code Generation, POK, Ocarina, Cheddar

SIGAda'09, November 1–5, 2009, St. Petersburg, Florida, USA.

1. INTRODUCTION

Safety-critical systems are widely used in domains like avionics, aerospace, medicine, and led to an increasing complexity at different levels. To provide their services, such systems rely on specific functionalities such as space and time partitioning of the ARINC653 [4, 9] avionics standard.

The criticality of such systems requires an appropriate development process and dedicated tools. A misconception or a failure can have significant impacts, such as loss of human life. For that reason, it is important to develop methods and tools that detect potential misconceptions or failures as early as possible in the development process. The main idea consists in validating the system, avoiding possible misconception.

Over the years, several approaches were developed to detect and avoid potential problems during the conception of safety-critical systems [15, 8]. They rely on different notations (system modeling and analysis, code certification, etc.) and focus on error/failure detection in order to detect errors as early as possible [8, 38]. However, these approaches rely on different languages and make difficult their integration in a unified development process. One solution would use one modeling language as a backbone for the whole development process.

Model-Driven Engineering (MDE) approaches are now considered as valuable for building safety-critical systems [32]. They propose a conceptual framework to capture, validate and implement systems using models. This technology puts an emphasis on models and thus offers the possibility to analyze and detect errors earlier in the software life cycle. This helps to increase reliability and robustness of safety-critical applications. This is crucial: [39] reports that at least 70% of errors are introduced during the specification process and before implementation efforts.

The Architecture Analysis and Design Language (AADL) proposes a component-based approach that fits with safety-critical systems needs. It is composed of several specialized hardware and software components that can be extended or refined to model safety-critical architectures with their requirements and properties. The use of AADL eases system analysis before implementation. It is used in several projects (such as Flex-eWare [17] or SAVI [14]) to model verify or implement safety-critical systems.

However, the semantics of AADL version 1 is not amenable to represent specific capabilities of some architectures such as the ARINC653 ones with respect to their isolation requirements. Such issues were discussed during the revision of the standard and AADL version 2 [43] now addresses them by introducing new components and a refined semantics to specify such requirements.

In this paper, we present experiments about the modeling, verification and implementation of ARINC653 systems using a MDE approach based on the AADL. The different steps of our approach are illustrated in figure 1.

We describe the modeling guidelines we elaborated in the ARINC653 annex of the AADL standard for the design and analysis of ARINC653 architectures. We also detail modeling patterns to describe safety-critical best design practices (for example, the use of Ravenscar and its constraints at a model-level). This part of our approach helps designers to create AADL models that describe ARINC653 architecture (top-ellipse in figure 1).

The Cheddar scheduling analysis tool checks scheduling requirements and resources dimensioning (such as buffer sizes). This analysis step (see left branch in figure 1) should be considered as a verification tool: it ensures specification correctness regarding system assumptions (for example: check that a buffer is never full).

Once system requirements are met, implementation code is automatically generated from AADL models with Ocarina (our AADL tool-suite written in Ada) and POK (our ARINC653/AADL runtime for C and Ada). This automatic process (see right branch in figure 1) ensures that implementation code was built according to the specification and avoids errors traditionally introduced by manual code. Then, generated code runs on top our ARINC653 compliant operating system, POK.

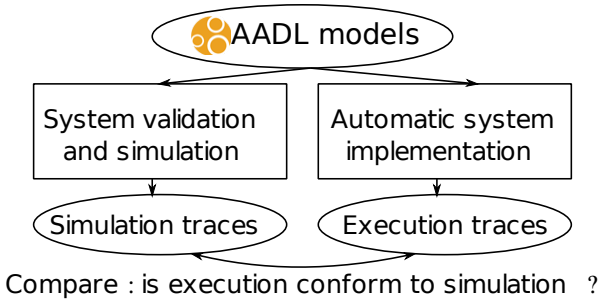

Figure 1: Proposed approach

Previous work already address scheduling analysis for ARINC653 architectures [45]. In this work, we propose modeling guidelines, analysis and verification of ARINC653 architectures and carry out both verification and implementation. In addition, the proposed approach validates resources dimensions and checks scheduling using appropriate modeling patterns.

Proposed tools use the same modeling language (AADL) and do not use different notations. By doing that, we illustrate that a MDE development process can be driven from the specifications to the implementation with respect to strong requirements as in ARINC653 architectures.

The paper is structured as follow. We first present the ARINC653 standard and the AADL. Then, we detail our modeling guidelines and patterns for the modeling of ARINC653 architectures. Section 4 discusses scheduling verification with Cheddar while the section 5 details the automatic implementation of ARINC653 systems from AADL using Ocarina and POK. Finally, a case-study (section 6) illustrates our toolchain and presents the differences between simulation and execution traces.

2. CONTEXT

This section gives an overview of the ARINC653 standard. It details services and emphasizes on scheduling concerns. Then, it introduces the AADL for the modeling of ARINC653 architectures.

2.1 ARINC653

ARINC653 [4] is an industrial standard that defines a set of services for the design of safety-critical avionics systems. The standard is focused on safety-criticality, by partitioning applications. Each partition is isolated in terms of time and space and runs as if it was executed on a single processor. Partitions are executed by a dedicated kernel/middleware: the ARINC653 module.

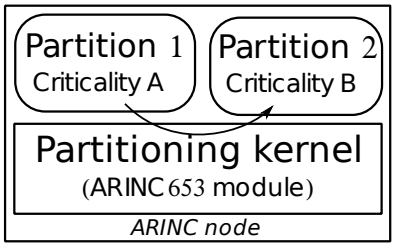

Figure 2: ARINC653 module with two partitions

The conceptual model behind ARINC653 is illustrated in figure 2. In this example, the system contains two partitions with different criticality levels, partition 1 has a higher criticality level than partition 2. A connection between the two partitions is supervised by the ARINC653 module. It ensures that data are only sent by partition 1 and only received by partition 2. The module handles both partitions time and space isolation. In consequence, it manages address spaces (to store and isolate partitions code and data) and time slots (to execute partitions).

Next subsections detail ARINC653 standard services.

2.1.1 *Partitioning services and scheduling policy*

ARINC653 isolates applications so a failure in a partition cannot affect other partitions that run on the same processor. The isolation is performed at two levels: `time` and `space`.

Time partitioning implies that each partition has a fixed time frame for its execution. In ARINC653, partitions execution is controlled by the module according to a cyclic and static scheduler.

Space partitioning means that each partition has a dedicated address space to store its code and data and that communications between partitions are supervised by the module.

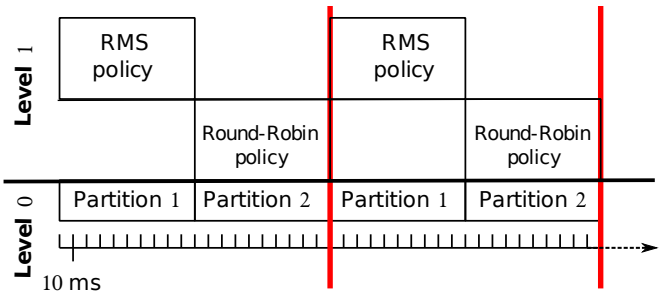

Figure 3: ARINC653 hierarchical scheduling example

ARINC653 uses a hierarchical scheduling model with two levels: *kernel* (or *module*) *level* and *partition level*. The module-level scheduler is static and executes each partition cyclically at a given rate. The partition-level is more flexible: the scheduling policy is defined by the system designer. Thus, each partition can use a different scheduling policy (static, Round-Robin, Rate Monotonic, ...) to execute their tasks.

An example of a such scheduling policy is depicted in figure 3. The first partition schedules its tasks with the RM (Rate Monotonic) scheduling protocol while the other partition uses a Round-Robin protocol. Partitions scheduling specifies that the first partition is executed for 100ms, then, the second is executed for 100ms. The partitions scheduler repeats infinitely this scheduling this pattern. Then, during their execution, partitions schedule their tasks according to their own concerns.

2.1.2 Tasking service *ARINC653 Process*

The *tasking* service proposes functions to create tasks (called *processes* in the ARINC653 standard) in partitions. Several facilities are described to express specific task requirements (period, execution time, stack size ...).

2.1.3 Intra-partition communication service

The *intra-partition* communication service proposes interfaces to enable communication between ARINC653 processes, located in the same partition. These functionalities do not use any module/kernel service and remain internal to the partition. Thus, a failure on an intra-partition communication cannot affect another partition. The standard defines four mechanisms:

1. *Buffer* stores multiple messages in message queues. Two queuing policies are proposed (FIFO, Priority).

2. *Blackboard* stores one instance of a message until it is cleared or overwritten by a new instance.

3. *Event* is a notification service to indicate the completion of a job (wait/notify concept).

4. *Semaphore* service is similar to traditional counting semaphores used to control access to shared resources.

2.1.4 Inter-partition communication service

The *inter-partition* communication service proposes functions to exchange data across partitions. They are monitored by the module and the ports routing policy (which partition is allowed to send or receive on a channel) is statically defined by the system designer. Partitions cannot bypass the routing policy and create covert channels.

The standard defines the following inter-partition communication functionalities:

1. *Queueing ports* store multiple messages in queues. This service is similar to the *buffer* service but for inter-partition communication.

2. *Sampling ports* carry successive updated messages of the same type. It is similar to the *blackboard* service for inter-partition communication.

2.1.5 Health Monitoring service

The *health-monitor* service defines mechanisms to catch potential errors during system execution.

Errors can be caught at different levels (*module/kernel*, *partition*, *process/task*), depending on their nature (scheduling, execution error, ...) and the component that generates it (module, partition or process).

For each potential error at each level, the system designer specifies an appropriate recovering policy (for example, restart or stop the faulty component) in order to keep the system stable. The system designer can also make its own recovery procedure.

2.1.6 ARINC653 systems validation needs

Despite the provided mechanisms to improve system reliability and robustness, several issues must be addressed during the development of ARINC653 systems:

- Partitions scheduling. The overall scheduling policy must be validated to check that tasks have enough time for their execution.

- Resources dimensioning. The ARINC653 standard defines services to send or receive data. However, it is necessary to check that resources dimensions are correct regarding runtime requirements. Such a validation would avoid an unexpected deadlock or crash. For example, checking correctness of *buffers* size with regards to runtime requirements ensures that no task is blocked on a full buffer at execution time.

Such a system configuration is usually achieved with a lot of tests, after implementation efforts. However, these requirements can be validated at a design-level, before any implementation work to reduce testing efforts and to detect errors early in the development process. To do so, we need to represent the system with its requirements and concerns with an appropriate semantics. For that purpose, we use the AADL, detailed in the next section.

2.2 AADL

This section details the AADL language, depicts an example and presents an existing toolset used on this case-study.

2.2.1 Overview of the AADL

AADL is a standard published by the *Society of Automotive Engineers* (SAE). It defines a component-centric language which allows the modeling of both software and hardware components. It focuses on the definition of block interfaces, and separates the implementations from these interfaces. The standard proposes both graphical and textual representation of the syntax.

An AADL description is made of *components*. The AADL standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`), execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid components (`system`).

Components describe elements of the architecture. *Subprograms* model application code. Since it is not an architectural element, it is reduced to a reference to another external piece of code. *Threads* model the active part of an application (such as POSIX threads). *Processes* model address spaces containing *threads*.

Processors model micro-processors and a minimal operating system (mainly a scheduler). *Virtual processors* model a part of the processor and could be understood in different ways : part of the physical processor, virtual machine, etc.

Memories model hard disks, RAMs. *Buses* model networks, wires. *Virtual buses* are not formally a hardware component, they are bounded to connections in order to describe their requirements. They can be used for several purposes (modeling protocol stacks, security layers, etc.) *Devices* model sensors or actuators.

Systems represent composite components that are made up of hardware components or software components or a combination of the two. For example, a *system* may represent a board with multiple processors and memory chips.

Components hierarchy of an AADL model is composed of several components and sub-components. The topmost component is an AADL system that contains processes, processors and other architecture components.

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). Each component describes their internals: sub-components, connections between these sub-components, etc.

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 allows *properties* to be associated with 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. The standard includes a predeclared set of properties and users can introduce additional properties through property definition declarations. For interested readers, an introduction to the AADL can be found in [22].

Other languages can be integrated in AADL models by means of annex libraries. These languages can be added on each component to describe other aspects. Some annex languages have been designed, such as the behavior annex [24] or the error model annex [42]. The error model annex define states of a component, its potential faults and errors and their propagation in the system.

AADL provides two major benefits for building safety-critical systems. First, compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. Second, the hybrid system components help refine the architecture as they can be detailed later on during the design process.

2.2.2 Example of an AADL model

An example of AADL model is depicted in figure 4. The corresponding textual model gives more details on the exact property and data types manipulated, and the type of subprograms to be executed. For sake of conciseness, it is not included[1].

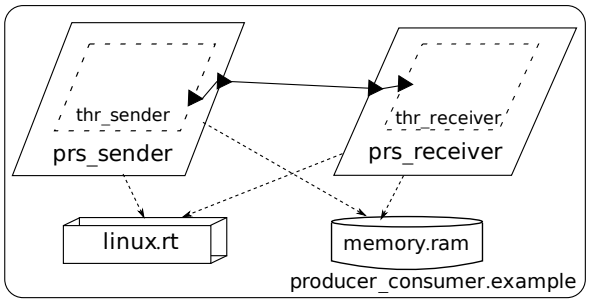

Figure 4: AADL producer/consumer

In this model, we define two processes that communicate (a basic consumer/producer example). These processes contain one threads and are executed in the same processor so they don't need network connection to exchange data. Then, each `thread` calls a subprogram that produces or consumes data.

Here, we can see the deployment specification with AADL: the bus that transport data is described, as well as processor or memory assignment. Such an additional information eases the analysis of the system and its deployment concerns.

[1]Complete textual examples can be found in Ocarina distribution

2.2.3 Existing AADL toolset

AADL has an extended toolset to model DRE (Distributed Real-Time Embedded) system, validate their architecture according their requirements and automatically implement them. At first, the Open Source AADL Tool Environment (OSATE) [1] provides a complete integration of the AADL into the Eclipse modeling framework. In addition, plug-ins provides validation facilities to check architecture correctness [23] and system requirements (such as security [26]). Other validation tools for AADL exist and provides guarantees crucial in the context of DRE systems (such as scheduling requirements [46]).

On the implementation side, architecture code can can be automatically produced from AADL models. The generated code manages system resources, enables communication across entities and enforces the requirements of each component (scheduling requirements of tasks, output rate for communication, ...). Application-level code (legacy C code or code derived from other modeling tools) is plugged on top of the generated code to be executed by generated tasks.

For that purpose, we have developed the Ocarina [55] AADL tool-suite. It contains an AADL compiler that generates Ada, RT-POSIX/C or ARINC653 compliant C code [18]. It relies on the services of a real-time executive for all concurrency and distribution features (e.g. Linux, RTEMS or VxWorks). Thanks to careful optimisations, the generated applications have low memory footprint and complexity.

3. MODELING ARINC653 ARCHITECTURES WITH THE AADL

This section presents the modeling patterns we designed for the modeling of ARINC653 [4] architectures. The presentation of our mapping follows the organization of section 2.1. Part of this work is also included in the ARINC653 annex document of the AADL, to be proposed for standardization by SAE.

3.1 Mapping partitions

An ARINC653 module (see section 2.1) is represented in AADL by means of a `processor` component. It models the underlying ARINC653 module that provides partitioning functionalities. Thus, it contains partitions runtime as sub-components and defines partitions scheduling policy as component properties.

ARINC653 partitions modeling is achieved with two AADL components: `virtual processor` and `process` components. A `virtual processor` component models the partition runtime (scheduling policy for partition tasks, partition resources, ...) whereas a `process` component models the partition address space.

The `process` component contains the partition content (`thread`, `data` and so on). The `virtual processor` component is contained in a `processor` component to model its association with its corresponding ARINC653 module.

The AADL property `Actual_Processor_Binding` associates the `virtual processor` and the `process`. We also describe memory segments allocation by associating a `process` component with a `memory` be defining the `Actual_Memory_Binding` property. This explicits the location of the partition (its memory address, available memory size, ...).

3.2 Mapping ARINC653 processes

AADL `threads` are used to model ARINC653 processes. Both represent the same concept: an instruction flow constrained by some requirements (period, deadline, execution time and so on). These requirements are specified on a AADL `thread` component

by means of the standard AADL properties. AADL `threads` are contained in AADL `process` components.

`Thread` ports describe use of *intra* or *inter* partition communications. When two connected threads belong to the same process, we assume this models an intra-partition service. When the connected threads belong to different processes, we consider this is an inter-partition communication channel.

3.3 Mapping intra-partition communication

Intra-partition communication functions of the ARINC653 standard are also represented in AADL. The proposed mapping is based on interaction mechanisms between AADL `thread` components.

Modeling of ARINC653 buffers is achieved with AADL `event data ports` connecting AADL `thread` components.

ARINC653 blackboards are mapped using AADL `data ports` connected between several AADL `thread` components. AADL `data port` do not use queuing mechanisms; thus, their semantics is equivalent to the concept of ARINC653 blackboards.

ARINC653 events service is mapped using AADL `event ports` connected between several AADL `thread` components. `Event ports` transport and queue signals without any data. It also can be shared across several `threads` by connecting them. Thus, this concept is the same as the ARINC653 events.

The ARINC653 semaphore mechanism is mapped using a shared AADL `data` component between several AADL `threads`.

3.4 Mapping inter-partition communication

The queuing port service of the ARINC653 standard is mapped using AADL `event data ports` connected across AADL `process` components. `event data ports` queue incoming data with respect to a given queuing policy. It corresponds to the concept of ARINC653 queuing ports.

The sampling port service of ARINC653 is mapped using AADL `data ports` connected across AADL `process` components. AADL `data ports` do not queue data and thus, are semantically similar to ARINC653 sampling ports. AADL properties can be specified on `port` declarations to refine their specifications (queuing policy, refresh period, ...).

3.5 Health monitoring mapping (AADL properties)

The Health Monitoring service detects faults and executes a recovering procedure when they are raised. It acts at three different levels: module, partition and process.

We map this service by describing the potential faults with their recovering strategies at each level. To describe these, we propose AADL properties (`ARINC653::HM_Errors` and `ARINC653::HM_Actions`). These properties are attached to each level of the ARINC653 architecture: *module/kernel* (AADL `processor` component), *partition* (AADL `virtual processor` component) and *process* (AADL *thread* component).

3.6 Example

An example of an ARINC653 system represented with AADL is shown in figure 5. This model defines a producer/consumer architecture (as in figure 4) compliant with ARINC653 requirements. To do so, it uses proposed modeling patterns.

In this example, we define one ARINC653 module (`arincmodule` with two partitions. Partitions are described with `processes` (partition address space, `prs_sender` and `prs_receiver`) and `virtual processor` components (partitions runtime, `part1_rt` and `part2_rt`).

We also model a memory hierarchy to model available memory segments (`memory.main`). Partition address spaces (AADL

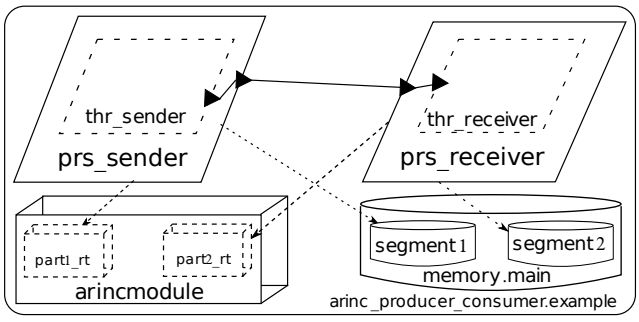

Figure 5: ARINC653 producer/consumer

`process` components) are associated with these memory segments (AADL `memory` components). Partition runtime (AADL `virtual processor` components) are also associated with partition address spaces (AADL `process` components). These associations (called *bindings*) are represented with dashed arrows (graphical version) or properties (textual version).

Each partition contains one task (AADL thread components) and we introduce an inter-partition communication channel between the partitions. The AADL ports that connect partitions are `data ports`. These ports correspond to ARINC653 sampling ports.

4. VERIFICATION OF AADL ARINC653 ARCHITECTURES

The previous section presented some modeling guidelines to describe ARINC653 architectures with AADL. We now discuss how such models can lead to automatic performance analysis. We focus on ARINC653 task timing constraints and resource dimensioning verifications. Such verifications can be performed thanks to various techniques like formal methods, queuing systems theory or real time scheduling theory. Here, we focus on real time scheduling theory.

Foundations for real-time scheduling theory came out in 1970 [36] and led to extensive researches. Real-time scheduling theory provides two verification methods:

1. Analytical methods also called feasibility tests
2. And algorithms in order to perform verifications with scheduling simulations.

Several tools implement real time scheduling: Rapid-RMA [51], Timewiz [50], MAST [27] or Cheddar [46].

Cheddar is a GPL open-source toolset composed of a graphical editor and a library of processing modules. The toolset is written in Ada95. The designer can express his real-time architecture thanks to the Cheddar editor. However, it is expected that designers perform the modeling activity with separate systems or software engineering tools (e.g. Stood [19], TOPCASED [21], IBM Rational Software Architect [37], ...). The Cheddar library implements most of current feasibility tests and classical real-time scheduling algorithms. This library also offers a domain specific language together with an interpreter and a compiler. This specific language is used for the design and the analysis of schedulers that are not implemented into the library.

In the sequel, we discuss how AADL ARINC653 models can be analyzed with the Cheddar toolset, either by feasibility tests or by scheduling simulations.

4.1 Verification by feasibility tests : the use of design patterns

Feasibility tests usually assume very simple architecture models that ease analysis.

The Liu and Layland real-time task model [36] is one of these simplified architecture models. Liu and Layland proposed to model each function of an architecture as a periodic task.

A periodic task periodically performs a given treatment and is usually defined by three parameters:

- Task period (P_i). The task period is a fixed delay between two release times of the task i. Each time the task i is released, it has to do a job.
- Task capacity (C_i). C_i is the bound of execution time of a job. For each release, the task will request C_i units of time of the processor to run the corresponding job.
- Task deadline (D_i). The task deadline is the timing constraint that the task must meet. At each release time t, the corresponding job has to be ended before time $t + D_i$.

4.1.1 Example of a feasibility test

From the simple periodic task model, many feasibility tests were proposed. As an example, Joseph and Pandia [25] have proposed a way to compute the worst case response time of a periodic task with a pre-emptive fixed priority scheduler as defined in equation 1.

$$r_i = C_i + \sum_{\forall j \in hp(i)} \left\lceil \frac{r_i}{P_j} \right\rceil \cdot C_j \quad (1)$$

where r_i is the worst case response time of the task i and $hp(i)$ is the set of tasks which have a higher priority level than i.

To check task timing constraint, this worst case response time must be compared to the task deadline.

4.1.2 Increasing feasibility test usability with design-patterns

Each feasibility test must be applied on architectures which are compliant with numerous assumptions. For example, with equation 1, we assume that all tasks are released on the same time (called critical instant), that the scheduler is pre-emptive, that tasks have no precedence relationships and that $\forall i : D_i = P_i$.

Furthermore, most of the time, this feasibility tests must be extended in order to take into account specific architecture behavior: task waiting time on data components, jitter on task release time, task precedence relationships, ...

It leads that numerous feasibility tests have been elaborated during the last 30 years. Since each feasibility test requires that the target system fulfills a set of specific assumptions, it may be difficult for a designer to choose the relevant analytical method. Unfortunately, there is currently a too limited support provided by design languages and software engineering tools to help the designer automatically applying real-time scheduling theory.

In [19], we have proposed an approach to ease the use of real time scheduling theory. This approach consists in coupling Cheddar with a modeling tool called Stood.

Coupling of modeling and analysis tools requires that both ends strictly comply with the same semantic definition of the exchanged model. Such a guaranty can be brought by use of AADL standard all along the tool-chain. Stood was chosen because it provides an extended support for AADL in addition to its compliance with the HOOD methodology [13]. Stood allows the designer to manage a complete software project by building libraries of reusable components, reversing legacy code and specifying the real-time application as well as its execution platform. Most of the modeling activities can be performed graphically and the corresponding AADL code is automatically generated by the tool.

To ease the interoperability between Stood and Cheddar, we have proposed a set of AADL design-patterns. In this context, a design-pattern is an architectural solution to a commonly concurrency occurring problem. For each pattern, we have proposed a set of feasibility tests that Cheddar is able to automatically compute. If a designer models his architecture with one of these design-patterns, he enforces compliance of his architecture model with feasibility test assumptions.

Four AADL design-patterns models usual real-time synchronization/threads-communication paradigms were proposed.

4.1.2.1 Synchronous data-flows design pattern.

With this design pattern, thread synchronization and communication is achieved with AADL data components.

Data component access is made by a clock synchronization of the threads as Meta-H [52] proposed it.

In this synchronization schema, the thread dispatch is not affected by the inter-thread communications that are expressed by pure data flows. Each thread reads its input data ports at dispatch time and writes its output data ports at completion time. Then, this design pattern does not require the use of a protocol on data components and actually, we do not take data components into account for schedulability analysis.

In this simple case, the execution platform consists in one processor running a scheduler such as Rate Monotonic [36].

4.1.2.2 Ravenscar design-pattern.

The main drawback of the previous design pattern is its lack of flexibility at run time. Each thread will always execute, read and write data at pre-defined times, even if useless. In order to introduce more flexibility, asynchronous inter-thread communications can be proposed.

An example of such a runtime environment is given by the Ravenscar profile. Ravenscar is a part of the Ada 2005 standard [48]. It is a set of Ada program restrictions usually enforced at compilation time, which guarantee that the software architecture is real-time scheduling theory compliant. Ravenscar is an Ada subset with which real-time applications are composed of a set of tasks and shared data.

Ravenscar assumes that tasks are scheduled with a fixed priority scheduler and that data components are accessed with ICPP (Inheritance Ceiling Priority Protocol)[48, 44].

In this second design pattern data component access may occur at any time.

4.1.2.3 Blackboard design pattern.

Ravenscar allows a thread to allocate/release several AADL data components. Real-time scheduling theory usually models such a shared resource as a semaphore to represent a critical section for example.

In classical operating systems, many synchronization design patterns exist such as critical sections, barriers, readers-writers, private semaphores and various producers-consumers synchronization [49].

The blackboard design pattern implements a readers-writers synchronization protocol. At a given time, only one writer can get the access to the blackboard in order to update the data component, as opposed to the readers which are allowed to read the data component simultaneously. The usual implementation of this protocol

implies that readers and writers do not perform the same semaphore access, then, it requires extra analysis.

4.1.2.4 Queued buffer design pattern.

In the blackboard design pattern, at any time, only the last written message is made available to the threads.

Some real-time execution platforms provide communication features which allow all written messages to be stored in a buffer. AADL also proposes such a feature with event data ports. The Queued buffer design pattern models such a communication.

For this design pattern, Cheddar provides some means to perform buffer dimensioning verifications.

4.1.3 Verification of an ARINC653 AADL model with feasibility tests

For a given ARINC653 architecture model, if we expect to automatically apply feasibility tests, we must find the design pattern that is compliant with ARINC653 features of the model.

This set of design patterns may naturally model some ARINC653 features:

- The scheduling of ARINC653 tasks inside a partition is compliant to the Ravenscar design pattern assumptions.
- ARINC task communications with queuing ports may be modeled with the Queued Buffer design pattern.
- The blackboard design pattern has numerous similarities with ARINC653 data and sampling ports.

4.2 Verification with exhaustive simulations

The hierarchical scheduling of ARINC653 (see figure 3) is not compliant with the task scheduling assumptions of the four design patterns described above.

Today, very few feasibility tests exist in the case of hierarchical scheduling. Thus, design patterns of section 4.1 can only be applied on ARINC653 architectures with no more than one partition on each processor.

Since building new feasibility tests is also a difficult and expensive work, when several partitions are deployed on the same processor, one may at least expect to verify performances with exhaustive scheduling simulations.

In contrary to feasibility tests, scheduling simulations can not lead to a schedulability proof. However, in the case of an ARINC653 architecture, we can expect deterministic schedulers and periodic tasks. Thus, scheduling simulations may be considered as schedulability proof if the system designer is able to compute the scheduling during the hyper-period [35]. We call an exhaustive scheduling simulation any scheduling simulation that is run during this hyper-period.

To perform exhaustive scheduling simulations on these architectures, one has to model its specific schedulers and task models.

Different languages and models were proposed for such a purpose. For example, CPN tools [54] provides simulation features based on Petri Net. Unfortunately, the use of these general purpose simulation tools usually implies that the designer models real-time scheduling low level abstractions such as task preemption. A second way is to develop ad-hoc simulation programs, but this solution implies a very low level of reusability.

The Cheddar library proposes a third way by the use of a domain specific language and a set of tools (compiler, interpreter, code generator). This domain specific language allows the designer to build models of his schedulers and task models and to automatically generate a simulation program.

In the sequel, we first give few details on the Cheddar domain specific language and then, we explain how it may be used to perform exhaustive simulations in the context of ARINC653 hierarchical scheduling.

4.2.1 A language for modeling of hierarchical real-time schedulers

Real-time schedulers are usually composed of two different functions:

1. Arithmetic and logical statements to select a task among a set of ready tasks or to compute task priorities.
2. Temporal constraints and synchronization between entities (e.g. tasks and schedulers). These synchronizations describe how entities must work all together in order to share processors.

In the Cheddar toolset, a scheduler is modeled with a set of Cheddar programs written with a domain specific language called the "Cheddar language". The Cheddar language is composed of an Ada subset for the modeling of arithmetic and logical statements and a timed automaton language for the synchronizations.

Arithmetic and logical statements operate on simulation data. Simulation data are associated to the entities composing the architecture to analyze (e.g. task release time, scheduler quantum, shared resource protocol). A Cheddar program is organized in subprograms called sections.

The language defines usual operators and statements. Schedulers can be modeled with loops, conditional tests or assignments. This domain specific language also provides statements and operators that are specific to real-time scheduling theory. For example, the *uniform*/*exponential* statements customize the way random values are generated during simulations ; the *lcm* operator computes least common multiplier of simulation data ; the *max_to_index* operator looks for the ready task which has the highest priority level, ...

The language is typed and provides usual types as integer, boolean or string. Some types related to real-time scheduling theory are also defined.

A Cheddar program may define timed automata similar to those proposed by UPPAAL [7, 10]. UPPAAL is a toolbox for the modeling and the verification of real-time systems. Timed automata are frequently used to express timing and synchronization requirements of real-time systems. There are some experiments to model and verify real-time schedulers with timed automata [6, 47, 31]. Numerous tools exist (editors, simulators and model-checkers such as UPPAAL or Esterel Studio [12]) and some standards are also based on such a formal model (e.g. UML Statecharts [20]).

A network of timed automata models timing and synchronization between schedulers and tasks. The Ada like language described above is enough to model schedulers which have fixed synchronization relationships between tasks and schedulers. By the past, we have shown that this language allows the modeling of simple schedulers like Earliest Deadline First, Rate Monotonic or Maximum Urgency First. However, hierarchical schedulers as the one proposed by ARINC653 require the modeling of complex synchronizations.

In the context of the Cheddar language, every automaton may fire a transition separately or synchronize with another automaton. Transitions may be guarded with time constraints. Delays may express time consumption at transition firing. Finally, at transition firing, automata may run Ada like subprograms in order to compute task priorities or to choose the next task to run.

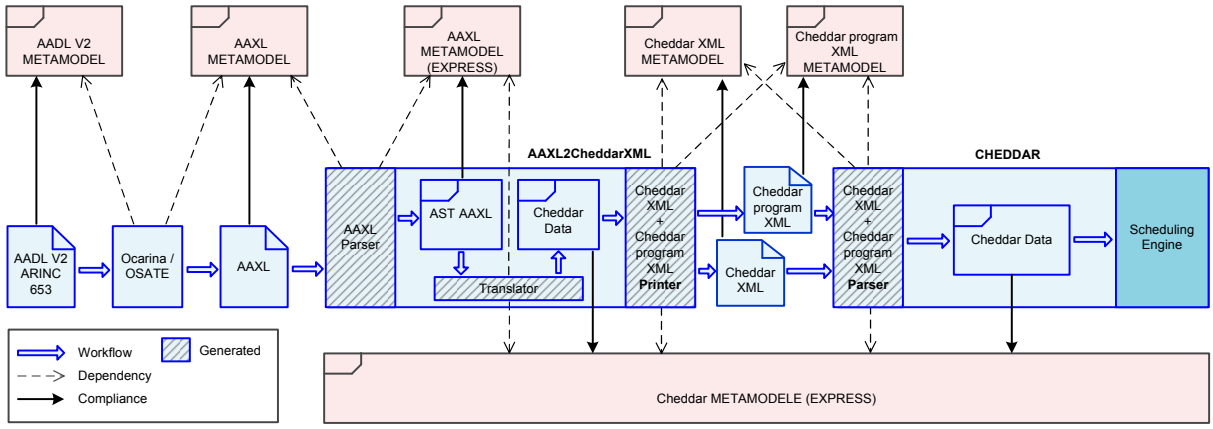

Figure 6: AADL V2 tool chain for Cheddar

4.2.2 *Performing scheduling simulations*

Figure 6 depicts the tools chain we plan to set up in order to process AADLv2 ARINC653 specifications with Cheddar. This model driven engineering process is implemented with the help of *Platypus*. *Platypus* [41] is a software engineering tool that we use to specify and implement models, meta-models and code generators handled by the Cheddar toolset. *Platypus* is using the STEP technology and especially the ISO 10303 data modelling language *EXPRESS* [29, 30]: the Cheddar data model, the Cheddar language meta model and any related code generators are all modelled using the *EXPRESS* language [46].

The scheduling analysis of an AADL ARINC653 model is a three steps process.

First step. The first step is the transformation of an AADL specification to an AAXL file using Ocarina or OSATE.

Second step. The second step is the processing of this AAXL file with a tool component called AAXL2CheddarXML. The outcome of this tool is a set of two XML files complying with two different XML formats expected by Cheddar. The first XML file is a Cheddar program modelling the actual hierarchical scheduling of the AADL ARINC653 model to be analyzed. The second outgoing XML file is a set of thread, process and processor components, expressed according to the Cheddar data model and in a syntax that Cheddar scheduling simulator engine is able to handle.

From an inside point of view, AAXL2CheddarXML is composed of an AAXL parser, a Cheddar XML printer, and a translator. The parser produces an abstract syntax tree. The translator processes this structure to create instances of the Cheddar data meta model. The printer produces the Cheddar XML files from this set of instances. The three sub-components of AAXL2CheddarXML are supposed to be generated from the underlying meta model with *Platypus*.

Third step. In this last step, the scheduling engine computes simulations with XML Cheddar files produced by AAXL2CheddarXML.

Scheduling simulation consists in predicting for each unit of time, the task to which the processor should be allocated. Checking if tasks meet their deadlines can be performed by analysis of the computed scheduling.

When an ARINC653 partition only contains periodic tasks and a deterministic schedulers, scheduling simulations lead to a proof whether tasks meet their deadline. For such a proof, the scheduling engine will perform scheduling simulation during the partition hyper-period that can be computed by [35, 16]:

$$[0, LCM(\forall i : P_i)] \quad (2)$$

where P_i is the period of the task i and LCM is the least common multiplier of all task periods of the ARINC653 partition. With equation 2, we assume that all periodic tasks of the partition have a same first release time.

5. USING VERIFIED SPECIFICATIONS FOR ARINC653 SYSTEMS IMPLEMENTATION

Previous sections detail the modeling and the verification of ARINC653 and safety-critical systems with AADL. In this section, we present the automatic implementation of ARINC653 systems using the same modeling artifacts. The traditional implementation process from AADL (designed in our previous work) is shown in figure 7. It is based on code generation functionalities of Ocarina (our AADL toolsuite written in Ada), a static AADL runtime (PoliORB-HI [17]) that provides access to the operating system and application-level code (provided by the user).

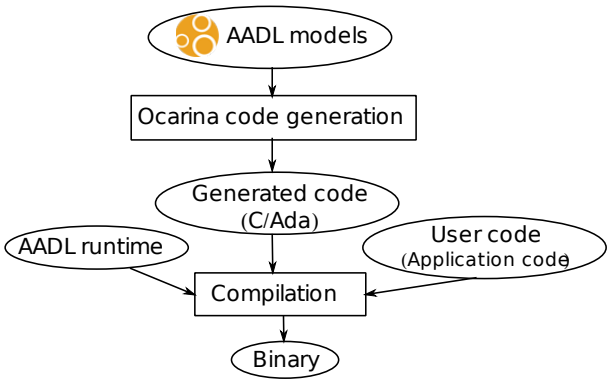

Figure 7: Ocarina development process to implement systems from AADL specifications

Ocarina automatically generates code from AADL models. This code is then compiled with the AADL runtime (left branch on figure 7) and user code (C or Ada functions, depicted by the right

branch on figure 7) to create executable binaries. The AADL runtime provides execution services specific to AADL generated code and relies on operating system services. Some services of this AADL execution platform are manually written, some other parts are automatically generated as well as glue code used to plug application components to the AADL runtime.

We improved this code generation process to be able to generate hierarchical architectures such as ARINC653. Thus, this new code generator creates code and configures each level (partition and kernel/module). We also designed a kernel and a runtime for the execution of each system layer (module/kernel and partitions).

The improved code generation process is detailed in figure 8. Our AADL-to-ARINC653 code generator (Ocarina) outputs kernel and partitions code compiled against an AADL/ARINC653 compliant operating system which provides functionnalities to the generated code. Configuration and resources of kernel and partitions are deduced from the analysis of architecture needs. Using the requirements of each layer, we generate tiny code and avoid potential overhead in resource management.

In the following, we discuss the ARINC653-toAADL code generation patterns we add to Ocarina, highlighting the differences with our previous work [17, 28].

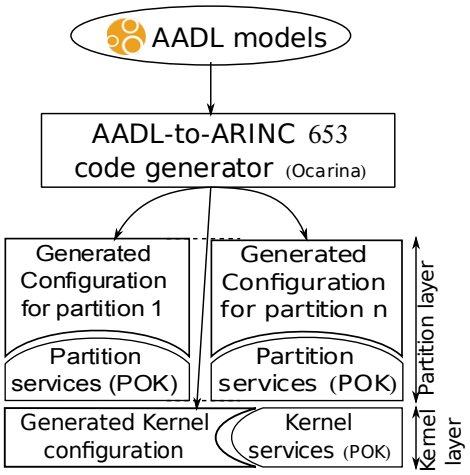

Figure 8: Detailed implementation process of ARINC653 systems from AADL models

5.1 From generic to ARINC653 code

Our past work with Ocarina were focused on code generation patterns for C [17] and Ada [28] which are optimized to reduce the overhead induced by the distribution general purpose middleware. Only the relevant services of the distribution middleware are integrated in the final executable. This generated code also enforces strong requirements, such as the Ravenscar profile [5].

Generated code usually relied on traditional operating systems which introduce a large overhead due to the integration of unused kernel services. Such a dead code may prevent the certification of safety-critical systems. In addition, the generated code used wide-known standards (as POSIX) that are not relevant in our context.

To address these issues, we adapt code generation patterns to configure the underlying kernel according to its requirements and properties. We configure each partition by including only needed services and by avoiding potential overheads. Moreover, we generate code for each layer of the hierarchical architecture in order to fulfill its requirements (such as time/space isolation).

We now describe the mapping strategies to transform AADL components into ARINC653 compliant code and detail the code generated in module/kernel and partitions layers.

5.1.1 Generating code for ARINC653 module (AADL processor)

When the model defines an AADL `processor` (which corresponds to an ARINC653 module), its contained `virtual processor` components (which correspond to ARINC653 partitions) are analyzed: the code generator deduces the required services and the ports routing policies to configure the module.

Once components are analyzed, the code generator produces a dedicated code to configure kernel/module services: partitions scheduling, memory isolation and inter-partition ports routing. This part is especially crucial since it configures the most critical part of the system (the partitioning policy).

5.1.2 Generating partition code (AADL virtual processor and process)

The code generator analyses the AADL `process` associated to each AADL `virtual processor` component (the runtime of an ARINC653 partition). During the analysis of this component, we configure the services of the partition, depending on its subcomponents and properties.

For each AADL `process` (which corresponds to the address space of the partition), incoming and outgoing ports (`event data` and `data` ports) are analyzed. Depending of their declaration, we enable or disable inter-partition communication services (ARINC653 queuing or sampling port). On the kernel configuration side, connections of these ports are analyzed in order to declare them with all their requirements (size, direction, etc.).

The code generator also configures each partition address space according to the associated `memory` component. Then, the contained AADL threads located in this AADL `process` (ARINC653 partition) are analyzed.

5.1.3 Generating processes code (AADL threads)

For each AADL `thread` component (ARINC653 process), we inspect incoming and outgoing ports. If these ports are connected to the contained `process` ports, the thread uses previously created inter-partition communication ports (ARINC653 queuing or sampling ports). If these AADL `ports` are connected to other AADL `thread` components located in the same `process`, they are mapped to ARINC653 intra-partition communication ports.

In that case, depending on its category (`event`, `data` or `event data`), a port is mapped to different intra-partition communication mechanisms (respectively ARINC653 `events`, `blackboards` or `buffers`). In addition, we set its requirements (size, direction, queuing policy, ...) according to the AADL model.

We also add function calls to create the associated ARINC653 process inside the partition according to its scheduling requirements (period, execution time and so on). Finally, we add configuration directives in the kernel/module to support the required amount of tasks.

5.1.4 Generating functions (AADL subprograms)

The code generator transforms AADL `subprograms` calls in each `thread` into a C call sequence.

For each `subprogram`, it generates a function/procedure. It enforces the AADL description (parameters, data types and so on) of the call sequence. Then, this function calls other generated functions or user-defined code (Ada or C function/procedure, potentially generated from application-level models).

5.2 From generated code to binaries

Although several commercial ARINC653 operating systems exist [33], they do not really fit with generated code needs. On the one hand, commercial operating systems have dedicated configuration directives, specific to each ARINC653 operating system provider. On the other hand, our code generator produces a fine grained configuration of the runtime according to the user requirements. The integration of the generated configuration and commercial operating system configuration is tedious due to the number of different directives and their vendor-specific aspects.

Our AADL runtime (POK) is compliant with ARINC653 requirements. It has two layers: a kernel layer that enforces time and space partitioning across partitions and a partition runtime layer that provides required functionalities and kernel-interfacing functions for each partition. The following subsection details each layer.

5.2.1 Kernel/module Layer

The module layer manages partitions and handles services that need privileged instructions. It contains only few services to remain small and potentially amenable for validation/certification.

On the contrary, unprivileged services are contained in the partition layer. This design guideline reduces the module size (critical part of the system) and puts a potential overhead in partitions (less critical part).

Our kernel layer (ARINC653 module) provides the following services:

- Partitioning support with isolation mechanisms
- Inter-partition communication ports (ARINC653 queuing and sampling ports are supported)
- Hierarchical scheduling and time isolation enforcement across partitions

The partitioning support consists in isolating partitions in distinct address spaces. It also manages partition threads and ensures their isolation in their associated partition address space. The inter-partition communication service handles ports. It also provides isolation across ports: one port belongs to a partition and cannot be used by another. Finally, the scheduling service ensures time isolation enforcement across partitions. It executes partitions according to their allocated time slots and schedules partition threads according to partitions scheduling policy.

5.2.2 Partition Runtime

The partition runtime layer contains more services than the module. Provided services are listed below:

- Intra-partition communication ports
- Standard libraries (such as the standard C-library or the math library)
- Memory allocator
- Device drivers

The intra-partition communication service is entirely handled in the partition runtime in order to avoid some useless functionalities in the kernel layer. It handles the four communication patterns of ARINC653 (buffers, blackboards, events and semaphores).

The standard libraries provides widely-used libraries to ease portability (standard C functions, math library, ...).

The memory allocator service provides functions to allocate or free memory (as in `malloc()` or `free()`). This service handles memory located in the partition address space; thus, no memory allocation is performed by the kernel layer.

Finally, device drivers are implemented in the partition runtime so that partitions can have a direct access to the hardware. In some cases, drivers require to perform privileged instructions. Then, we provide interfacing functions with the module to perform privileged instructions with respect to isolation requirements (the kernel rejects instructions that may break the partitioning policy). For instance, if a partition is allowed to perform some privileged instructions, others will not be allowed to do them.

In both parts (kernel and partition runtime), the developer can automatically include or remove some services using configuration directives. This way, useless services are not included in order to reduce the memory footprint (dead code avoidance) and ease the certification/verification. For example, if no partition uses ARINC653 queuing ports, their associated functions are not included in the kernel/module layer and their interfacing functions are not included in the partition layer.

Our partition runtime was initially designed with the C language to ease the design of low-level kernel functionalities. However, an Ada version is currently being written. This new version will take advantage of some specific features of the GNAT compiler (use of profiles, restrictions, ...) and make partitions code more reliable. Both versions (Ada and C) are compliant with the API defined in the ARINC653 standard.

POK is released under the BSD license [3] and is available on two architectures (x86 and PowerPC). It can be tested with emulators such as QEMU [11]. Although the use of AADL have many benefits, developers can also configure POK manually.

6. CASE STUDY

The following case study illustrates the use of our toolset for the verification and implementation of ARINC653 architectures. In the following, we verify and automatically implement the system from the same AADL model, as detailed in figure 1 (section 1).

The case study is composed of:

1. An AADL model that describes architecture concerns. It is used by both verification and implementation processes.
2. C application-specific code. It is only used for the implementation process (it is the code executed by partitions tasks).

The verification process is fully backed with Cheddar while the implementation process is achieved with Ocarina (code generation from AADL to ARINC653/C) and POK (AADL runtime for the generated code). Files and associated materials are available for download on our AADL portal [2].

In the following, we first present the case study, the associated AADL model and its modeling patterns. Then, we detail the verification process on this model and discuss the automatic implementation. Finally, we compare the results of both execution and simulation.

6.1 Case-study overview

Our case-study is composed of three partitions that run on the same processor: *Synchronous*, *Ravenscar* and *Queued Buffer*. The application part of the system is not described here since its perform some computations on data and is not relevant in the context of architecture analysis.

The *Synchronous* partition contains three tasks (ARINC653 processes) scheduled with a non-preemptive scheduler. Two tasks (*Thread 1* and *Thread 2* on the model) share one data (*shared data*), which is not protected by dedicated mecanisms (such as mutex or semaphore).

On the contrary, protection of the data is achieved by the non-preemptive scheduler. It allocates a fixed amount of time for each task that is sufficient to complete tasks job. The remaining job (*Thread 3*) does not use the data, it just prints some informations.

The *Ravenscar* partition has the same architecture than the *Synchronous* partition. However, it schedules its tasks with a preemptive scheduler and use protection mecanisms (an ICPP mutex) for accesses to the shared data.

The *Synchronous* partition is different from the previous. It is composed of three tasks that communicate through a queued buffer (ARINC653 Buffer). Two tasks (*Sender 1* and *Sender 2*) send data to a receiver task (*Receiver*).

6.2 The AADL model

The corresponding AADL model is depicted in figure 9. Due to a lack of space, we didn't include the textual representation (it is available on our AADL portal [2]).

Each partition has its own runtime that schedules its threads and its own memory segment to store code and data. As AADL graphical diagrams do not indicate a way to model component properties, we detail the properties through several tables.

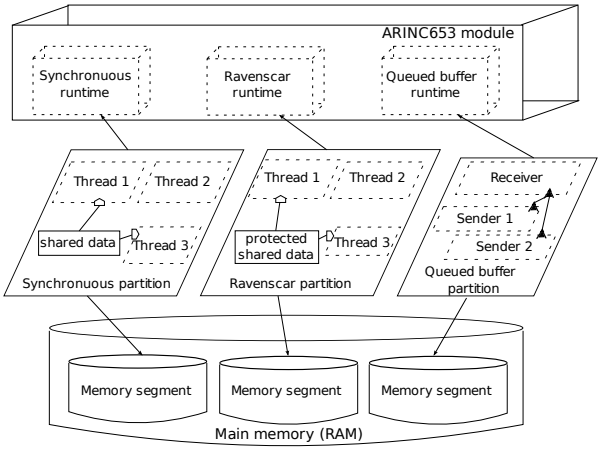

Figure 9: Case study

Scheduling requirements of each partition are reported in table 1. We can notice that the only difference between partitions *Ravenscar* and *Synchronous* is the preemptive scheduling policy, which impacts the shared data access policy.

Partition	Scheduling policy	Preemptive scheduler	Execution time
Synchronous	FIFO	no	200ms
Ravenscar	FIFO	yes	200ms
Queued-Buffer	FIFO	yes	100ms

Table 1: Partitions Requirements

In addition, we indicate the requirements of partitions tasks in tables 2 and 3. Tasks of *Synchronuous* and *Ravenscar* partitions have the same scheduling requirements. On the contrary, the shared data use the ICPP concurrency control protocol in the *Ravenscar* partition whereas the *Synchronous* partition does not use one.

6.3 Verification

Figure 10 shows a screenshot of the Cheddar tool. Cheddar displays analysis results in two parts. In the top part of the Cheddar's window, the scheduling of each AADL thread is drawn. In the bottom part of the window, performance criteria such as thread worst case response times are presented. Depending on the ADDL model to analyze, these criteria will be computed either by feasibility tests or by scheduling simulation analysis.

Task	Priority	Execution time	Period
Thread 1	4	3 ms	50 ms
Thread 2	3	2 ms	50 ms
Thread 3	1	6 ms	50 ms

Table 2: Requirements of *Ravenscar* and *Synchronous* partitions

Task	Priority	Period
Sender 1	2	20 ms
Sender 2	2	20 ms
Receiver	1	10 ms

Table 3: Requirements of the *Queued Buffer* partition

In the context of our case-study, the scheduling analysis is performed by simulation since no feasibility test exists for this kind of architecture. According to its partition scheduling, the hyper-period for this ARIN653/AADL model is 500 ms. Since the hyper-period is short enough, we can perform an exhaustive scheduling simulation : we can check that all thread deadlines will be met in the hyper-period, which means that no thread deadline will be missed at execution time.

6.4 Automatic implementation

As detailed in figure 8, ARINC653 compliant code is generated with Ocarina and compiled/linked against the POK runtime. The code generation step outputs code for kernel (ARINC653 module) and partition layers.

On kernel side, the generated code describes partitions requirements in terms of scheduling (the different time frames for each partitions) and memory. In particular, this code:

1. Allocates a memory segment for each partition. The size of the segments is specified according to the AADL model with the memory components (*Memory segment* on figure 9).
2. Schedules each partition according to their time slots (200ms for the *Synchronous* and *Ravenscar* partitions, 100ms for the *Queued-Buffer* partition).

On partitions side, the generated code initializes resources, handles communication (send/receive data from other tasks) calls application code and locks shared data when needed. In this case-study, the generated code provides the following functionnalities:

1. Tasks handling (all partitions)
2. Locking resources management (*Ravenscar* partition)
3. Communication management (*Queued-Buffer* partition)

Once the code is generated, it is automatically compiled against POK to produce binaries. POK provides an efficient toolchain to automatically generates code, compiles it with appropriate compilers and starts simulation (with emulator such as QEMU [11]). Code is compiled with a traditional compiler (*Gnu C Compiler - GCC*) and produced binaries are statically linked.

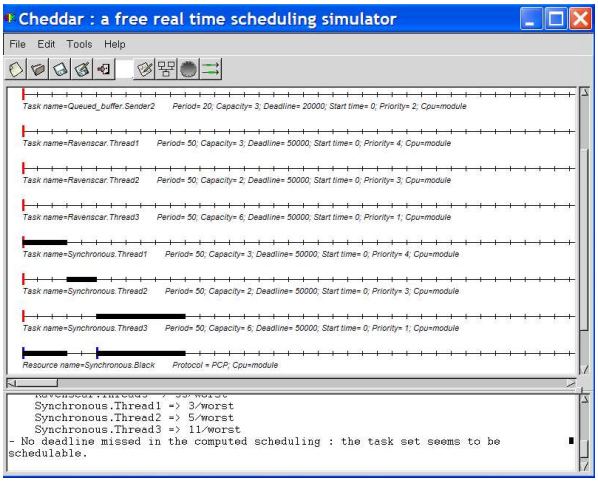

Figure 10: Scheduling simulation of the case study

6.4.1 *Execution traces*

We modify the scheduler and instrument the code to get relevant informations about partitions scheduling. Table 4 reports our execution traces and indicates tasks release time. We report only events that occur during the first 500ms of the execution. Due to a lack of place, we cannot add more information. However, it is sufficient to compare execution traces with the simulation. In the following, we discuss these results while section 6.5 compare simulation and execution results in terms of scheduling.

Time isolation across partitions

At first, the time isolation across partition is well enforced: the *synchronous* and *ravenscar* partitions are executed during 200ms while the *queued buffer* partition is executed for 100ms.

Partitions execution time is allocated according to the specifications. This timing enforcement show that:

1. The time isolation achieved by the scheduler of the POK kernel is correct.
2. The generated configuration file is correct. In other words, the translation of high level representation (AADL models) into C configuration code preserves specified requirements.

Then, we need to check that scheduling inside the partition is correct regarding the specification.

Initialization matters

In each partition, there is always a latency time between the expected release time and what is really done by the system. For example, in every partition, the first task starts one millisecond after the release of the partition.

This *"latency time"* is due to an initialization task executed when a partition starts. This initialization task creates other tasks, shared resources and communication channels. It consumes few time, but, in every case, it consumes at least 1 ms. However, once it created partitions resources, this task is no longer scheduled.

6.4.2 *Memory footprint*

One particular aspect in safety-critical systems is their constraints in terms of memory. For these reasons, generated code must be as small as possible.

People usually care about the overhead introduced by the generated code [53, 34]. Former tools and approaches introduced an overhead in the generated code that makes difficult the use of such technology in the safety-critical domain.

Partition	Task	Release time
Synchronous	Thread 1	1, 50, 100,
	Thread 2	1, 51, 100, 150
	Thread 3	2, 52,
Ravenscar	Thread 1	201 ,251, 301, 351
	Thread 2	201, 252, 301, 351
	Thread 3	202, 253, 302, 352
Queued Buffer	Sender 1	401, 421, 441, 461, 481
	Sender 2	402, 421, 441, 461, 481
	Receiver	411, 422, 431, 442, 451, 462, 471, 482, 491

Table 4: Release time of each task

We present the size of the generated application to show that a code generation process could fit with the contraints of safety-critical applications.

Table 5 presents the memory size of the generated kernel and partitions. Sizes of the *Synchronous* and *Ravenscar* partitions are similar due to their similar content. The difference between these partitions (use of locking functions) does not have a significant impact on memory requirements. The *Queued-Buffer* partition is more important because it uses more functionnalities than the other partitions. For this reason, additional code is added in this partition.

The total size of the system (kernel + partitions) is under 50 kB. It includes partitions functionnalities and a full OS support. It can be considered as *small*: typical RTOS (such as RTEMS [40]) have a memory footprint higher than 100 kB.

This experiment also shows that code generation approaches can create code that meets safety-critical systems requirements. A deeper analysis of the generated code would show the compliance of the generated code regarding constraints of safety-critical system (such as code coverage). However, this topic is beyond the scope of this article.

Component	Size
Kernel	14 kB
Synchronous Partition	11 kB
Ravenscar Partition	11 kB
Queued-Buffer Partition	13 kB

Table 5: Memory size of generated kernel and partitions

6.5 Simulation vs. Execution

Simulation and execution of this case study were discussed in sections 6.3 and 6.4. If we compare them, we can notice some differences in the release time of the tasks.

These differences are due to:

1. The execution time of the application code in the implementation. The simulation schedules tasks according to a constant task execution time which models the task worst case execution time (WCET). In contrary, the implementation code runs a real C sub-program with an execution time that is bounded by the simulation constant task execution time.
2. The *"latency time"* introduced by the initialization tasks. This issue was discussed in section 6.4.1.

However, regarding these concerns, simulation and execution show correctness of partition isolation in terms of time since:

1. Scheduling of partitions is **exactly** the same between simulation and execution.
2. Execution order of the tasks is the same between simulation and execution.

It demonstrates that:

- Simulation and implementation of ARINC653 systems can be driven from the same modeling artifact.
- Scheduling can be simulated from an architecture model and derived implementation code can reflect simulation results.
- It also shows that isolation can be verified at model level and enforced in the implementation of the system.

7. CONCLUSION

Design, verification and implementation of ARINC653 systems are very complex tasks due to their criticality and their dedicated services (time and space isolation).

The goal of this paper is to propose an appropriate MDE-based development process to capture architectures requirements using the AADL modeling language and its ARINC653 annex. The AADL was selected as a backbone language for our development process because its semantics is suitable for the modeling of safety-critical architectures. Moreover, the use of a single representation of the system all over the development process helps to relate requirements to implementation solutions.

The presented development approach exploits the user AADL models to: *(i)* check scheduling and dimensioning aspects with regards to runtime requirements using Cheddar and *(ii)* automatically generate the system, using an AADL-to-ARINC653 code generator (Ocarina) and a dedicated AADL/ARINC653 runtime (POK).

The main advantage of this development approach is the use of verification mechanisms to automatically detect potential problems before implementation efforts. Code generators also ensure that the produced code is compliant with the specification. A last advantage is to provide an ARINC653 compliant runtime to operate the produced system. These different points tend to reduce development costs and to increase reliability of safety-critical systems.

We also show that a MDE approach can produce implementation code that reflects the results of simulation/validation tools. Here, we carried out experiments on a highly-critical aspect of the system: its scheduling policy.

Acknowledgments

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

Cheddar is an open-source toolset and many people have helped the Cheddar team. We would like to thank all contributors of Cheddar (see http://beru.univ-brest.fr/~singhoff/cheddar/). We also would like to thank Ellidiss Technologie who supports Cheddar and contributes to the modeling of AADL design-patterns.

POK is an open-source AADL/ARINC653 kernel that involves many people from different schools and companies. We would like to thank every contributor of POK. A complete list of the contributors is available in the documentation of POK.

8. REFERENCES

[1] Open source AADL tool environment. Technical report, Software Engineering Institute - Carnegie Mellon University, 2006.

[2] AADL Portal at Telecom ParisTech. http://aadl.telecom-paristech.fr/, 2009.

[3] POK Website. http://pok.gunnm.org/, 2009.

[4] Airlines Electronic Engineering. Avionics Application Software Standard Interface. Technical report, Aeronautical Radio, INC, 1997.

[5] Alan Burns, Brian Dobbing, and Tullio Vardenega. Guide for the use of the Ada Ravenscar Profile in high integrity systems. 2003.

[6] K. Altisen, G. Gossler, and J. Sifakis. Scheduler Modeling Based on the Controller Synthesis Paradigm. *Real Time Systems journal*, 23(1):55–84, 2002.

[7] R. Alur and D. L. Dill. Automata for modeling real time systems. Proc. of Int. Colloquium on Algorithms, Languages and Programming, Vol 443, LNCS, pages 322–335, 1990.

[8] B. Atchison and P. Lindsay. Safety validation of embedded control software using z animation. *High Assurance Systems Engineering, 2000, Fifth IEEE International Symposim on. HASE 2000*, pages 228–237, 2000.

[9] W. Barnes. ARINC 653 and why is it important for a safety-critical RTOS. April 2004.

[10] G. Behrmann, A. David, and K. G. Larsen. A Tutorial on UPPAAL. Technical Report, Department of Computer Science, Aalbord University, Denmark, 2004.

[11] F. Bellard. Qemu, a fast and portable dynamic translator. In *ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[12] G. Berry. Getting Started with Esterel Studio 5.3. Technical report, Esterel technologies SA. Available from http://www.esterel-technologies.com/technology/getting-started/, Apr. 2005.

[13] A. Burns and A. Wellings. HRT-HOOD: A Design Method for Hard Real-time Systems. *Real Time Systems journal*, 6(1):73–114, 1994.

[14] J. Chilenski. Aerospace Vehicle Systems Institute Systems and Software Integration Verification Overview. *AADL Safety and Security Modeling Meeting*, Nov. 2007.

[15] P. Conmy, M. Nicholson, and J. McDermid. Safety assurance contracts for integrated modular avionics. In *SCS '03: Proceedings of the 8th Australian workshop on Safety critical systems and software*, pages 69–78, Darlinghurst, Australia, Australia, 2003. Australian Computer Society, Inc.

[16] F. Cottet, J. Delacroix, C. Kaiser, and Z. Mammeri. *Scheduling in Real Time Systems*. John Wiley and Sons Ltd editors, 2002.

[17] J. Delange, J. Hugues, L. Pautet, and B. Zalila. Code generation strategies from aadl architectural descriptions targeting the high integrity domain. In *4th European Congress ERTS*, Toulouse, 2008.

[18] J. Delange, L. Pautet, and F. 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.

[19] P. Dissaux and F. Singhoff. Stood and Cheddar : AADL as a Pivot Language for Analysing Performances of Real Time Architectures. Proceedings of the European Real Time System conference. Toulouse, France, Jan. 2008.

[20] D. Drusinsky. *Modeling and Verification using UML StateCharts*. Elsevier inc. editor, 2006.

[21] P. Farail, P. Gaufillet, A. Canals, C. L. Camus, D. Sciamma, P. Michel, X. Crégut, and M. Pantel. TOPCASED : An Open Source Development Environment for Embedded Systems. *Chapter 11, From MDD Concepts to Experiments and Illustrations, ISTE Editor*, pages 195–207, 2006.

[22] P. H. Feiler, D. P. Gluch, and J. J. Hudak. The Architecture Analysis and Design Language (AADL): An Introduction. Technical report, 02 2006.

[23] P. H. Feiler and J. Hansson. Flow Latency Analysis with the Architecture Analysis and Design Language (AADL). Technical report, Software Engineering Institute, 2007.

[24] R. Frana, J.-P. Bodeveix, M. Filali, and J.-F. Rolland. The AADL behaviour annex – experiments and roadmap. *Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on*, pages 377–382, July 2007.

[25] L. George, N. Rivierre, and M. Spuri. Preemptive and Non-Preemptive Real-time Uni-processor Scheduling. INRIA Technical report number 2966, 1996.

[26] J. Hansson and A. Greenhouse. Modeling and Validating Security and Confidentiality in System Architectures. Technical report, CMU/SEI, 2008.

[27] M. G. Harbour, J. G. Garca, J. P. Gutirrez, and J. D. Moyano. MAST: Modeling and Analysis Suite for Real Time Applications. pages 125–134. Proc. of the 13th Euromicro Conference on Real-Time Systems, Delft, The Netherlands,, 2001.

[28] J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. *ACM Transactions in Embedded Computing Systems (TECS)*, 7(4):1–25, jul 2008.

[29] ISO 10303-1. *Part 1: Overview and fundamental principles*, 1994.

[30] ISO 10303-11. *Part 11: edition 2, EXPRESS Language Reference Manual*, 2004.

[31] T. K. Iversen, K. J. Kristoffersen, K. G. Larsen, R. G. Madsen, M. Laursen, S. K. Mortensen, P. Pettersson, and C. B. Thomasen. Model-Checking Real Time Control Programs : Verifying LEGO Mindstorm Systems Using UPPAAL. Technical report, BRICS RS-99-53, Dec. 1999.

[32] R. N. Kashi and M. Amarnathan. Perspectives on the use of model based development approach for safety critical avionics software development. In *International Conference on Aerospace Science and Technology*, 2008.

[33] L. Kinnan, J. Wlad, and P. Rogers. Porting applications to an ARINC 653 compliant IMA platform using Vxworks as an example. *Digital Avionics Systems Conference, 2004. DASC 04. The 23rd*, 2:10.B.1–10.1–8 Vol.2, 24-28 Oct. 2004.

[34] M. Leone and P. Lee. A Declarative Approach to Run-Time Code Generation. In *In Workshop on Compiler Support for System Software (WCSSS)*, pages 8–17, 1996.

[35] J. Leung and M. Merril. A note on preemptive scheduling of periodic real time tasks. *Information processing Letters*, 3(11):115–118, 1980.

[36] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environnment. *Journal of the Association for Computing Machinery*, 20(1):46–61, Jan. 1973.

[37] E. Maes. Validation de systèmes temps-réel et embarqué à partir d'un modèle MARTE. Thales RT, Journée Ada-France 2007, Brest, 2007.

[38] J. McDermid. Software hazard and safety analysis, 01 2004.

[39] National Institute of Standards and Technology (NIST). The Economic Impacts of Inadequate Infrastructure for Software Testing. Technical report, 2002.

[40] OARCorp. Rtems - http://www.rtems.com.

[41] A. Plantec and V. Ribaud. EUGENE: a STEP-based framework to build Application Generators. *AWCSET'98, CSIRO-Macquarie University*, 1998.

[42] A.-E. Rugina, P. H. Feiler, K. Kanoun, and M. Kaaniche. Software dependability modeling using an industry-standard architecture description language. In *Proceedings of 4th European Congress ERTS, Toulouse*, Jan 2008.

[43] SAE. *Architecture Analysis & Design Language v2.0 (AS5506)*, September 2008.

[44] L. Sha, R. Rajkumar, and J. Lehoczky. Priority Inheritance Protocols : An Approach to real-time Synchronization. *IEEE Transactions on computers*, 39(9):1175–1185, 1990.

[45] F. Singhoff and A. Plantec. AADL modeling and analysis of hierarchical schedulers. In *SIGAda '07: Proceedings of the 2007 ACM international conference on SIGAda annual international conference*, pages 41–50, New York, NY, USA, 2007. ACM.

[46] F. Singhoff, A. Plantec, and P. Dissaux. Can we increase the usability of real time scheduling theory ? The Cheddar project. pages 240–253. 13th International Conference on Reliable Software Technologies, Ada-Europe, Lecture Notes on Computer Sciences, Springer-Verlag editor, volume 5026, Venice, June 2008.

[47] V. Subraminian, C. Gill, C. Sanchez, and H. B. Sipma. Reusable Models for Timing and Liveness Analysis of Middleware for Distributed Real-Time and Embedded systems. Proceedings of the 6th ACM and IEEE International conference on Embedded software EMSOFT '06, Oct. 2006.

[48] S. T. Taft, R. A. Duff, R. L. Brukardt, E. Ploedereder, and P. Leroy. *Ada 2005 Reference Manual. Language and Standard Libraries. International Standard ISO/IEC 8652/1995(E) with Technical Corrigendum 1 and Amendment 1*. LNCS Springer Verlag, number XXII, volume 4348., 2006.

[49] A. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, 2001.

[50] TimeSys. *Using TimeWiz to Understand System Timing before you Build or Buy*. White paper, http://www.timesys.com/index.cfm?bdy=home_bdy_library.cfm, 2002.

[51] Tri-Pacific. *Rapid-RMA : The Art of Modeling Real-Time Systems*. http://www.tripac.com/html/prod-fact-rrm.html, 2003.

[52] S. Vestal. Metah user's manual, version 1.27. Technical report, 1998.

[53] D. Weil, V. Bertin, E. Closse, M. Poize, P. Venier, and J. Pulou. Efficient compilation of esterel for real-time embedded systems. In *CASES '00: Proceedings of the 2000 international conference on Compilers, architecture, and synthesis for embedded systems*, pages 2–8, New York, NY, USA, 2000. ACM.

[54] L. Wells. Performance Analysis using CPN Tools. ACM International Conference Proceeding Series; Vol. 180, Proceedings of the 1st international conference on Performance evaluation methodolgies and tools., 2006.

[55] B. Zalila, J. Hugues, and L. Pautet. *Ocarina user guide*. TELECOM ParisTech.