Modeling and Validation of ARINC653 architectures

Julien Delange[1], Laurent Pautet[1], Fabrice Kordon[2]

1: TELECOM ParisTech – LTCI UMR 5141, 46 rue Barrault, F-75634 Paris CEDEX 13, France
2: LIP6, Université Pierre et Marie Curie, 4 place Jussieu, 75252 Paris Cedex 05, France

Abstract: Avionics systems must be carefully designed due to their criticality since fault may lead to loss of life. These systems must be verified and certified. However, design of avionics architectures becomes more and more complex due to an increasing demand of new functionalities. It makes very difficult to analyze systems and detect potential faults that may cause damages.

This paper presents an approach to model and validate avionics systems. Architecture requirements, properties and constraints are described with the Architecture Analysis and Design Language (AADL) and its associated ARINC653 annex. Then, we apply validation rules to check system correctness and constraints enforcement. This approach provides a high-level view of the system and eases the development of avionics system by validating their requirements at a model-level, before any implementation efforts

Keywords: AADL, ARINC653, validation, model-based, Ocarina, REAL.

1. Introduction

Context Safety-critical systems have strong requirements to be enforced all over the development process. To prevent damages from occurring errors, safety-critical architectures are based on dedicated services isolating software components and enforcing safety requirements.

The ARINC653 standard addresses such issues and introduces the concept of partitioned architectures for the design of avionics software. The main purpose is to increase system reliability and dependability. To do so, ARINC653-compliant operating systems (OS) isolate software components in terms of space and time and provide fault detection/recovery mechanisms. They also provide configuration tables to associate recovery procedures with each potential fault that may occur at runtime.

Problem We identify three problems in the design of ARINC653 architectures in terms of representation and analysis.

First, it is difficult to design ARINC653 architectures due to their amount of requirements and their associated services (communication, fault management, etc.). Since the ARINC653 standard does not provide an abstract representation of the architecture, ARINC653 systems analysis and review are made by means of code analysis, which is tedious, error-prone and OS dependent.

Second, critical services of ARINC653 architectures and OS must be analyzed before implementation efforts. These services (hierarchical scheduler, fault recovering, etc.) must be automatically validated to ensure that specified requirements can be fulfilled.

Third, the partitioning strategy must be verified to check that failure in a partition cannot affect another one. This is of particular interest since ARINC653 architectures can host components having different criticality levels. Thus, a fault that occurs in a component at a given criticality can impact other components at a higher criticality level. This behavior must be detected and avoided as soon as possible in the development process.

Proposed Approach To overcome these problems, we propose to model and validate ARINC653 systems to check for safety requirements. To do so, we rely on a modeling language providing an appropriate semantics for safety-critical architectures with isolation requirements. We need a modeling language that enables automation of verification efforts.

Among currently proposed languages, the Architecture Analysis and Design Language (AADL) introduces a component-based approach to describe both hardware and software aspects of the system. It defines several components that are aggregated by engineers to model the system according to its requirements and properties.

This paper proposes an approach to model and validate ARINC653 systems with AADL. This new representation of this kind of architecture eases system analysis and validation.

We describe ARINC653 partitioned architectures with their time and space isolation concerns (hierarchical scheduling, partitions confinement in memory segments, etc.). Modeling patterns are based on the last version of the AADL (version 2). It introduces new components relevant for the modeling of ARINC653 constraints. These

patterns are being integrated in the AADL standard as an annex document (the ARINC653 annex).

We also introduce validation rules to enforce ARINC653 requirements in AADL models. These rules check for isolation correctness (memory, scheduling, and communications requirements) as well as potential impact between components having different criticality levels.

We then show how these modeling patterns and associated verification rules can help systems designers to develop safer systems. We also present the tools that support our modeling patterns and our verification rules.

Outline Section 2 presents the ARINC653 standard and the specific services and requirements of ARINC653 architectures and OS. Section 3 introduces the AADL modeling languages and describes our modeling patterns to model ARINC653 architectures. Section 4 details ARINC653 requirements validation using AADL models. It first presents our AADL-dedicated validation language, REAL, and details its use for ARINC653 systems validation. Finally, section 5 concludes and gives an overview of incoming work on this topic.

2. ARINC653

2.1. Overview of the standard

ARINC653 [1] is an industrial standard that defines a set of services for the design of safety-critical avionics systems. The main principle consists in partitioning applications according to their criticality level. A partition is isolated in space and time and executes software components as if it was running on a dedicated processor.

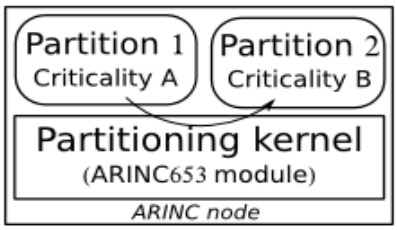

Figure 1 - Overview of an ARINC653 system

Partitions are executed on top of a dedicated kernel/middleware: the ARINC653 module. The conceptual model behind ARINC653 is illustrated in figure 1. In this example, the system contains two partitions with different criticality levels: the one of partition 1 being higher than the one of partition 2. A connection between the two partitions is supervised by the ARINC653 module to ensure that data sent by partition 1 is only received by partition 2.

The module handles both partitions time and space isolation. As a consequence, it manages address spaces (to store and isolate partitions code and data) and time slots (to execute partitions).

2.2 Time and space partitioning policies

ARINC653 isolates applications so a failure in a partition cannot affect other partitions that run on the same processor. This isolation is achieved through two partitioning policies:

1. **Time partitioning**: each partition is executed during a fixed and pre-defined time slice. The ARINC653 module schedules partitions using a cyclic algorithm repeated at a given period, called the major time frame. Typically, the value of the major time frame is equal to the sum of partitions time frames. At each major time frame, inter-partitions communication buffers are also flushed (data sent by one partition is available to its recipients).
2. **Space partitioning**: each partition owns a dedicated address space for its execution. In addition, inter-partitions communications are supervised by the module. This ensures that only allowed entities exchange data through a communication channel.

2.3 Services

The following subsections detail ARINC653 services.

2.3.1. Intra-partition communication services

Intra-partition communication services propose interfaces to enable communication between ARINC653 processes, located in the same partition. They do not use any module/kernel service and remain internal to the 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 used to control access to shared resources (e.g. counting semaphores).

2.3.2 Inter-partition communication services

Inter-partitions communication services propose functions to exchange data across partitions. They are supervised by the module, which ensure data transport. Communication policy (list of connected partitions) is statically defined by the system designer so that partitions cannot create covert channels.

Inter-partitions communications are flushed at each major time frame: data sent by a partition is

only received by its recipients during the next scheduling period. This behavior ensures communication determinism and eases buffers dimensioning.

The standard defines the following inter-partition communication functionalities:

1. **Queuing ports** store multiple messages in queues. This service behaves like the buffer service.
2. **Sampling ports** carry successive updated messages of the same type. They are similar to the blackboard.

2.3.2 Health Monitoring service

The health-monitor service defines mechanisms to catch potential errors at run-time. Errors can be caught at different levels (module/kernel, partition, process/task), depending on their nature (scheduling, execution error, etc.) and the component they are issued from (module, partition or process).

For each potential error, the system designer specifies an appropriate recovering policy (for example, restart or stop the faulty component) in order to keep the system stable. He can also provide a dedicated recovery procedure.

2.4. ARINC653 systems constraints

Due to their partitioning policy, ARINC653 systems have strong requirements that must be validated:

- Time isolation policy must guarantee that:
  1. Each partition is scheduled at least one time during each scheduling period.
  2. The value of the major time frame is consistent with partitions time frames.
- Space partitioning policy must allocate a distinct memory segment for each partition.
- Health Monitoring (HM) policy must ensure that all potential faults are bound to a recovery policy. Designers must ensure that each level of the layered architecture (module, partition, process) uses a recovering policy for each potential fault.

Such a validation is difficult to achieve hrough code review since it requires a good knowledge of the ARINC653 operating system internals. In addition, it is of special interest to analyze ARINC653 architectures at a specification-level. It helps certification engineers by finding faults that are difficult to detect, such as the impact between partitions evaluated at different criticality levels. For example, a partition at a low criticality level could impact another evaluated at a higher criticality level through a communication channel. If a fault is raised in the first partition, it could stop sending data to the other. The absence of fresh data in the highest-critical partition could lead to an application error. As a consequence, the fault raised in the lowest-critical partition is propagated to the highest one. For that reason, impacts of faults between partitions having different criticality levels must be analyzed.

3. Modelling ARINC653 architectures

3.1. Introduction to architecture modeling with AADL

AADL [2] is a standard published by the Society of Automotive Engineers (SAE). It defines a component-centric language to model both software and hardware components. It focuses on the definition of block interfaces, and separates the implementations from these interfaces.

An AADL description is made of components. The standard defines software components (`data`, `thread`, `thread group`, `subprogram`, `process`), execution platform components (`memory`, `bus`, `processor`, `device`, `virtual processor`, `virtual bus`) and hybrid component (`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 bound 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 build from hardware components, 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 a type and provides `features` (e.g. communication ports). Components communicate one with another by connecting their `features` (the `connections` section). Each component describes their internals: subcomponents, connections between these sub-components, etc.

An implementation of a thread or a subprogram component 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 components characteristics and constraints. Examples are the period and execution time of threads, the implementation language of a subprogram, etc. The standard includes a pre-declared 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 [3].

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 [11] or the error model annex [12]. It provides a convenient way to specify other aspects of the system (fault propagation, behavior, etc).

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 refining the architecture as they can be detailed later on during the design process.

3.2 ARINC653 modeling patterns

This section presents patterns we designed for the modeling of ARINC653 [1] architectures. It follows the same organization as section 2.3. This work is also included in the ARINC653 annex document of the AADL, proposed for standardization by SAE.

3.2.1 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 time and space isolation. It contains partitions runtime as subcomponents and defines isolation requirements with AADL properties.

Partitions are specified with two AADL components:

1. A virtual processor for the modeling of runtime concerns (tasks scheduling, partition resources, etc).
2. A process that describes the content of the partition (thread, data, etc).

The association between these components is defined with the Actual_Processor_Binding AADL property. The virtual processor is contained in a processor to model its containment in its related module.

Space isolation (memory segments allocation) is specified by associating the process to a memory component with the AADL property Actual_Memory_Binding. Memory components describe segment requirements (size, etc).

3.2.2 Mapping ARINC653 processes

AADL threads model ARINC653 processes because they share the same concept: an instruction flow constrained by some requirements (period, deadline, execution time and so on – described with AADL properties). ARINC653 processes are contained in a partition so that AADL threads are contained in an AADL process.

Inter and intra-partition communications are mapped in AADL by connecting components ports. When two connected threads belong to the same process, the connection models an intra-partition service. When they belong to distinct process components, it represents an inter-partition communication channel.

3.2.3 Mapping intra-partition communication

An ARINC653 buffer is represented with a connection of AADL event data ports between AADL thread components.

Modeling of ARINC653 blackboards is made with the connection of AADL data ports between several AADL threads. AADL data ports do not queue data; and thus, are semantically equivalent to the concept of ARINC653 blackboards.

ARINC653 events are described using AADL event ports between several AADL thread components. AADL event ports queue signals without any data. Thus, this concept is the same as the ARINC653 events.

The ARINC653 semaphore mechanism is represented using a shared AADL data component between several AADL threads. The concurrency characteristic of the semaphore is specified using the Concurrency_Control_Protocol property.

3.3.4 Mapping inter-partition communication

An ARINC653 queuing port is represented by connecting AADL event data ports between several AADL process components. AADL event data ports queue incoming data with respect to a given queuing policy, which corresponds to the concept of ARINC653 queuing ports.

The modeling of ARINC653 sampling port service is achieved with the connection of AADL data ports between several AADL process

components. AADL `data ports` do not queue data and thus, are semantically similar to ARINC653 sampling ports.

AADL properties are associated to `ports` to specify their characteristics (queuing policy, etc.).

3.5 Health monitoring mapping

The Health Monitoring service detects faults at different levels (module, partition, process) and executes a recovering procedure for each one. For its description with AADL, we introduce a property to represent faults (`ARINC653::HM_Errors`) and associate it with another property that models recovering procedures (`ARINC653::HM_Actions`). Both properties are associated to a component (`processor`, `virtual processor` or `thread`) that models a layer of the ARINC653 architecture (respectively module, partition or process).

3.6 Example

The modeling of an ARINC653 system with AADL is illustrated in figure 2. Two partitions (isolated in a memory segment) are executed on top of an ARINC653 module. One partition sends data to the other (an inter-partition channel).

The ARINC653 module is depicted with the AADL processor (`arincmodule`) and contains two `virtual processor` components (`part1_rt` and `part2_rt`) that represent partitions execution environment. Partitions content is specified using an AADL `process`, each partition having its own (`prs_sender` for the first partition, `prs_receiver` for the second).

An AADL component (`main`) models the organization of the memory with its segments (AADL `memory` sub-components). Partition address spaces (AADL `process` components) are then associated with them to specify the space isolation policy.

Each partition (AADL `process`) contains one task (an AADL `thread` component). We introduce an inter-partition communication channel between the partitions to model an inter-partition communication channel.

Partitions are connected using AADL data `ports`. According to our modeling patterns, this communication mechanism is an ARINC653 sampling port.

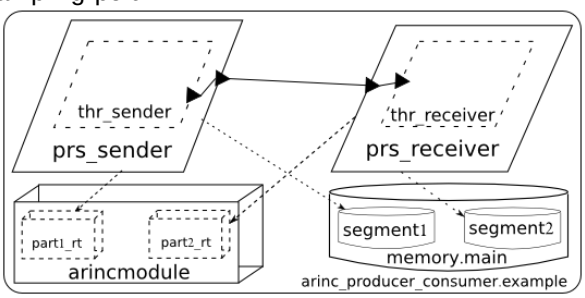

Figure 2 - ARINC653 example with AADL

4. ARINC653 architectures validation

This section shows how we use a constraint language to enforce in critical systems: time isolation, fault coverage and assessment of the all the fault-recovery strategy.

4.1. Introduction to REAL

REAL [13] (Requirements Enforcement Analysis Language) is a constraint-based language for AADL. It aims at checking constraints on architectural descriptions at the specification step, saving significant time over verification at execution time.

REAL concept is similar to formal methods such as B [14] by checking requirements on a set of elements using a dedicated language. 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.

A basic example of a REAL theorem is illustrated in listing 1. This theorem checks that all `processor` components contained in the model have at least one virtual processor. On the model in figure 2, this theorem is verified: the main processor (`arincmodule`) contains two virtual processors (`part1_rt` and `part2_rt`).

```
theorem Processor_Contains_Partitions_Runtime
  foreach cpu in Processor_Set do
    vps := {x in Virtual_Processor_Set |
        Is_Subcomponent_Of (x, cpu)};

    check (Cardinal (vps) > 0);
end Processor_Contains_Partitions_Runtime;
```

Listing 1 – REAL theorem example

Next sections present theorems to verify ARINC653 constraints (c.f section 2.4) using AADL models.

4.2. Time isolation

Time isolation is enforced by validating that:

1. Each partition contained in an ARINC653 module is executed at least one time during each scheduling period.
2. The consistency of the major time frame according to partitions time frames.

The first theorem (listing 2) checks that each `processor` component (ARINC653 module) references at least one time each contained `virtual processor` (ARINC653 partition) in its allocated time frames (AADL property `ARINC653::Slots_Allocation`). Its validation ensures that each partition is executed at least one time during each period.

```
theorem Partitions_Execution
  foreach cpu in Processor_Set do
    vps := {x in Virtual_Processor_Set |
        Is_Subcomponent_Of (x, cpu)};
    check (Is_In
      (vps,Get_Property_Value
        (cpu, "ARINC653::Partitions_Slots")));
end Processor_Contains_Partitions_Runtime;
```

Listing 2 – Theorem for partition execution enforcement

The second theorem (listing 3) checks that the major time frame of each AADL `processor` component (ARINC653 module) is equal to the sum of partitions time frames (property `ARINC653::Partitions_Slots`). This ensures that the scheduling period is consistent with partitions time frames (see section 2.2 for a description of the requirements of the major time frame).

```
theorem Scheduling_Major_Frame
 foreach cpu in Processor_Set do
   Check
    (Float
     (Property (cpu,
                "ARINC653::Module_Major_Frame"))
      =
      Sum (Property (cpu,
           "ARINC653::Partition_Slots"))));
end Scheduling_major_frame;
```

Listing 3 – Theorem for ARINC653 major time frame validation

4.3. Space isolation

To ensure space isolation, we have to verify that each memory segment is associated with a single partition. Theorem on listing 4 checks that each AADL `memory` component (a segment of the main memory) is associated with a single AADL `process` component (which contains partitions resources – data, tasks, etc.).

```
theorem Memory_Bound
foreach s in System_Set do
   mainmem :=
    {y in Memory_Set|Is_Subcomponent_Of (y, s)};
   partmem :=
    {x in Memory_Set|Is_Subcomponent_Of
    (x, mainmem)};
   partitions :=
       {x in Process_Set | Is_Bound_To (x,
partmem)};
   check (Cardinal (partitions) = 1);
end Memory_Bound;
```

Listing 4 – Theorem for the validation of space isolation

This theorem also checks model correctness, ensuring that system memory is divided into several segments. It first retrieves the main `memory` component (`mainmem`) and analyzes its `memory` sub-components (in `partmem`) that represent memory segments.

By doing so, this theorem ensures that system designer divides the main memory into several memory segments dedicated to a partition.

```
theorem check_error_coverage
 foreach thr in Thread_Set do
  Prs := {x in Process_Set |
          Is_Subcomponent_Of (thr, x)};
  VP  := {x in Virtual_Processor_Set |
          Is_Bound_To (Prs, x)};
  CPU := {x in Processor_Set |
          Is_Subcomponent_Of (VP, x)};

  var errors :=
    List ("Module_Config", "Module_Init",
          "Module_Scheduling",
          "Partition_Scheduling",
          "Partition_Config",
          "Partition_Handler", "Partition_Init",
          "Deadline_Miss", "Application_Error",
          "Numeric_Error", "Illegal_Request",
          "Stack_Overflow", "Memory_Violation",
          "Hardware_Fault", "Power_Fail");

  var actual_errors :=
    (property (CPU, "ARINC653::HM_Errors") +
     property (VP, "ARINC653::HM_Errors") +
     property (thr, "ARINC653::HM_Errors"));

  Check (Is_In (errors, actual_errors) and
         Is_In (actual_errors, errors));
end Check_Error_Coverage;
```

Listing 6 – Theorem for the validation of the fault coverage policy

4.4 Fault coverage

In ARINC653 architectures, errors may be raised at three different layers of the architecture (module, partition, process).

To check that all faults are recovered, we verify that al faults are handled during the execution of each ARINC653 process (AADL `thread` component). To do so, the associated theorem (listing 6) analyses each `thread` component (process level), its associated `process` and `virtual processor` components (partition level) and the `processor` that supports the partition (module level).

For each AADL `thread` component (that represents an ARINC653 process), the theorem computes the list (in the `actual_errors` variable) of the errors recovered by the `thread` itself but also by its associated `virtual processor` (ARINC653 partition) and `processor` (ARINC653 module). Then, it compares this list to the one of all potential errors (variable `errors`) that may be raised in the architecture.

```
theorem Check_Omission_Transient
 foreach src in process_set do
  thr     := {x in Thread_Set |
              Is_Subcomponent_Of (x, src)};
  spart   := {x in Virtual_Processor_Set |
              Is_Bound_To (src, x)};
  dst     := {x in Process_Set |
             Is_Connected_To (src, x)};
  dpart   := {x in Virtual_Processor_Set |
              Is_Bound_To (dst, x)};
  var allowed_actions :=
      List ("Partition_Restart",
            "Process_Restart",
            "Confirm");
  var src_actions :=
    (Property (spart, "ARINC653::HM_Actions") +
     Property (thr, "ARINC653::HM_Actions"));

  check
  (
   ((cardinal (src) > 0)                    and
    (cardinal (dst) >= 0)                   and
    (is_in (allowed_actions, src_actions)) and
    (max
      (property(dpart,"ARINC653::Criticality"))<
     max
      (Property
        (spart,"ARINC653::Criticality"))))
 Or
  (Not (Is_In(allowed_actions, src_actions))));
end Check_Omission_Transient;
```

Listing 7 – Theorem for the analysis of partitioning policy trade-off, transient errors

4.5 Assessment of the fault-recovery strategy

Another validation theorem checks that a recovery procedure in a partition at a low criticality level could impact another partition at a higher level. When a fault is raised in a process, the recovery policy impacts its partition. These entities (processes of the partition) stop sending or receiving data to/from the other partitions. This could be an issue if they are classified at a different criticality level.

To detect this issue, we distinguish two types of errors:

- **Transient errors** are temporary and happen when the recovering policy of the sender restarts the process or its partition. In that case, data is not sent for a temporary period. It impacts receiver components for period but once the recovering strategy is finalized, the system continues to operate as normal.
- **Permanent errors** happen when the recovery policy stops the process or its partition. Data is no longer sent, which can potentially affect recipients, especially if they are classified at a high criticality level. In that case, data will not be sent unless the task or its partition is restarted.

```
theorem Check_Omission_Permanent
 foreach src in process_set do
  thr := { in Thread_Set |
           Is_Subcomponent_Of (x, src)};
  spart := {x in Virtual_Processor_Set |
            Is_Bound_To (src, x)};
  dst := {x in Process_Set |
          Is_Connected_To (src, x)};

  dpart := {x in Virtual_Processor_Set |
            Is_Bound_To (dst, x)};
  var allowed_actions :=
     List ("Partition_Stop",
           "Process_Stop_And_Start_Another",
           "Process_Stop");
  var src_actions :=
   (Property (spart, "ARINC653::HM_Actions") +
    Property (thr, "ARINC653::HM_Actions"));

  check
 ((
   (Cardinal (Src_Prs) > 0)  and
   (Cardinal (Dst_Prs) >= 0) and
   (Is_In (allowed_actions, src_actions)) and
   (Max
    (Property (dpart,"ARINC653::Criticality")) <
    Max
    (Property (spart,"ARINC653::Criticality"))))
  Or
 (Not (Is_In (allowed_actions, src_actions))));
end Check_Omission_Permanent;
```

Listing 8 – Theorem for the analysis of partitioning policy trade-off, permanent errors

Analysis of the partitioning policy is illustrated in two theorems. The first one (listing 7) detects transient errors between partitions having different criticality levels. The second (listing 8) detects permanent errors.

They analyze each AADL `process` component (ARINC653 partitions) and its connected `process` (ARINC653 partitions that receive data from the former partition). Then, it retrieves the list of recovery

actions (in `src_actions`) that are used when a fault is raised in the source partition (`spart`).

Then, this theorem checks that:

- The receiver is classified at the lower criticality level if the fault-recovery policy of the sender generates transient data omission.
- The fault-recovery policy of the sender may not lead to transient omission.

Theorem 8 follows the same validation pattern looks for permanent errors. Compared to theorem 7, the values of the `allowed_actions` variable contain recovery actions that imply a permanent data omission.

5. Conclusion

This paper presents an approach for the modeling and validation of ARINC653 architectures.

To do so, we first introduce modeling patterns to represent ARINC653 systems and their characteristics using the AADL modeling language.

We then define, thanks to the REAL language, a set of theorems that check dedicated properties on the AADL model. This verification rules allow engineers to check for a set of predefined rules ensuring state-of-the-art correctness properties.

Altogether, these two contributions set up an automatable approach to ensure a good design of safety-critical systems with regards to safety properties. Such an approach is a particular interest for avionics systems that rely on partitioned architectures and have to fulfill strong certification requirements. It helps AADL models to be processed by certification tools to system design prior to implementation by means of code generation.

6. References

[1] Airlines Electronic Engineering Committee: "*Avionics Application Software Standard Interface*", Aeronautical Radio INC. December 2005.

[2] SAE: "*Architecture Analysis and Design Language v2.0 (AS5506)*", September 2008.

[3] Peter Feiler, David Gluch and John Hudak: "*The Architecture Analysis and Design Language (AADL) : An Introduction*", Technical Report, Software Engineering Institute, Feb 2006.

[4] Peter Feiler and Jorgen Hansson: "*Flow Latency Analysis with the Architecture Analysis Design Language (AADL)*", Technical Report, Software Engineering Institute 2007.

[5] Bechir Zalila, Jérôme Hugues and Laurent Pautet: "*Ocarina User Guide*", TELECOM ParisTech, Technical Report, 2006.

[6] Software Engineering Institute: "*Open Source AADL Tool Environment (OSATE)*", 2006.

[7] AADL Committee: "*AADL official website*", http://aadl.info.

[8] TELECOM ParisTech: "*TELECOM ParisTech official portal*", http://aadl.telecom-paristech.fr.

[9] William Barnes: "*ARINC 653 and why it's important for an safety-critical RTOS*", 2004.

[10] John McDermid: "*Software Hazard and Analysis*", Technical Report, Jan 2004.

[11] R. Frana. J-P Bodeveix, M. Filali and J.F Rolland : "*The AADL Behavior Annex – Experiments and Roadmap*", In 12th IEEE conference on Engineering Complex Computer Systems, 2007.

[12] A. Rugina, P,H, Feiler K, Kanoun and M, Kaaniche: "*Software Dependability Modeling Using an Industry-Standard Architecture Description Language*", In Proceeding of the 4th European Congress ERTS, 2008.

[13] Olivier Gilles and Jérôme Hugues: "*Validating requirements at model-level*", In Ingénierie Dirigée par les modèles (IDM'08), June, 2008.

[14] J. R. Abrial : "*The B-Book : Assigning Programs to Meanings*", Cambridge University Press, 1996.