

# Validating safety and security requirements for partitioned architectures

Julien DELANGE<sup>1</sup>, Laurent PAUTET<sup>1</sup>, and Peter FEILER<sup>2</sup>

<sup>1</sup> TELECOM ParisTech  
LTCI UMR 5141  
46, rue Barrault,  
F-75634 Paris CEDEX 13, France  
delange@enst.fr, pautet@enst.fr  
<sup>2</sup> Software Engineering Institute  
Carnegie Mellon University  
phf@sei.cmu.edu

**Abstract.** Design and validation of safety-critical systems are crucial because faults or security issues could have significant impacts (loss of life, mission failure, etc.). Each year, millions of dollars are lost due to these kinds of issues. Consequently, safety and security requirements must be enforced. Systems must be validated against these requirements to improve safety and security and to make them more reliable and robust. We present our approach to avoid such issues by modeling safe and secure systems with both safety and security requirements. We rely on a modeling language (AADL) to model and design partitioned systems with their requirements and constraints. We then validate these models to ensure security and safety enforcement. We also discuss how this approach can be used to automatically generate and build safe and secure partitioned systems.

## 1 Introduction

Every system has safety and security requirements. Unlike many domains, they are especially important in the safety-critical domain since a fault can have major impact (mission failure, loss of life, etc.). Thus, we have to ensure that security and safety requirements are enforced across the system to avoid faults and failures and to provide better reliability.

Many standards and approaches were designed to solve security and safety problems. Some of them provide a methodology to check requirements at a high level while others define low-level functionalities to enforce safety or security requirements. Despite their importance, it is still difficult to validate security and safety requirements together, since existing validation techniques work with different abstractions instead of a common conceptual framework.

Several years ago, the concept of partitioned architecture [1] has been proposed to address security and safety issues. The idea behind this concept consists in isolating pieces of software in *partitions* and make them appear independent, as if they run on different processors. By doing that, we separate system resources and isolate each partition so they cannot interfere each others.

The most known standard of partitioned architecture, ARINC653 [2], provides support for safety-critical software by defining a partitioned architecture that isolates software components in partitions. It also defines a complete set of functionality to create and manage resources inside partitions.

The MILS [3] approach classifies and separates components according to their security levels. In consequence, each security level is considered as independent. To do that, MILS proposes the concept of *separation kernel* that is close to the *partitioned kernel* defined in ARINC653 [2].

Other approaches address the problem of security and safety policies enforcement. Security policies describe allowed operation between software and hardware components regarding their security levels. Once again, all these methods do not act at the same level and do not use the same representation format which makes the verification of all requirements tedious and error prone.

As mentioned earlier the different approaches operate at different levels of abstraction and with different concepts. This makes it difficult to validate both security and safety requirements without developing separate analyzable models and then having to demonstrate consistency between the model in order to assure validity of the results. Therefore, we have chosen an approach that utilizes an extensible architecture language with a set of concepts that provides a semantically strong framework for reasoning about security and safety. By doing that, we help system designers: our approach detects errors in the system before any implementation. It is very important, because 70% of errors are introduced in systems design[4], prior to implementation efforts.

To solve these issues and to make the development of safe and secure system easier, we propose an approach to model and validate partitioned architectures with their requirements. The modeling language describes partitioned architectures with both security and safety constraints. We check the correctness of the system by validating its security and safety policies enforcement. In our approach, we use the Architecture Analysis and Design Language (AADL) [5] to model partitioned architectures with their requirements and properties. AADL is an extensible language for modeling real-time and safety-critical systems, which led us to check models through different kind of analysis like reliability analysis [6] or schedulability analysis [7].

This work is also a part of the *POK* (PolyORB Kernel) project, whose purpose is to create safe and secure systems from their specification. In this project, we rely on the AADL language because its semantics fits with the needs of safety-critical systems. It defines components and properties that ease the specification of software and hardware parts of the system with their requirements (scheduling, memory, etc.).

This paper details the modeling approach we use to describe real-time embedded architectures with their security and safety requirements. It also explains their validation, detailing our model validator, *POK Checker*.

In the remainder of this paper, we first describe different approaches and standards dedicated to security and safety in the context of safety-critical and embedded systems. Then we present the AADL [5] and detail the modeling of safe and secure system using this modeling language. Finally we describe AADL models validation against security and safety policies and present our implementation.

## 2 Security and safety approaches

This section summarizes the state of the art on security and safety in the safety-critical domain. We present ARINC653 standard, MILS and other security policies.

### 2.1 Security and safety guidelines

In the context of safety-critical systems, safety means that we want to avoid all possible occurring faults and keep the system in an error-free state. To do so, we must handle all faults and define their associated recovery procedures. Beyond that, some faults can be not handled. In that case, we want to limit their impact and avoid their propagation so they cannot affect other partitions of the system. Moreover, we separate resources across different entities of the system. We also want to prevent any unallowed usage of them. We present the ARINC653 [2] standard that defines the concept of partitioned architectures and handled faults and exceptions inside each partition.

On the security side, we want to check that only allowed entities have an access to some data according to their security clearances. Consequently, all data are transported through explicit channels and we avoid covert channels that may break the security policy. This section presents the MILS [8] approach to create independent levels of security. It also describes some well-known security policies such as Bell-Lapadula [9, 10] or Biba [11]. Other security policies could be found in [12].

### 2.2 ARINC653

ARINC653 [2] is an industrial avionics standard published by Aeronautical Radio. It defines a set of services and functionalities to create reliable systems for the avionics domain. It defines the concept of partitioned systems.

Partitioned architectures isolate software components in an entity called partition. Resources (such as processor, network interface, etc.) are isolated and separated across partitions. A dedicated kernel provides isolation in terms of:

- **Space:** each partition owns a unique address space to store its code and data. Also, it cannot access to other address spaces.
- **Time:** timeslices are allocated for each partitions to execute their threads.

Space partitioning means that partitions have separated address space and cannot read, write or execute data from other address spaces. This mechanism isolates partition memory and prevent any modification from other partitions.

Time isolation means that each partition has at least one timeslice to execute their threads and a partition cannot overrun its time budget. In ARINC653 systems, partitions are scheduled according to a static timeline protocol. It means that partitions are scheduled according to a static scheduled algorithm which is repeated at a given rate (also called the major time frame).

Partitions can communicate through legal channels supervised by the kernel. No other channel can be used for inter-partition communication. Thus, no covert channel

can be created. This functionality ensures data isolation and prevents data propagation across partitions.

Partitions are isolated and are considered as independent. Moreover, a failure inside one partition cannot affect the others. When a fault is raised, a dedicated program recovers it to keep the system in an error-free state. The fault can be caught at different levels (thread level, partition level or kernel level).

ARINC653 defines all possible faults, errors or exceptions that could happen in the system. They range from software fault (e.g: division by zero) to hardware fault (e.g: loss of power). For each possible fault, the system designer indicates at which level the fault is raised and what is its the recovering procedure. ARINC653 defines three levels of faults: kernel (called *module*), partition and thread (called *process*). For each possible fault, the system designer associates a recovering procedure.

Following these guidelines, an ARINC653-compliant system ensures that:

1. Resources are separated across partitions so a partition cannot use all resources
2. Occuring faults are recovered
3. Faults cannot be propagated outside a partition

## 2.3 MILS

MILS (which stands for **M**ultiple **I**ndependent **L**evels of **S**ecurity) is an approach defined by John Rushby [8]. It isolates software components according to their security levels. It means that a software component at a certain security level may not be collocated nor exchange data with another software component at another security level. The main idea behind this concept is to prevent the mix of different security levels.

As in ARINC653, we need to isolate software components. But unlike ARINC653, the isolation is achieved to isolate different security levels. MILS also defines a separation kernel that isolates and separates resources across partitions. In an ideal MILS system, each partition may have one security level (all software components inside a partition operate at the same security level). The separation kernel isolates partitions in terms of space and time.

MILS introduces a security classification for each component of the system. A component can be characterized as SLS, MSLS or MLS. An SLS (Single Level of Security) component handles only one security level. An MLS (Multiple Level of Security) uses different security levels and does not enforce isolation between these security levels. An MSLS (Multiple Separated Level of Security) component handles several security levels but enforce isolation between the different security levels.

Unlike ARINC653 that deals with safety, the MILS approach is focused on security and isolate security levels. Communications between partitions are also supervised by the kernel, which enforces communication isolation, prevents covert channels and ensures that different security levels are not mixed.

There is no standard that defines which functionalities reside in a MILS-compliant system and how a MILS kernel should be designed. However, we can validate the enforcement of the MILS guidelines at different level (kernel, partition, thread, ...), checking security levels isolation, components classification at an architecture level.

## 2.4 Other security policies

Several security policies exist and address several kind of enforcement. Each of them states allowed operations regarding subjects and objects security levels. Objects represent accessed data (for example, shared memory between partitions) whereas subjects are the entities that manipulate (read, write or execute) objects (the partitions that manipulates the shared memory). Subjects and objects are evaluated at a certain security level (confidential, etc.).

A security policy describes allowed operations between subjects and objects regarding their security levels. Famous security policies are Bell-Lapadula[9], Biba [11] or Chinese Wall (other security policies could be found in [12]). Each of them defines subjects, objects and allowed or denied operations (read/write/execute) according to their security levels.

## 3 Modeling safety and security using AADL

In this section, we first present the AADL. Then, we present the modeling approach we use to describe real-time embedded systems with their safety and security requirements. We also map ARINC653 and MILS concepts to AADL models. However, readers must keep in mind that other approaches dedicated to safety or security [6] exist.

### 3.1 Overview of AADL

AADL is a component-centric language which allows the modeling of both software and hardware components. It focuses on the definition of clear block interfaces, and separates the implementations from these interfaces. A graphical as well as a textual representation of the syntax exist.

Only non-functional aspects of components can be described with AADL, e.g.: security, safety or timing properties, memory footprints, interface specification and how components are interconnected.

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 well identified elements of the actual architecture. *Subprograms* model procedures as in C or Ada. *Threads* model the active part of an application (such as POSIX threads). *Processes* model address spaces that contain the *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, . . . . In our case, we use it to model the runtime accessible from a partition. *Memories* model hard disks, RAMs, *buses* model all kinds of networks, wires, *devices* model sensors, etc.. *virtual buses* are not formally a hardware component, they are bounded to connections in order to describe their requirements and can also be used for several purposes. They can model protocol stacks, security layers of a bus or other requirements of a connection.

*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 may be hierarchical, i.e.: components can contain other components (called subcomponents in this case). In fact, an AADL description is always hierarchic, with the topmost component being an AADL system that contains—for example—processes and processors, with the processes containing threads and data.

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: subcomponents, connections between these subcomponents, 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 and the bandwidth of a bus. 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 [13].

Other languages can be integrated in AADL models using 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 [14] or the error model annex [6]. The error model annex define states of a component, its potential faults and errors and their propagation in the system.

An example of AADL model is shown in figure 1. The model defines a simple system which has two processes that exchange data (one partition send a *ping* to another). We give a more detailed description later in the paper.

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. These, among other reasons, are why AADL is a good design vehicle for such systems.

### 3.2 Modeling security and safety requirements

To model systems with their security and safety requirements, we must add properties and design appropriate modeling patterns. In particular, we want to:

- Model partitioned architectures
- Specify the security levels of each software component
- Specify the security classification of each partition
- Define faults, their handling and their propagation

We dedicate a section to each modeling requirements. We also present another way to describe possible faults, their containment and propagation.

---

```

virtual processor implementation partition.one           1
properties                                             2
  Scheduling_Protocol => (RATE_MONOTONIC_PROTOCOL);     3
  POK::Recovery_Errors => (Illegal_Request);           4
  POK::Recovery_Actions => (Partition_Restart);        5
end partition.one;                                     6
                                                       7

virtual processor implementation partition.two        8
properties                                             9
  Scheduling_Protocol => (RATE_MONOTONIC_PROTOCOL);   10
  POK::Recovery_Errors => (Memory_Violation);         11
  POK::Recovery_Actions => (Partition_Restart);       12
end partition.two;                                     13
                                                       14

processor ppc                                         15
end ppc;                                              16
                                                       17

processor implementation ppc.impl                    18
subcomponents                                       19
  part_one : virtual processor partition.one;        20
  part_two : virtual processor partition.two;        21
properties                                       22
  Scheduling_Protocol => (STATIC_TIMELINE);           23
  POK::Major_Frame => 60ms;                          24
  POK::Scheduler => static;                          25
  POK::Slots => (10ms, 20ms, 30ms);                 26
  POK::Slots_Allocation => (reference (part_one),    27
                           reference (part_two),reference (part_one)); 28
  POK::Recovery_Errors => (Hardware_Fault, Power_Fail); 29
  POK::Recovery_Actions => (Ignore, Kernel_Restart); 30
end ppc.impl;                                       31
                                                       32

process send_ping_process                             33
features                                             34
  dataout : out event data port integer;           35
properties                                       36
  POK::Needed_Memory_Size => 58000 KByte;           37
end send_ping_process;                               38
                                                       39

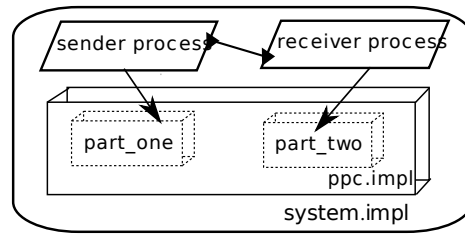
process receiver_ping_process                        40
features                                             41
  datain : in event data port integer;             42
properties                                       43
  POK::Needed_Memory_Size => 24000 KByte;           44
end receiver_ping_process;                           45
                                                       46

system implementation node.one                       47
subcomponents                                       48
  cpu : processor ppc.impl;                          49
  pr1 : process send_ping_process;                   50
  pr2 : process receiver_ping_process;              51
connections                                       52
  port pr1.dataout -> pr2.datain;                   53
properties                                       54
  actual_processor_binding => (reference (cpu.part_one)) applies to pr1; 55
  actual_processor_binding => (reference (cpu.part_two)) applies to pr2; 56
end node.one;                                       57

```

---

**Listing 1.1.** Modeling of a partitioned system using AADL



**Fig. 1.** Partitioned producer/consumer system AADL

### 3.3 Modeling partitioned architectures

In the last version of the AADL standard [5], `virtual processor` and `virtual bus` components were introduced. Their semantics are suitable to model partitioned architectures. The `virtual processor` is an abstraction for a processor. It models a runtime with its own requirements. We rely on this component to model partition runtime. In addition, the `process` component models an address space. Combination of `virtual processor` and `process` components models a partition. In this way, the `process` component models space isolation and the `virtual processor` models partition runtime.

We associate each `virtual processor` component with a `process` component to combine both space and time isolation. To do that, we bound a `process` to a `virtual processor` using a specific property (`Actual_Processor_Binding`).

The `virtual processor` components are contained in a `processor` component, which model the partitioned kernel responsible to enforce partitioning functionalities. In this component, we model the time isolation requirement using two properties: `Slots` and `Slots_Allocation`. The `Slots` property models allocated timeslices while the `Slots_Allocation` property models timeslices allocation.

We also specify the memory requirements of each partition. Each partition has an address space to store its code and data. Consequently, we have to model this address space. We can also model the memory requirements of threads inside the partition using the standard property set of the AADL.

The figure 1 depicts a partitioned system with two partitions and the listing 1.1 is the textual representation of this model. We add two `virtual processor` inside the main `processor` (lines 20-21) to model the partitioned kernel partitions runtime. `Process` components are bound to `virtual processor` (lines 55-56) to associate address spaces with partition runtimes. The connection between the ports of the `process` components (line 53) models inter-partitions communication.

Scheduling requirements are defined in the main `processor` component (lines 23-30) which contains and controls the partitions. The property `POK::Slots` defines all scheduling slots in the kernel and the `POK::Slots_Allocation` property defines the slots usage policy (which slot is used by which partition). In this example, the first partition is executed during 10 ms, then, the second partition is executed for 20 ms and the first partition is finally executed again during 30ms. Scheduling requirements of each partition is specified in the `virtual processor` component (line 3 and 10): here, each partition schedules its threads using a Rate Monotonic algorithm.

### 3.4 Modeling security levels

Modeling security requirements in AADL models offers a way to check security policies at a high-level. We annotate the model with requirements and properties dedicated to a security policy. For example, in the case of the Bell-Lapadula and Biba models, we distinguish subjects (which manipulate data) and objects (that correspond to data). In the following, we present our approach to add information on the model to be compliant with the Bell-Lapadula security policy.

The mapping of the Bell-Lapadula security policy to AADL is clear and easily understandable. For every subject (thread, process, etc.), we add a property that models their evaluated security level. On the other hand, objects are not formally identified in the model. Instead, we define security level for each port involved in a connection. Ports and connections link two subjects and manipulate objects so we must specify the security level of manipulated objects. For each connection, we add properties that describe the security level of transferred objects. We take this security level as the one used to manipulate object.

We model the security levels using the `virtual bus` component. The virtual bus models a security layer, describing its own security level. Thus, we define the property `POK::Security_Level` to specify the security level of a virtual bus. Then, we assign the security level of a partition by binding the `virtual bus` that represent it (with the `Provided_Virtual_Bus_Class` property). We also specify the security level of ports using the `Allowed_Connection_Binding_Class` property.

The listing 1.2 illustrates the modeling of security requirements. In this example, we define two `virtual bus` components, each of them models a single security level. The first `virtual bus` (`vb_secret`) models the secret security level while the other (`vb_unclassified`) models the unclassified security level. Then, these security levels are added to partitions (to specify which security levels the partition handles) and to ports. In our example, the sender partition is classified as secret (this partition is bound to the `vb_secret` virtual bus) while the receiver partition is considered as unclassified (the partition is bound to the `vb_unclassified` virtual bus).

---

```
virtual bus vb_secret 1
properties 2
  POK::Security_Level => 4; 3
end vb_secret; 4
5
virtual bus vb_unclassified 6
properties 7
  POK::Security_Level => 1; 8
end vb_unclassified; 9
10
virtual processor partition_secret 11
properties 12
  Provided_Virtual_Bus_Class => (classifier (vb_secret)); 13
end partition_secret; 14
15
virtual processor partition_unclassified 16
properties 17
  Provided_Virtual_Bus_Class => (classifier (vb_unclassified)); 18
end partition_unclassified; 19
```

---

**Listing 1.2.** Modeling security levels with AADL

### 3.5 Modeling MILS classification

We previously define the mapping of a partitioned kernel, we must also map remaining MILS concepts in AADL models. The concept of security levels has been addressed in the previous section, we now define how to map components classifications in AADL.

To do that, we define a property `MILS::Classification` that defines the components classification. The property can have the value SLS, MLS or MSLS. By doing that, the system designer can easily define the classification of its components, detailing their security levels and isolation enforcement across them.

The MILS classification are also defined in the listing 1.3. In this example, each partition is an SLS component: it handles only one security level. On the other hand, the main system that contains these partitions would be an MSLS component since it handles several security levels at the same time but enforce isolation between partitions using space isolation mechanisms. We can validate this example against the MILS rules.

```
virtual processor partition_secret 1
properties 2
  Provided_Virtual_Bus_Class => (classifier (vb_secret)); 3
  MILS::Classification => SLS; 4
end partition_secret; 5
virtual processor partition_unclassified 6
properties 7
  Provided_Virtual_Bus_Class => (classifier (vb_unclassified)); 8
  MILS::Classification => SLS; 9
end partition_unclassified; 10
end partition_unclassified; 11
```

**Listing 1.3.** Modeling partition runtime with MILS requirements

### 3.6 Modeling ARINC653 faults

As we said in the previous section, faults in ARINC653-compliant systems can be declared and handled at three different levels: kernel, partition and thread. For each component that models one of these levels, we must specify what faults are handled and their associated recovery actions.

For that, we introduce two dedicated properties:

- The `Recovery_Errors` property describes all possible faults that may be raised.
- The `Recovery_Actions` property describes available recovery procedures.

These properties are added to the `processor` (kernel level), `virtual processor` (partition level) and `thread` (thread level) components and describe which faults and handled in each level and how we recover them.

The listing 1.1 includes the definition of ARINC653-related properties. We attach to the `processor` all caught errors at kernel-level. In our example, the kernel ignores hardware faults and restarts when a power failure is detected. We use the same mechanisms for the partition level (the properties are then declared inside the `virtual processor` component) and thread level (with the `thread` component). The list of potential errors described in our property set could be extended by the system designer.

## 4 Validation of safety and security requirements

In this section, we present our validation techniques to check security and safety requirement in AADL models. In this validation process, we check that systems satisfy the ARINC653 and MILS requirements. We also verify that security policies are enforced at the architecture level.

First, we explain models validation according to partitioned architectures. Then, we check that safety requirements are enforced and all faults are handled in the system. We also present our validation techniques regarding the MILS requirements and explain how we can check other security policies using AADL models.

All these analysis techniques are made on the same architecture model. This eliminates the need to create separate models for each requirement, and led us to verify all requirements together in the same model.

### 4.1 Validate partitioned architectures

This first validation consists in verifying AADL models against partitioned architecture requirements. The AADL provides many components and it does not define a single component to model partitioned architectures. Consequently, we have to check the components aggregation:

- Each `virtual processor` is contained in a `processor component`.
- Each `process component` is bounded to a single `virtual processor`. A `virtual processor component` must be bound to exactly one `process` (1-1 relation).
- All required properties of the partitioned architecture are set and their values are correct (scheduling, memory requirements and so on)

The first verification checks that partitions (`virtual processor`) are contained to a partitioned kernel (`processor`) component. The second verification ensures that each separated address space is associated with a part of the partitioned runtime. It ensures space isolation (because the `process` models a separate address space). Then, the other verification checks others requirements. We validate scheduling requirements inspecting the timeslices allocated for partitions: we check that each partition has at least one timeslice for its execution. We also validate memory requirements of kernel and partitions according to the requirements of their subcomponents (for example, the size of a partition is correct if its threads do not need more memory than the partition memory space).

Such verifications help system designers to ensure that requirements are met and models are correct. Once it has been validated, we can make further analysis and check security and safety requirements.

### 4.2 MILS requirements validation

Then, we can also check MILS requirements. MILS defines the concept of a separation kernel, which is quite similar to the concept of partitioned kernel. However, we must check component connections and flows according to their classification level. As we explain in section 2, a component can be classified SLS, MSLS and MLS.

In the case of an SLS component, we verify that components handle only one security level. To do that, we inspect all incoming and outgoing ports and check they are at the same security level.

In the case of MLS components, we verify that incoming and outgoing ports have different security levels. If all ports are at the same security level, the MILS classification is obviously wrong and should be SLS.

The MSLS component is more complex since it handles different security levels but enforces isolation between security levels. To validate this kind of component, we inspect all flows inside the component and verify that incoming and outgoing ports are at the same security level. By doing that, we ensure security levels isolation.

### **4.3 ARINC653 fault-containment validation**

The validation of partitioned architectures was detailed previously, we now focus on fault handling. We must verify that various faults are handled at any time during system execution by thread, partition or kernel level mechanisms.

For each thread, we analyze the faults handled by itself, its contained partitions and the partitioned kernel. We consider these three levels and check that all kind of faults are handled and managed. By doing that, we can determine the coverage of faults and check if every kind of faults could be recovered for each potential runnable entity.

This techniques ensures that at any time, any potential fault would be recovered. So this sytem will be more reliable and errors would not have an impact on the system.

### **4.4 Validate security policy enforcement**

As we said in section 2, many security policies exist, each of them uses different properties to analyze and validate a system. In the remainder of this article, we consider the Bell-Lapadula and the Biba security policies to illustrate the verification of security policies in AADL models.

Bell-Lapadula and Biba security policies deal with objects and subjects. Subjects manipulate objets and each of them is evaluated at a certain security level. Then, the policy allow operations (read/write) depending on the security level of each component. Security levels are added in each component with properties. If components do not have a security level, we use the one declared in their parent component (if a thread does not have a security level, we consider the one of its containing process).

Security analysis is focused on two main verifications. At first, the compliance of connections according to the security policy. It ensures that a connection does not break the security policy. Then, the compliance of the hierarchy of components against the security policy. It detects errors prevent modeling approach that are wrong regarding the partitioned architecture requirements.

In our model, the security levels of each connection is identified by a `virtual bus` component which models security requirements of the connection. For each connection between two components, two security levels are involved: the one from the source and the one from the destination. Therefore, we can use these information to check connection legality according to a security policy requirements. Using this information,

we can state if the security policy is enforced on each connection. For example, if we check the model against the Bell-Lapadula security policy, we check that the security level of the port source does not write any data at a lower-security level and security level of destination port does not read data at a higher-security level.

However, we must consider the underlying executive in our verifications. Indeed, in partitioned architectures, each partition is executed in its own address space so all threads inside a partition share the same address space. Even if threads do not own a data at the model-level, they can read, write and update data contained in their partition at the implementation-level.

Consequently, we have to address this issue and adapt the verification of security according to runtime internals. Thread components inside the same partition are evaluated at the same security level than their containing partition (`process` or `virtual processor` component) because they can manipulate the same data.

#### 4.5 Other validation techniques

AADL supports a range of model consistency checks and analysis that help system designers develop a predictable system and validate it against requirements. There is also a validation method for scheduling requirements that can be interesting in our context. In partitioned systems, scheduling is achieved at two levels : kernel and partition level. The kernel schedules partitions using a fixed timeline algorithm while partitions use their own scheduling algorithm to schedule their threads. We can check the scheduling feasibility using analysis tools like Cheddar [7] which validate AADL models regarding their scheduling requirements.

#### 4.6 Model validator implementation and experiments

We implement these validation patterns in our model validator, *POK checker*. It validates models against our modeling approach and check that:

- Models describe a partitioned architecture
- Each kind of faults is handled at each level (kernel, partition and thread) so we ensure that occurring faults would be recovered at run time.
- Security policy is enforced at the architecture level.

At this time, our software validates AADL models against Bell-Lapadula, Biba and MILS security policies.

Use-cases and examples that demonstrate the correctness of our implementation are available in the releases of the *POK* project<sup>3</sup>. They are also detailed on our AADL portal (<http://aadl.enst.fr>). The current implementation is a command-line based tool that relies on Ocarina [15] functionalities. We are currently working on another implementation as an Eclipse plug-in to validate models in this modeling platform. It will also give the possibility to combine our validation technique with other requirements validation approaches available on this platform.

---

<sup>3</sup> Interested readers can learn more about the project on <http://pok.gunnm.org>

## 5 Conclusion and future work

In this paper, we have proposed an approach to model security and safety concerns in AADL. In doing so we utilize a common conceptual framework for safety and security analysis. It allows us to model how partitions can be used as a common runtime mechanism to support fault isolation and separation of security levels. It improves software reliability, dependability and reduces costs of development.

The result of the work discussed in this paper will be reflected in the ARINC653 annex document to the AADL standard [5]. This paper is currently in a work in progress state and will be published with the other annexes.

This work is also part of the *POK* project, whose purpose is to create safe and secure systems from AADL specification. Once requirements are met at a model level, code generation patterns are used by a code generator to produce code for a partitioned runtime system implementation. We have implemented a working prototype of such a code generator [16] in Ocarina [15] that produces Ada and C code from AADL models. We are also working on an ARINC653 and MILS compliant AADL runtime.

## References

1. John Rushby: Partitioning for avionics architectures: Requirements, mechanisms, and assurance. NASA Contractor Report CR-1999-209347, NASA Langley Research Center (1999)
2. Airlines Electronic Engineering: ARINC Specification 653 (2003)
3. Alves-Foss, J., Harrison, W.S., Oman, P., Taylor, C.: The MILS Architecture for High-Assurance Embedded Systems. International journal of embedded systems (2005)
4. National Institute of Standards and Technology (NIST): The economic impacts of inadequate infrastructure for software testing. Technical report (2002)
5. SAE: Architecture Analysis & Design Language v2.0 (AS5506). (September 2008)
6. Rugina, A.E., Feiler, P.H., Kanoun, K., Kaaniche, M.: Software dependability modeling using an industry-standard architecture description language. In: Proceedings of 4th European Congress ERTS, Toulouse. (Jan 2008)
7. Frank Singhoff, A.P.: AADL Modeling and Analysis of Hierarchical Schedulers. In: ACM SIGAda Ada Letters. (2007)
8. John Rushby: The design and verification of secure systems. In: Eighth ACM Symposium on Operating System Principles (SOSP), Asilomar, CA (December 1981)
9. Bell, D.E., LaPadula, L.J.: Secure computer system: Unified exposition and multics interpretation. Technical report, The MITRE Corporation (1976)
10. John Rushby: The Bell and La Padula Security Model. Computer Science Laboratory, SRI International, Menlo Park, CA. (1986) Draft Technical Note.
11. Biba, K.J.: Integrity considerations for secure computer systems. Technical report, MITRE
12. Kalkowski, S.: Security policies in Nizza on top of L4.sec. PhD thesis, University of Technology Dresden (2006)
13. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (aadl) : An introduction. Technical report (2006)
14. Frana, R., Bodeveix, J.P., Filali, M., Rolland, J.F.: The AADL behaviour annex – experiments and roadmap. Engineering Complex Computer Systems (2007) 377–382
15. Zalila, B., Hugues, J., Pautet, L.: Ocarina user guide. TELECOM ParisTech
16. Delange, J., Pautet, L., Kordon, F.: Code Generation Strategies for Partitioned Systems. In: 29th IEEE Real-Time Systems Symposium (RTSS'08), IEEE Computer Society (2008)