# Security Annex

## Rationale

The purpose of this annex document is to provide modeling guidelines and appropriate syntax support to capture security characteristics in an architecture model. In this section, we give the rationale of this annex, describing its capabilities and benefits.

Safety-critical embedded systems are now extremely software-reliant and operate in an untrusted, hostile environment. They are exposed to attackers that are trying to exploit, take advantage of potential defects in order to corrupt the system or stealing important (e.g. classified) information.

Security is now a critical aspect of cyber-physical systems (CPS). Security aspects must be correctly specified, implemented and validated through the system development life-cycle. To do so, engineers need an appropriate notation to capture these requirements and trace them to the code. This security annex is trying to fill that gap by adding the necessary elements to the AADL language to specify security aspects.

For a long time, security has been mostly a matter of coding issues. For example, buffer overflows, use of unsafe functions (e.g. use of legacy functions from the C standard libraries - *strcpy* - instead of their secure versions - *strncpy*), types mismatches, etc have long been the root causes of security issues. However, since the past few years, it has been shown that many vulnerabilities are are rooted in the architectural design: lack of isolation between components at heterogeneous security levels and integrated on the same execution platform, inappropriate use of encryption on network buses physically accessible, etc.

Capturing a software architecture with its security characteristics using a formal notation constitutes the basis for security analysis and vulnerability discovery early in the life cycle. This motivates the need of AADL-specific modeling patterns to capture security requirements. AADL models are then processed by appropriate tools to validate and implement the security policies.

The next sections details the capabilities added in the annex and the related benefits provided by its use.

### Capabilities

The AADL security annex aims at extending the core AADL language with security-specific capabilities:

* **encryption**: encryption mechanisms used to encrypt data transfered through distributed nodes or within a hardware memory.
* **exposure**: physical access of the system hardware (device, processor, memory) that is exposed to potential attackers
* **authentication**: authentication mechanisms between connected elements within the architecture
* **resources isolation**: protection of shared resources between entities operating at different security levels.

### Benefits

This AADL security annex brings the following benefits:

* **verification of security policies**: the annex provides the necessary elements to validate a model against a specific security policy. It will not only check the communications and connections among components (secure system communicating with unclassified entities) but leverage the modeling language (e.g. deployment of components, isolation mechanisms, etc.) to discover potential vulnerabilities related to unsafe resource sharing
* **generation of security assurance**: the annex provides the ability to generate security assurance documents from the models. Potential security-related analysis are (but not limited to): attack surface (list of architectural vulnerabilities and their impact), attack tree (list of vulnerabilities with their related dependencies), attack impact analysis (propagation of a vulnerability within the system).
* **implementation of security policies**: AADL models with adequate security information and implementation details can be processed to generate the system security configuration (e.g. encryption protocols, deployment of encryption keys, configuration of isolation mechanisms). Code generator can leverage the security-specific information to produce the security-related code (generation of security configuration, attack tests, etc).

## Modeling Guidelines

This section details the modeling guidelines to capture each security concept with AADL using this annex. A final example illustrates their usage in a model.

### Security levels and domains

AADL components are associated with security levels. A security level captures the criticality of a component. Common accronym are *secret*, *top-secret* or *unclassified*. Security levels are ordered: the lower the value the higher the security level. The security policy can then explicitly defines security ordering requirements (*e.g. can an unclassified component sends data to a classified one and with what conditions*).

In addition to security level, the AADL security annex captures the concept of *domain*. A domain specifies on *what* the component is operating. Capturing the domain(s) or operation of each component help to analyze if a component has access to too many domain and potentially re-architecture the system to separate domains from each other.

A security *domain* can be interpreted as a tag that defines security boundaries among components. It is designed to provide a second dimension for filtering components in your security policy.

In AADL, security levels are associated with AADL entities using the security\_levels property. The property is a list of integer. Integer values captures the security levels.

Security values are defined on a range from 0 to 100. High values means low security (100 is considered as unclassified). Lowest values means very critical from a security perspective.

This annex defines the following security levels constants: \* top\_secret: associated with a value of 10 \* secret: associated with the value 40 \* unclassified: associated with the value 100

By default, a security level of 100 is associated to each component.

This annex does not define the security policy (legal operations between components according to their security levels). This annex provides the necessary artifact to specify the security constraints. The security policy can be defined (and eventually validated) using a constraint language (such as resolute) or the AADL constraint annex.

The domains are associated with AADL entities using the domains property. The property is a list of string, each string being a domain of interest for the application being specified. A security domain can define a trust boundary or an application domain. A domain represent a set of component operating for a particular purpose and that should be isolated from other domains.

### Verified components

When analyzing an architecture, some components might break the security policy. This type is typical when a component sends or receives data classified at different security levels. However, when designed, analyzed and verified carefully, such a component may not be a security threat, such as a firewall or a gatekeeper. For example, a gateway component that receives secret data, filters it by removing the classified part and sends it to an unclassified component. If not carefully designed and verified, its might leak classified information to unclassified components and security analysis tool must detect this type of design defect. However, this potential security defect might have been considered and the component might have been validated to avoid it. In that case, no warning needs to be issued.

The property trust is an integer that captures the confidence put in a component. The value is within the range 0 (no trust) to 100 (component can be trusted and has been developed with sound and appropriate methods to guarantee isolation of security levels or domains). When the value is 100, the component should not propagate the consequences of a vulnerability.

By default, the trust property value is set to 0 so that analysis tools are said to be pessimistic. Modeling users have to set their own value according to their knowledge of the system. This annex does not define how the value should be assigned: the value is user- and project-dependent.

### Encryption

The concept of encryption is captued using the encryption property.

The encryption specifies the encryption method and requirements to encrypt data. This annex provides the ability to capture two kinds of encryption: 1. Encryption at the **application level**: encryption is performed by the application itself and is not dependent on the underlying runtime (protocols, bus, etc.). Encryption at this level is captured by attaching the encryption property on components ports. 2. Encryption at the **execution runtime level**: encryption is then provided by the underlying execution platform and not provided by the application itself. This type of encryption is then provided by the communication protocol (for example, SSL) or underlying middleware.

The encryption property is a record that captures all information necessary to encrypt the data, either for a protocol or an application interface (interfaces of an AADL thread, parameters of an AADL subprogram). The record members are: \* method: capture the type of encryption used. Possible values are symmetric, assymmetric or clear (no encryption) \* algorithm: capture the algorithm used to encrypt the data. Potential values: des, tripledes, rsa, aes, blowfish \* private\_key: private key used for asymmetric protocols \* public\_key: public key used for asymmetric encryption protocols \* key: encryption key for symmetric encryption algorithms \* operation\_mode : mode of operation for block cipher algorithms

### Authentication

Authentication is specified using the authentication\_method property. This property captures the method used to authenticate an entity in the system. This property is associated to a virtual bus. The virtual bus can then

Supported authentication methods are defined in the enumeration list supported\_authentication\_methods. They are currently set to the following list: \* shared\_password: a single password to access a resource \* user\_password: a combination of username and password. \* key: an encryption key (symmetric, assymetric) \* ipaddr: the ip address of the sender

The list of supported authentication method can be refined by the annex users in order to match the authentication mechanisms used in their systems.

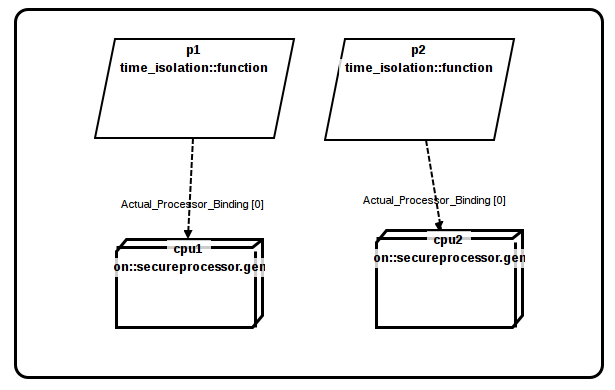
### Time Isolation

This is important to ensure that components executed on the same processor can share the timing resource (e.g. processor) in a secure manner. Timing can be the source of many security issues. A component should not be able to take all the processor resource and let the other tasks without any potential execution time. This can be a *Denial of Service* attack. Similarly, timing information (e.g. execution time of a task) can be used to predict the task behavior and create attacks. Such attack can be avoided by using a periodic and fixed time slice for all tasks.

Time isolation is implemented by ensuring that a software component a fixed, deterministic time frame to be executed. Time isolation can be achieved by physical of logical isolation.

Physical time isolation is implemented by separating software components on separate processors. Being separated on separate processor, one task cannot interfere with another. However, if these functions are connected, this is necessary to ensure space isolation on the communication layers (such as encryption of bus, check data integrity)

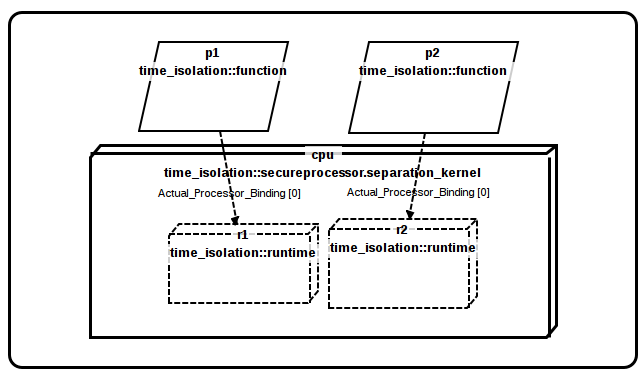
Physical time isolation is captured by allocating the AADL process components to different AADL processor components. The allocation is specified with the actual\_processor\_binding property from the core standard.



Physical Time Isolation

Logical time isolation is implemented using a dedicated operating system, also called a separation kernel. The separation kernel allocates a pre-defined, fixed time-slice to each component. The component uses its allocated time slice to execute its code. The component cannot use more or less time than the one being allocated.

Logical time isolation is specified by reusing the modeling patterns from the AADL ARINC653 annex: the operating system (isolation kernel) is specified using an AADL processor component and each partition is defined using an AADL virtual processor component bound to the processor. Then, the ARINC653::Module\_Schedule property captures the list of partitions slots allocated to each partition. For more information about this modeling pattern, please read the AADL ARINC653 modeling annex.



Logical Time Isolation

### Space Isolation

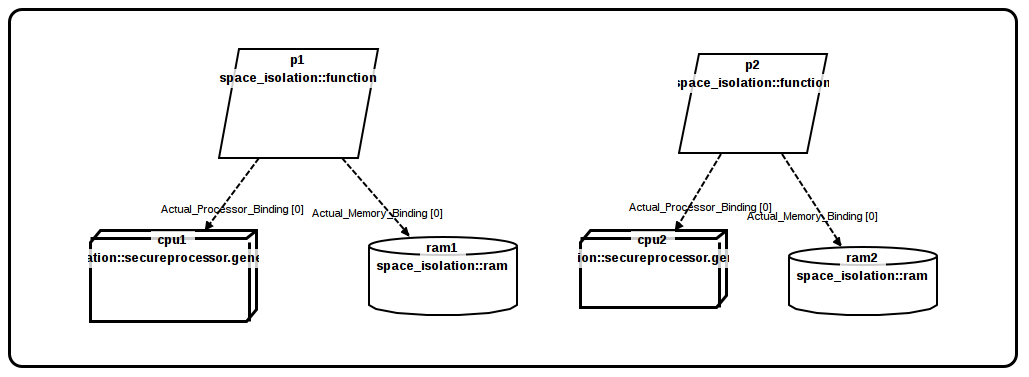
Space isolation consists of ensuring that data is protected from unauthorized entities. Only allowed entities can access to a memory area and/or communicate with other entities.

An AADL process component specifies an address space. This address space is associated with a memory area by binding the process to a memory component.

AADL thread components located in the same process have access to all data within the process address space. For that reason, it might be important to avoid threads with different security levels within the same process.

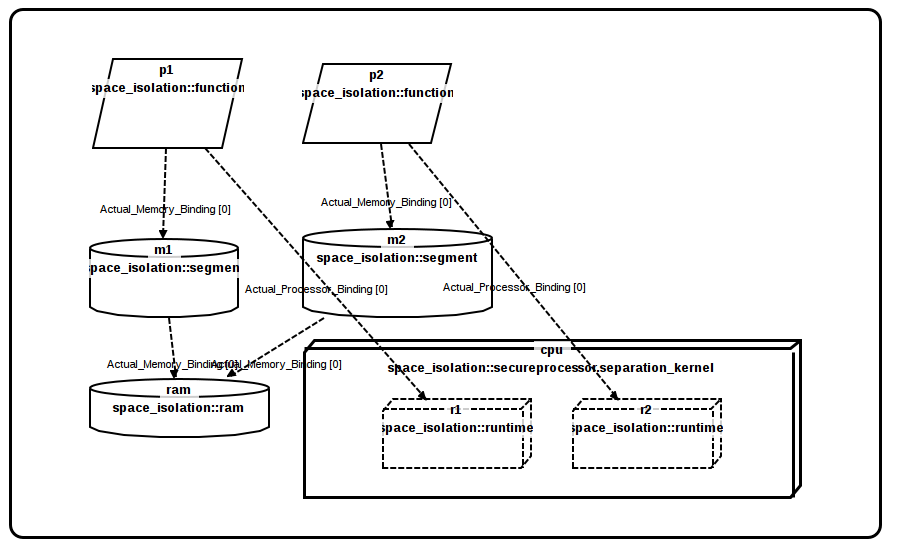
As for time isolation, space isolation can be implemented physically or logically.

Physical memory isolation consists in isolating processes in separate hardware memories. From an AADL perspective, each AADL process component (address space) is associated with a single AADL memory component. As processes are located on different and separate memory components, this is not possible for one component to access (e.g. read/write) data from the other.



Physical Space Isolation

Logical memory isolation consists in isolating processes in memory segments from the same hardware memories. Processes are executed on the same processor and use the same hardware memory. The underlying operating system uses the Memory Management Unit (MMU) to separate processes into memory segment and control data access (e.g. read/write) to these segments. From an AADL perspective, logical isolation is realized by associating AADL process components to AADL memory components that are ultimately associated to a unique AADL memory components (which captures the hardware memory). Logical memory isolation requires a processor with a Memory Management Unit (MMU) and an operating system that is able to use this capability to isolate process in memory segments.



Logical Space Isolation

These mechanisms ensure data protection in local (hardware) memory. However, it does not protect the data being exchanged using communication channels. Appropriate mechanisms should be used to ensure space isolation accross distributed components.

### Exposure to the physical environment

Components can be physically exposed to their environment. This exposure offers access to the resources that can then be modified, altered, tampered by attackers and ultimately cause a potential security threat. For example, having a bus physically accessible might be a security threat: random attackers can try to take advantage of this resource and compromise devices and computers connected to it.

The level of exposure depends on the components and its deployment in the environment. Exposure can be intrinsic to the component (e.g. a wireless network is accessible no matter how it is deployed) or depend on its usage (e.g. a network cable can be physically exposed or hidden and unaccessible to potential attackers). For that reason, components exposure cannot be captured with a boolean value and need to be specified using a number.

Exposure of a component is captured using the exposure property from the security annex. The exposure is an integer number within the range 0 ... 100. Low value indicates low level of exposure (0 means that the component is isolated). High values indicates high level of exposure. Default value is 100 (the annex assumes by default that components are exposed to external attackers).

## Model Validation

Using the annex, model analysis tools can check the models enforces a security policy. As AADL is a description language, it is not designed to include requirements. However, the model and its associated security annotations are sufficient to check security requirements.

The list of security requirements that can be validated using the model includes (but is not limited to): 1. security levels isolation 2. time isolation of components 3. encryption of data 4. use of authentication on physical connections 4. potential covert channels in the component assembly

Various analysis tools can be used to check the model. Constraints can be defined using the AADL constraints annex. At the time of writing this document, the constraints annex has not yet be published so that security constraints cannot be expressed using it for now.

## Examples

### Security Levels & Encryption with Shared Bus

In this first example, three different systems are included: 1. A sender producing top-secret data 2. A filter, taking the top-secret data, filters the unclassified part and output it 3. The consumer reads and processes the unclassifier data

In this example, we have physical and logical isolation: \* **logical isolation** use of encryption protocols on software connections. The connections between all systems is using the same bus (shared resource). The different security levels are protected by using a different encryption mechanism. Processes (sender and filter) exchanging top-secret (through connection c0) data protect data from potential attackers by encrypting it (binding to p\_https). The unclassified data is not protected (binding to p\_http that does not specify any security mechanism).

* **physical isolation** is achieved by using a separate processor for each system, the execution of one system cannot interrupt the other one.

package example\_mils  
public  
 --  
 -- This example show a MILS system and how we achieve isolation of security levels  
 -- The system consists in three systems  
 -- 1. A sender that produces top secret data  
 -- 2. A filter that takes the secret data, filter it and just output the unclassified part  
 -- 3. A consumer that reads the unclassified filtered data and processes it  
 --  
 -- Isolation is achieved like this:  
 -- 1. Each process is on its own processor (physical isolation)  
 -- 2. The connection that transport secret data is using encryption  
 --  
 -- On the other hand, the following aspects must be checked:  
 -- 1. The secret data being transported is encrypted because  
 -- the bus transports topsecret and unclassified data  
 -- 2. Each system is isolated on a separate processor  
 -- 3. The encryption method used to encrypt data is not too weak  
 --  
  
 with security\_properties;  
 with data\_model;  
  
 data mydata  
 properties  
 Data\_Model::Data\_Representation => Integer;  
 Data\_Model::Integer\_Range => 0 .. 100;  
 Data\_Model::Number\_Representation => signed;  
 end mydata;  
  
 virtual bus tcp  
 end tcp;  
  
 virtual bus ssl  
 properties  
 security\_properties::encryption =>  
 [ method => symetric;  
 algorithm => des;  
 ];  
 allowed\_connection\_binding\_class => (classifier (tcp));  
 end ssl;  
  
 virtual bus http  
 properties  
 -- allowed\_connection\_binding\_class specifies what are the bus you can  
 -- bind to. In this example, http can be bound on top of other  
 -- protocols, such as tcp (no encryption) or ssl (encryption)  
 allowed\_connection\_binding\_class => (classifier (tcp), classifier (ssl));  
 end http;  
  
  
 bus eth  
 end eth;  
  
  
 system producer  
 features  
 dataout : out data port mydata;  
 properties  
 security\_properties::security\_levels => (security\_properties::top\_secret);  
 end producer;  
  
  
 system filter  
 features  
 dataout : out data port mydata  
 {security\_properties::security\_levels => (security\_properties::top\_secret);};  
 datain : in data port mydata  
 {security\_properties::security\_levels => (security\_properties::unclassified);};  
 properties  
 security\_properties::trust => 100;  
 security\_properties::security\_levels => (security\_properties::top\_secret,security\_properties::unclassified);  
 end filter;  
  
  
 system consumer  
 features  
 datain : in data port mydata;  
 properties  
 security\_properties::security\_levels => (security\_properties::unclassified);  
 end consumer;  
  
 system integration  
  
 end integration;  
  
 processor cpu  
 features  
 b : requires bus access eth;  
 end cpu;  
  
  
 system implementation integration.i  
 subcomponents  
 prod : system producer;  
 filt : system filter;  
 cons : system consumer;  
  
 -- Part of the shared bus is exposed to public  
 -- but still difficult to access  
 b : bus eth {security\_properties::exposure => 50;};  
  
 -- The computer is not physically accessible  
 cprod : processor cpu {security\_properties::exposure => 10;};  
  
 -- The computer is in a private room  
 cfilt : processor cpu {security\_properties::exposure => 20;};  
  
 -- The computer is in a public room, showing traffic information  
 -- on a screen (physically accessible to the public);  
 ccons : processor cpu {security\_properties::exposure => 100;};  
  
 p\_http : virtual bus http;  
 p\_https : virtual bus http;  
 p\_ssl : virtual bus ssl;  
 p\_tcp : virtual bus tcp;  
 connections  
 b0 : bus access cprod.b <-> b;  
 b1 : bus access cfilt.b <-> b;  
 b2 : bus access ccons.b <-> b;  
  
 c0 : port prod.dataout -> filt.datain;  
 c1 : port filt.dataout -> cons.datain;  
  
 properties  
 actual\_processor\_binding => (reference (cprod)) applies to prod;  
 actual\_processor\_binding => (reference (cfilt)) applies to filt;  
 actual\_processor\_binding => (reference (ccons)) applies to cons;  
  
 -- Description of the network stack  
 actual\_connection\_binding => (reference (p\_ssl)) applies to p\_https;  
 actual\_connection\_binding => (reference (p\_tcp)) applies to p\_ssl;  
 actual\_connection\_binding => (reference (p\_tcp)) applies to p\_http;  
  
 -- Finally, the lower end of the stack is associated with the bus  
 actual\_connection\_binding => (reference (b)) applies to p\_tcp;  
  
 actual\_connection\_binding => (reference (p\_https)) applies to c0;  
 actual\_connection\_binding => (reference (p\_http)) applies to c1;  
 annex resolute {\*\*  
 prove (check\_mils\_policy ())  
 \*\*};  
 end integration.i;  
end example\_mils;

### Security Domains & Shared Processor

In the following examples, we have five processes sharing security domains. The goal of this example is to illustrate how to model processes with isolation of security domains.

The different processes are the following: 1. audio, video and display belong to the same security domain (**entertainment**) 2. filter belongs to three security domains: **entertainment**, **navigation** and **control** 3. navigation belongs to a single security domain (**navigation**) 4. can\_controller belongs to a single security domain (**control**)

In the following model, the filter component handles three different security domains. We assume the component separates the different domains correctly because it has been verified (the property trust is set to be 100).

In this model, each process is separated in terms of space (allocation to a separate runtime) and time (allocation to separate memory segment). The main execution platform provides isolation between partitions (property runtime\_protection\_support on the processor set to true).

package example\_domains  
public  
 with security\_properties;  
 with arinc653;  
  
 ------------------------------  
 -- Processor & Partitions --  
 ------------------------------  
  
 bus wifi  
 end wifi;  
  
 bus can  
 end can;  
  
 virtual processor partition  
 end partition;  
  
 processor secure\_platform  
 properties  
 runtime\_protection\_support => true;  
 end secure\_platform;  
  
 ------------------------  
 -- Memory component --  
 ------------------------  
  
 memory hardware\_memory  
 end hardware\_memory;  
  
 virtual memory segment  
 end segment;  
  
 process audio  
 properties  
 security\_properties::domains => ("entertainment");  
 end audio;  
  
 process display  
 features  
 sensorsdata : in data port;  
 properties  
 security\_properties::domains => ("entertainment");  
 end display;  
  
 process video  
 properties  
 security\_properties::domains => ("entertainment");  
 end video;  
  
  
 process filter  
 features  
 outsensorsdata : out data port;  
 drivingctrl : in data port;  
 insensorsdata : in data port;  
 actuatorctrl : out data port;  
 properties  
 security\_properties::trust => 100;  
 security\_properties::domains => ("entertainment", "navigation", "control");  
 end filter;  
  
  
 process navigation  
 features  
 drivingctrl : out data port;  
 properties  
 security\_properties::domains => ("navigation");  
 end navigation;  
  
 process can\_controller  
 features  
 rawsensorsdata : in data port;  
 processedsensorsdata : out data port;  
 properties  
 security\_properties::domains => ("control");  
 end can\_controller;  
  
  
 system integration  
 end integration;  
  
 system implementation integration.i  
 subcomponents  
 --  
 -- Processors  
 --  
 entertainment\_runtime : virtual processor partition;  
 filter\_runtime : virtual processor partition;  
 can\_runtime : virtual processor partition;  
 navigation\_runtime : virtual processor partition;  
 cpu : processor secure\_platform;  
  
 --  
 -- Memories  
 --  
 hw\_memory : memory hardware\_memory;  
 filter\_segment : virtual memory segment;  
 can\_segment : virtual memory segment;  
 entertainment\_segment : virtual memory segment;  
 navigation\_segment : virtual memory segment;  
  
 --  
 -- Software components  
 --  
 audio\_pr : process audio;  
 video\_pr : process video;  
 display\_pr : process display;  
 filter\_pr : process filter;  
 can\_controller\_pr : process can\_controller;  
 navigation\_pr : process navigation;  
 properties  
 actual\_processor\_binding => (reference (cpu)) applies to entertainment\_runtime, filter\_runtime, can\_runtime;  
 actual\_processor\_binding => (reference (entertainment\_runtime)) applies to audio\_pr, video\_pr, display\_pr;  
 actual\_processor\_binding => (reference (filter\_runtime)) applies to filter\_pr;  
 actual\_processor\_binding => (reference (navigation\_runtime)) applies to navigation\_pr;  
 actual\_processor\_binding => (reference (can\_runtime)) applies to can\_controller\_pr;  
  
 actual\_memory\_binding => (reference (hw\_memory)) applies to entertainment\_segment, filter\_segment, can\_segment;  
 actual\_memory\_binding => (reference (entertainment\_segment)) applies to audio\_pr, video\_pr, display\_pr;  
 actual\_memory\_binding => (reference (filter\_segment)) applies to filter\_pr;  
 actual\_memory\_binding => (reference (navigation\_segment)) applies to navigation\_pr;  
 actual\_memory\_binding => (reference (can\_segment)) applies to can\_controller\_pr;  
  
 arinc653::module\_major\_frame => 60 ms applies to cpu;  
 arinc653::module\_schedule =>  
 (  
 [partition => reference(entertainment\_runtime) ; duration => 20 ms ; periodic\_processing\_start => true;],  
 [partition => reference(filter\_runtime) ; duration => 10 ms ; periodic\_processing\_start => true;],  
 [partition => reference(can\_runtime) ; duration => 10 ms ; periodic\_processing\_start => true;],  
 [partition => reference(navigation\_runtime) ; duration => 20 ms ; periodic\_processing\_start => true;]  
 ) applies to cpu;  
 end integration.i;  
  
end example\_domains;

## Properties

property set security\_properties is  
  
 --  
 -- The security\_levels property indicates what is the levels of a component.  
 -- This is a list because a component can handle more than one level. Also,  
 -- when using several levels, the component is said MLS (Multiple Level of Security).  
 -- We assume that the highest number (100) means low security level (e.g. unclassified) while  
 -- low numbers are high security (e.g. top-secret).  
 --  
 security\_levels : list of aadlinteger => (100) applies to (all);  
  
 -- The following are constants we can use as security levels. The intent  
 -- is to be able to use natual language when adding security levels.  
 top\_secret : constant aadlinteger => 10;  
 secret : constant aadlinteger => 40;  
 unclassified : constant aadlinteger => 100;  
  
 --  
 -- The domains property indicates all the domain a component or a data can handle.  
 -- The domain separate the concern of the component (on what he is working).  
 -- Separation of domains is important (components handle only the domains)  
 -- they are associated with.  
 --  
 domains : list of aadlstring applies to (all);  
  
 --  
 -- The trust property captures how much confidence can be put into  
 -- a component. The 0 value means that the component cannot be trusted  
 -- at all. On the other hand, a value of 100 means that the component  
 -- has been designed and implemented with sound techniques that  
 -- take care of many security issues.  
 --  
 trust : aadlinteger 0 .. 100 => 0 applies to (all);  
  
 -- exposure is a property that indicates how much a component is accessible  
 -- and exposed to external threats. Examples are an ethernet network  
 -- of a wifi network. For a ethernet network, we could then assume  
 -- that it is more difficult because there is a physical access required.  
 -- For the wireless, it is exposed to everybody, so, this is more difficult  
 -- to protect. Thus, the exposure for the wireless should be greater than  
 -- the ethernet.  
 --  
 -- The exposure can be applied to:  
 -- \* a bus - physical access to the wire. Depending  
 -- on the location and protocol, it might  
 -- be higher or not (e.g. cable vs. wireless)  
 -- \* a virtual bus - exposure of the data (lack of protection)  
 -- \* processor - physical access to the processor, potential  
 -- flash of the chip  
 -- \* memory - physical access to a storage unit - can flash it  
 -- \* device - can replace the device with another device  
 -- \* system - the system is exposed physically and attackers  
 -- can attack its content  
 exposure : aadlinteger => 100 applies to (bus, virtual bus, processor, device, system, memory);  
  
 --  
 -- Encryption property indicate the encryption mechanism for a port  
 -- or a virtual bus. The property is attached to the port/bus/virtual bus  
 -- to specify is the data is encryption and if yes, what protocol it uses.  
 encryption : security\_properties::encryption\_type applies to (port, virtual bus, bus, memory, access);  
  
 encryption\_type : type record (  
 -- Do we need to distinguish the method from the algorithm?  
 -- It might be useful to define the method without having  
 -- to specify the algorithm.  
 --  
 -- On the other hand, it is possible to deduce the method  
 -- from the algorithm.  
 --  
 method : security\_properties::supported\_encryption\_method;  
 algorithm : security\_properties::supported\_encryption\_algorithm;  
 private\_key : aadlstring;  
 private\_key : aadlstring;  
 key : aadlstring;  
 operation\_mode : security\_properties::supported\_operation\_mode;  
 );  
  
 supported\_encryption\_method : type enumeration (symetric, assymetric, clear);  
 supported\_encryption\_algorithm : type enumeration (tripledes, des, rsa, blowfish, aes, clear);  
  
  
 -- Operation mode for block cipher algorithms.  
 -- See. https://en.wikipedia.org/wiki/Block\_cipher\_mode\_of\_operation  
 supported\_operation\_mode : type enumeration (ecb, cbc, pcbc, cfb, ofb, ctr);  
  
 --  
 -- The supported\_communication\_protocol is an enumeration of potential protocols  
 -- used on bus and virtual buses. They are used with the property  
 -- security\_properties::protocol  
 --  
 supported\_communication\_protocols: type enumeration (http, ethernet, ftp);  
  
 protocol : security\_properties::supported\_communication\_protocols applies to (bus, virtual bus);  
  
 --  
 -- The supported\_authentication\_methods represents the different type of authentication  
 -- available for modeling communication.  
 --  
 supported\_authentication\_methods: type enumeration (shared\_password, user\_password, key, ipaddr);  
  
 authentication\_method : list of security\_properties::supported\_authentication\_methods applies to (bus, virtual bus);  
  
end security\_properties;