SCADE AADL

Thierry Le Sergent, Adnan Bouakaz, Guilherme Goretkin (ANSYS)

thierry.lesergent@ansys.com
ANSYS Esterel Technologies
9, rue Michel Labrousse,
31100, Toulouse, France

adnan.bouakaz@ansys.com
ANSYS Esterel Technologies
Twins 1; 679 avenue Julien Lefebvre
6270, Villeneuve-Loubet, France

guilherme.goretkin@ansys.com
2600 ANSYS Dr.
Canonsburg, PA, 15317, USA

Keywords: Model Based System Engineering, System analysis, Software Engineering, AADL, SysML, Domain Specific Language, Model Transformation, SCADE

1 Introduction

AADL [1], acronym for Architecture Analysis and Design Language, is a standard promoted by SAE International, AS-2C committee, driven by inputs from the avionics and space industry. It is a mature, living standard with version 1.0 published in 2004. The current version is v2.2 and active work is on-going for the future v3.0 version.

AADL is dedicated to real-time embedded system, modeling both the software and hardware resources [5,6,8]. It is extensible through annexes, both standard and custom, bringing detailed specifications especially for verification of non-functional systems requirements such as performance, safety [7,9,10]. Several tools, for example the open-source tool OSATE support the different AADL representations in graphical, textual, and XML format [3].

SCADE [4], acronym for Safety Critical Application Development Environment, is a suite of strongly integrated tools covering the following aspects:

- SCADE Architect: Model Based Systems Engineering tool, based on SysML and further extensible to support Domain Specific Languages (DSL) via a dedicated module named “Configurator” [12]. This module has been used to set up a complete solution for avionics systems, from functional, software, and platform architecture with details for ARINC 429 and AFDX avionics communication protocols [13].
- SCADE Suite: Industry-proven solution dedicated to the development of safety critical embedded software [11,12]. The SCADE Suite code generator is qualified according to DO-178C/DO-330 at TQL-1. SCADE Suite and SCADE Architect complement each other. SCADE Architect models provide the system level view with the key objective to define the interface of the software components, and propagate the exchanged information through the architecture of the system. SCADE Suite models allow detailed definition and qualified code generation of the software components. Though supported by completely different languages, parts of the SCADE Architect and SCADE Suite models can be synchronized to each other.
- SCADE Display: Model-based HMI software design solution, designed for displays with safety objectives, with an associated prototyping and development process [16]. SCADE Display models connect to SCADE Suite model at two levels: For overall application behavior and for internal behavior of complex SCADE Display widgets.
- SCADE Test: Complete set of verification and validation tools that supports requirements-based testing with re-use of simulation cases for host and target testing and model coverage analysis [17].

The objective of SCADE AADL is to:

1. Provide a graphical AADL modeler tool compliant to the AADL standard to benefit from the large set of analysis tools that input AADL models. SCADE AADL tool imports and exports standard AADL files.
2. Simplify the modeling and understanding of AADL models through straightforward, direct, and complete definition of the components as single objects.
3. Provide a seamless path to SCADE Suite, for the development of software components that benefit from the qualified tool chain, code generation and tests.

The capabilities of SCADE AADL modeler are summarized in Figure 1.

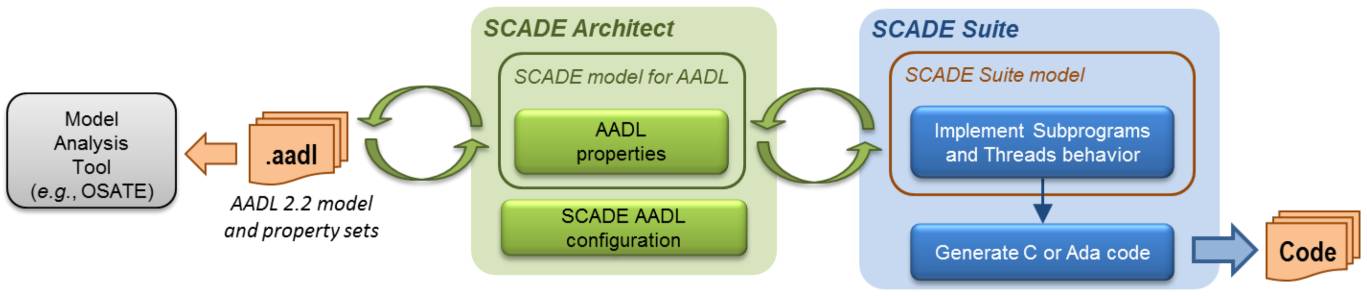


Figure 1 SCADE AADL modeler workflow

The focus of this paper is to detail how the AADL and SCADE worlds have been connected and how appropriate DSL simplifications made possible by original tool capabilities can make AADL modeling significantly easier than current methods. Sections 2.1 and 2.2 provide the rationale and means for AADL simplifications and mapping to SCADE Architect concepts.

An important aspect of AADL is its extensibility with annexes, defined by both meta-model extensions and property sets, as described in Section 2.4.

For the implementation of the software components, i.e. the algorithms, defined through AADL Threads or Subprograms, we propose to rely on the SCADE Suite language and toolset. The SCADE AADL solution provides a means to keep the AADL description of threads and subprograms and their corresponding implementation in SCADE Suite synchronized. This synchronization is explained in Section 3.

Section 4 details an example model to illustrate how SCADE AADL elegantly captures the specification of a particular software architecture deployed on different platforms.

Finally, Section 5 compares our work with related work, while Section 6 concludes and gives our vision for the next step: Designing systems with consistent multi-views including compliance with different standards such as AADL and FACE [18].

2 AADL simplifications and mapping to SCADE Architect

AADL provides as part of its core language explicit definitions of software and hardware component categories like process, thread, subprogram, device, processor, memory, bus, and others. Each comes with its own set of properties, interface definition, and relationships with the other component categories. Detailed presentation of the language can be found at [1,5,6,8].

The AADL language is based on object oriented constructs, providing powerful high-level abstraction capabilities. For the specialists who are modeling the system and not necessarily trained as software engineers familiar with these paradigms, these abstractions may add unnecessary complexity. Indeed, navigating from component instance to its implementation, to its type and possibly to supertypes, abstractions and prototypes to get the full knowledge on a component may not be easy, even though good model based tools might help. The abstractions supported in an AADL model can be summarized in three levels:

1. Prototypes and Abstract components: later extended and refined into concrete categories of Components.
2. Component Types and Component Implementations: The benefit is that an interface definition set in a Component Type can have multiple implementations in different Component Implementations. But interface definition in a Component Type is mandatory before specifying implementation; thus, there are always two separated objects to create and manage in the tool, and/or in AADL textual format.
3. Instantiation: Component Instances are references to component types and implementations, that must be in-lined for analysis. In-lining/instantiation is done as an explicit tool action in the OSATE tool [3] to get an instantiated model.

The proposal in the SCADE AADL solution is a significant simplification for system engineers with direct management of the component instances while still allowing full compatibility with the AADL standard. The simplification is supported by AADL abstraction and inheritance in-lining and AADL instance-based modeling. These two levels of simplification are explained in the next subsections.

2.1 SCADE AADL as a DSL on SCADE Architect

SCADE Architect is a domain-agnostic system modeling language and toolset, implemented transparently on top of UML and SysML. It defines a layer that can be naturally extended to accommodate domain-specific needs without starting from scratch: The domain specialist designs a domain-specific language as a meta-model that specializes SCADE Architect's domain-agnostic meta-model. Additional information is conveyed by appropriate attributes in specializing meta-classes and relationships among them and additional constraints imposed by the domain are enforced by redefinitions of existing relationships. SCADE Architect transparently handles the intricacies of UML profile management and dynamically adjusts its interface to the domain-specific "configuration", offering new modeling constructs (in toolbars, property pages, etc.) while forbidding others according to constraints of the domain.

SCADE Architect Configurator enables easy definition and one-click generation of DSLs. The SCADE AADL DSL is defined through the following steps:

- Import the official AADL meta-model as a SCADE Architect configurator metamodel (starting from the aadl.ecore file). This is mainly an Ecore to UML transformation.
- Map AADL meta-classes to SCADE Architect meta-classes using inheritances. AADL constraints such as containment constraints are imposed by redefinition of existing relations in SCADE Architect DSL.
- Automatically generate a new configuration plugin (including among several technical artifacts the new UML profile corresponding to the AADL meta-model extension).
- By loading this configuration plugin, the SCADE Architect modeler becomes an AADL modeler with customized "creation palette", object properties, and editing helpers.

The major step presented here is the AADL-SCADE Architect SysML subset mapping. This is what specifies the capabilities and constraints, including the language simplifications, for the end-users in the creation of their SCADE AADL models. All concrete SCADE AADL meta-class must inherit from a predefined SCADE Architect meta-class. The corresponding SCADE AADL objects will be supported in the modeling tool exactly in the way SCADE Architect objects are manipulated. For example, the AADL meta-class "EventPort" that is part of the possible interface for a "Device" is mapped to a SysML "FlowPort", and "Device" is mapped to SysML "Block".

The key considerations for this mapping are 1) non-violation of SysML constraints, 2) expected user interface for each SCADE AADL objects; for example, the "EventPort" can be graphically set on the edge of a "Device" represented on a diagram as a "box", in the same way a SysML "FlowPort" is represented on the edge of a "Block". In practice, this mapping is quite natural and leads to an intuitive modeling tool: all AADL components are mapped into SCADE Architect "Blocks", AADL features are mapped into SCADE Architect "Ports", AADL connections are mapped into SCADE Architect "Connectors", AADL Modes bindings are mapped into SCADE Architect "Allocations", etc.

The class diagram below shows an extract of SCADE AADL meta-model. The black color meta-classes come from the AADL meta-model. The green color "Block" meta-class is the SCADE Architect predefined meta-class supporting SysML Blocks. SCADE AADL defines the red color "Process" meta-class; it inherits from both "ProcessType", where the AADL Process interface is defined, "ProcessImplementation", where the AADL Process content is defined, and "ProcessSubcomponent", which represent instances of AADL processes in implementations, and indirectly, from SCADE Architect "Block" meta-class. That way, a single object kind, "Process" will be used in SCADE AADL to define and use AADL processes. The absence of distinctions between "definition" and "instance" is possible thanks to the SCADE Architect Replica mechanism [14].

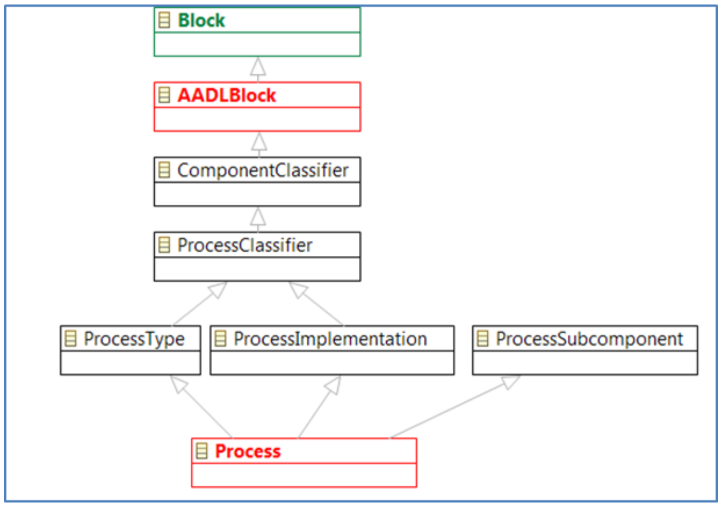


Figure 2: Extract of SCADE AADL meta-model

Graphical styles are also applied to set the shape and icon for each SCADE AADL object so that the tool interface follows the standard graphical presentations defined in the AADL standard, as opposed to the typical SysML appearance.

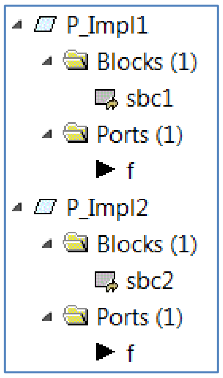
Though the mapping between SCADE Architect and AADL concepts is in most of the time straightforward and intuitive, some concepts are “suppressed” from the configuration, and handled by the AADL to SCADE AADL transformation, as described in Section 2.2 in more details. This choice of simplifying the SCADE AADL DSL was driving by our objective to implement an ergonomic AADL modeler targeting industry usage.

2.2 Model transformation between standard AADL and SCADE AADL

Concepts such as abstract, refinement, extension, and prototype are removed in SCADE AADL (the corresponding meta-classes are not mapped to any SCADE Architect meta-classes). The AADL models defined in other tools, or simply in the textual AADL format, can still be loaded in SCADE AADL through model transformation. This section illustrates this transformation with simple examples; the complete AADL language is very rich and complex and the transformation handles all its subtleties.

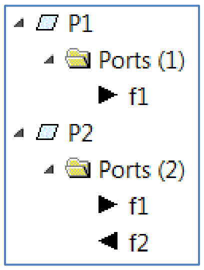
1. A SCADE AADL concept such as “Process” presents at the same time a Process type and a Process implementation. The import merges the two definitions in a single object, while the export splits the definition of a SCADE AADL component into a type and an implementation. The following example illustrates this transformation.

```
process P
  features
    f: in data port Base_Types::Unsigned_16;
end P;

process implementation P.Impl1
  subcomponents
    sbc1: data Base_Types::Unsigned_16;
end P.Impl1;

process implementation P.Impl2
  subcomponents
    sbc2: data Base_Types::Float_64;

end P.Impl2;
```


```
P_Impl1
  Blocks (1)
    sbc1
  Ports (1)
    f
P_Impl2
  Blocks (1)
    sbc2
  Ports (1)
    f
```

2. Extension is handled by in-lining (i.e., inheriting) elements and properties of the super component into the children, and considering refinements if any. In the following example, Process P2 extends Process P1, and refines its feature.

```
process P1
  features
    f1: in data port;
end P1;

process P2 extends P1
  features
    f1 : refined to in data port  Base_Types::Unsigned_16;
    f2 : out data port Base_Types::Unsigned_16;

end P2;
```

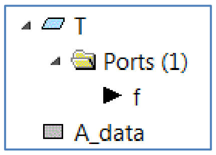

```
P1
  Ports (1)
    f1
P2
  Ports (2)
    f1
    f2
```

3. Prototypes are resolved by the transformation, and ignored if they are not bounded. In the following example, feature “f” of SCADE AADL Process P2 is typed with Unsigned_16 since data prototype “p” has been bounded to this data type; it is however not typed in Process P1.

```
process P1
  prototypes
    p: data;
  features
    f: in data port p;
end P1;

process P2 extends P1 (p => data Base_Types::Unsigned_16)

end P2;
```

4. Abstract elements are imported by the transformation only if they are refined into concrete elements. In the following example, abstract component A has been imported as a Data type only because it is used as a data classifier for feature “f” of Thread “T”.

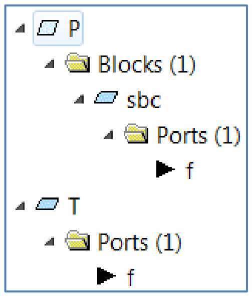
```
abstract A
end A;

thread T
  features
    f: in data port A;

end T;
```



T
Ports (1)
f
A_data

5. AADL subcomponents are references to component types and implementations that must be in-lined for analysis. In-lining/instantiation is done as an explicit tool action in the OSATE tool [3] to get an instantiated model. SCADE Architect provides a direct instantiation mechanism through the usage of replicas, explained in Section 2.3. The transformation takes advantage of this mechanism and represents subcomponents as replicas, and considers refinements and prototypes resolution. Feature “f” of P::sbc is typed with Unsigned_16 according to prototypes resolution.

```
thread T
  prototypes
    p: data;
  features
   f: in data port;
end T;

process P
end P;

process implementation P.impl
  subcomponents
    sbc : thread T (p => data Base_Types::Unsigned_16);
end P.impl;
```



P
Blocks (1)
sbc
Ports (1)
f
T
Ports (1)
f

2.3 *SCADE Architect replica: means for direct instance model*

A modeling language that supports the definitions of “blocks” whatever they represent, which can be reused several times in the model through “parts” is a great benefit for systems engineers. Both SysML and AADL offer that feature. This “multi-instantiation” of components is naturally used to model the fact that, for example, an equipment is “duplicated” in the system. But this paradigm raises a difficulty; System engineers need to specify information related to each individual component instance in the system.

Classical SysML modeling tools handle system components as a set of blocks, each containing parts typed by other blocks. The problem arises on the second level of the “conceptual hierarchy”: the tools handle only one object to represent each part at the second level. There is no direct means to specify information on the individual “conceptual parts” at the second level, e.g. set the pressure of each of the four tires of a car if the tire is defined as a part of a wheel block that is instantiated four times in the car; there is only one “object handle” for the tires.

In AADL, the issue is solved through a syntactic means to address these objects with a path.

```
properties
        Pressure => 23 applies to Car.Wheel_FrontLeft.Tire
```

This way allows specifying all expected information, with the inconvenience of “spreading” the information over the AADL specification; one has to navigate between the different level of refinements to get a complete view of each component.

Users launch the Instantiate feature to create an instantiated model, generating all the components replica as specified through the components implementations and components instances, and inheritance and refinement.

To allow system engineers to manage, in a straightforward manner, their architecture designs, SCADE Architect manages automatically the “replica” from the specification specified with blocks and parts [14]. All intermediate “block types” are automatically created internally to comply with the SysML standard, and are managed by the tool to remain consistent with the user design. That way, each conceptual object has a concrete immediate existence in the tool; in short, the instance model is managed dynamically during the modeling activity.

The example shown in Section 4 illustrates the way this replication feature is represented in SCADE AADL tool interface: each "conceptual" object is represented in the model tree view; it is used for representation in diagrams, for property value settings, for bindings, but as replica of a component definition, its internal structure can only be modified from the component definition itself.

2.4 AADL Property Sets and Annexes

The same mechanism that allows the definition of SCADE AADL on top of SCADE Architect DSL, SCADE Architect Configurator allows the creation of new DSLs on top of SCADE AADL: the concepts of the new DSL must inherit from the SCADE AADL meta-classes or from the SCADE Architect meta-classes. This flexible mechanism of defining new DSLs on top of existing ones allows the extension of the SCADE AADL configuration plug-in with new Annexes, defined statically as new configurations plug-ins and applied on demand to a SCADE AADL project.

Property sets could also be implemented as meta models with the SCADE Architect Configurator, but an automatic solution is more suitable. The SCADE AADL modeler takes advantage of the built-in mechanism of annotations of SCADE Architect to handle property sets. This mechanism is like the property sets mechanism:

- The user defines an annotation type file that decorates meta-classes in the DSL with more attributes.
- The annotation types file can be loaded on demand and applied to the project. SCADE Architect dynamically adjusts the interface to allow adding such annotations to objects.

Annotations are hence a natural and intuitive representation of AADL properties:

- SCADE AADL modeler allows importing property sets along with normal AADL models. The property sets are transformed into annotation types files and applied to the project.
- Properties are transformed into annotations and added to the corresponding objects. The export transforms the annotations back into properties.

The following example briefly illustrates this process. The property set "PS" defines a new property type called "Security_Level" that can be only applied to objects of kind Thread. That property is given value 5 for Thread "T".

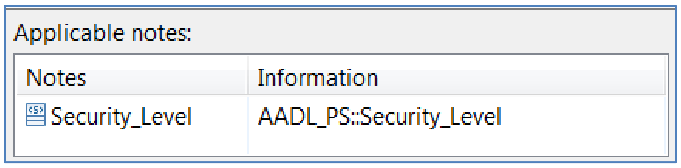
```
property set PS is
  Security_Level: aadlinteger  applies to (thread);
end PS;
```

```
thread T
  properties
    PS::Security_Level => 5;
end T;
```

When importing the property set, the property type is translated into the following annotation type:

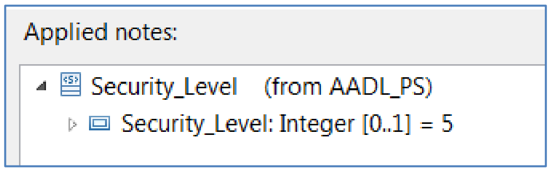
```
Security_Level ::= SEQUENCE OF {SEQUENCE {
annot_object OID, name STRING,
        information {
                Security_Level INTEGER
    }
}}
AADL-Thread ::= {
  {Security_Level F 0 1}
}
```

SCADE Architect dynamically adjusts the interface to enable addition of annotations of kind Security_Level to only objects of kind Thread.


Applicable notes:

Notes	Information
Security_Level	AADL_PS::Security_Level

The import attaches the following annotation to Thread "T".


Applied notes:

```
◢ Security_Level    (from AADL_PS)
    ▷ Security_Level: Integer [0..1] = 5
```

3 Synchronization SCADE AADL – SCADE Suite

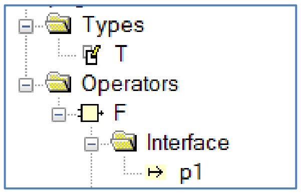
The AADL modeler provides a seamless path to SCADE Suite for the development of software components that benefit from the qualified tool chain, code generation and tests. Automated synchronization between SCADE AADL and SCADE Suite can be used in a top-down, a bottom-up or a round-trip workflow, as illustrated in Figure 1.

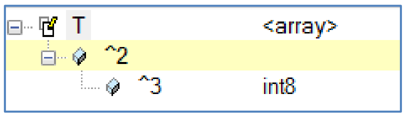
In a top-down workflow, the synchronization automatically creates a skeleton SCADE Suite model for a given AADL Thread or Subprogram, together with all corresponding data types ready to be used for implementation in SCADE Suite. In a bottom-up workflow, the synchronization can be used to initialize an AADL model from an existing SCADE Suite component.

The objective of this synchronization with SCADE Suite is to enable the implementation of the behavior of such components in SCADE Suite, and benefit from certified code generation. The objective of the synchronization is not to simulate the AADL model in a synchronous language and the SCADE Suite model is not semantically equivalent to the AADL model.

The following example illustrates the synchronization of an AADL Subprogram “F”. The subprogram is translated into a SCADE Suite operator, together with its parameters translated into SCADE Suite input and output variables. Data types are transformed into SCADE Suite data types in accordance with the AADL Data Modeling Annex where properties are used to describe the nature of the data type, its size, etc.

```
subprogram F
  features
    p1: in parameter T;
end F;

data T
  properties
    Data_Model::Data_Representation => Array;
    Data_Model::Base_Type => (classifier (Base_Types::Integer_8));
    Data_Model::Dimension => (2, 3);
end T;
```





4 Example

The example used to illustrate SCADE AADL is a simple self-driving car proposed and detailed in [8]. It allows illustrating how system latency and safety requirements can be specified, analyzed, and the way different deployment platforms for the same software architecture design affect these properties.

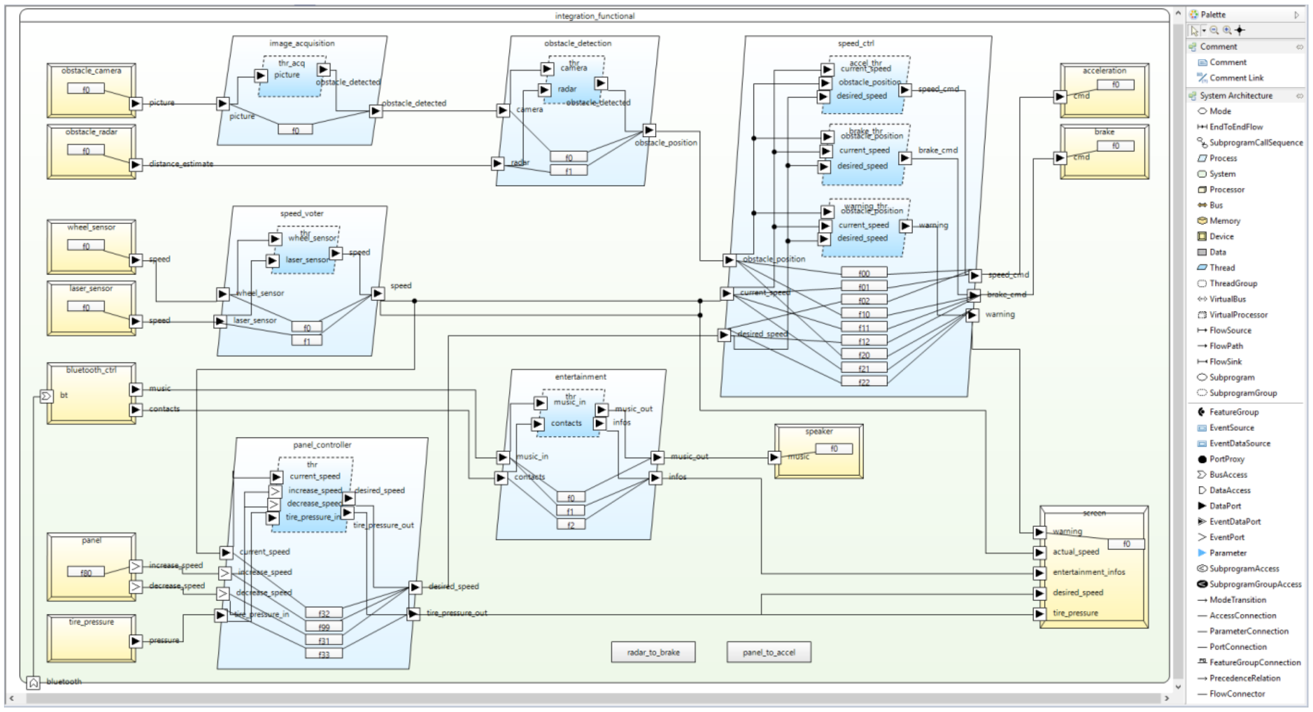


Figure 3: CarSystem model integration.functional graphical diagram

The example, provided as a set of ten AADL textual files has first been imported without any change in SCADE AADL, demonstrating the benefits of relying on a standard: models can be exchanged between AADL tools without ad'hoc modifications. Unfortunately, the standard does not prescribe any artifacts for the graphical representation, but the visual representation of each AADL component. The CarSystem graphical diagram has been redrawn from dragging and dropping each element listed in the model browser. The screenshot figure 4 shows the representation of the Devices, Processes and nested Threads and Flows in SCADE AADL.

The original example specifies two deployments of the software architecture depicted above through the AADL "extends" constructs:

```
system implementation integration.functional
  subcomponents
    acceleration  : device aadlbook::devices::acceleration;
    bluetooth_ctrl     : device aadlbook::devices::bluetooth_controller;
    brake : device aadlbook::devices::brake;
    entertainment      : process aadlbook::software::entertainment::entertainment.i;
    image_acquisition  : process aadlbook::software::image_acquisition::image_acquisition.i;
  -- cut
end integration.functional;

system implementation integration.variation1 extends integration.functional
  subcomponents
    cpu1 : processor aadlbook::platform::ecu;
    cpu2 : processor aadlbook::platform::ecu;
    can1 : bus aadlbook::platform::can;
    can2 : bus aadlbook::platform::can;
    can3 : bus aadlbook::platform::can;
  properties
    actual_processor_binding => (reference (cpu2)) applies to panel_controller, entertainment;
  -- cut
end integration.variation1;

system implementation integration.variation2 extends integration.functional
  subcomponents
    cpu : processor aadlbook::platform::ecu;
    can1 : bus aadlbook::platform::can;
    can2 : bus aadlbook::platform::can;
  -- cut
end integration.variation2;
```

As explained in 2.2, the "extends" inheritance feature is in-lined in SCADE AADL at model importation. The result is that the elements of `integration.functional` are duplicated in both integration variations; not fulfilling the modeling objective to specify one software architecture with two different deployments.

To reach this objective, the original model has been modified to rely on a simple instantiation of the software architecture `integration_functional` in each system deployment `integration_variation1` and `integration_variation2`:

```
system implementation integration.variation1
  subcomponents
    root_function: system integration_functional;
    cpu1 : processor aadlbook::platform::ecu;
    cpu2 : processor aadlbook::platform::ecu;
    can1 : bus aadlbook::platform::can;
    can2 : bus aadlbook::platform::can;
    can3 : bus aadlbook::platform::can;
  properties
    actual_processor_binding => (reference (cpu2)) applies to panel_controller, entertainment;
  -- cut
end integration.variation1;

system implementation integration.variation2
  subcomponents
    root_function: system integration_functional;
    cpu : processor aadlbook::platform::ecu;
    can1 : bus aadlbook::platform::can;
    can2 : bus aadlbook::platform::can;
  -- cut
end integration.variation2;
```

The replica mechanism allows to view, and perform directly in the IDE the bindings to Processors and Buses for each individual component of the software architecture in each system deployment variation independently. The screenshot below shows the model browser view with the replica of the software architecture.

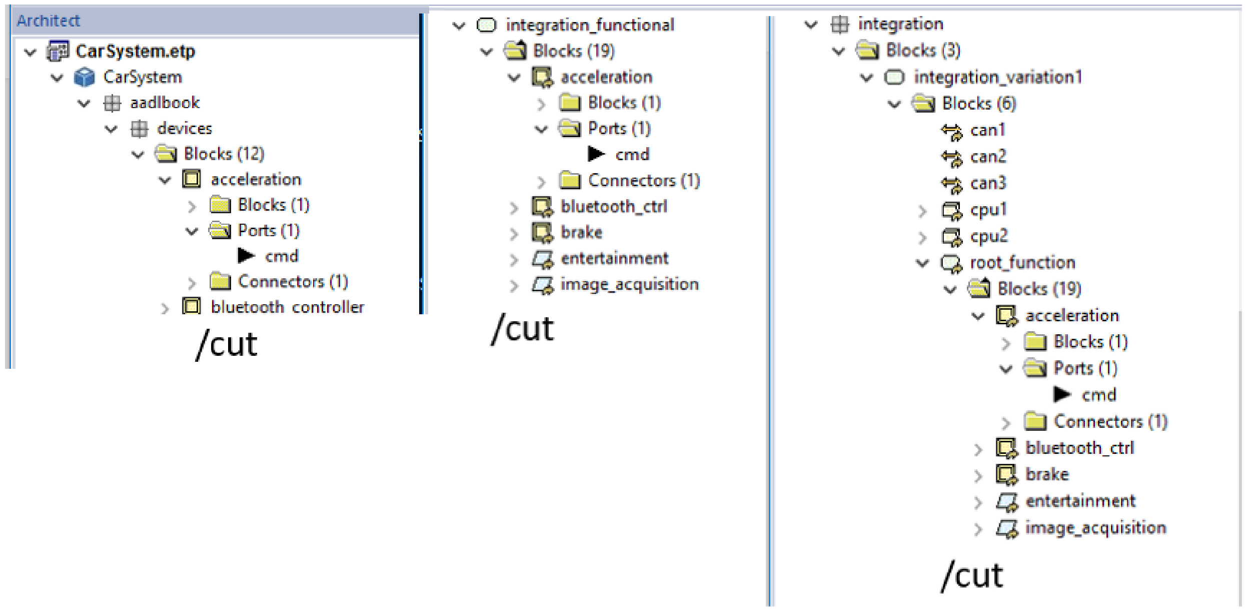


Figure 4: Extract of SCADE AADL CarSystem model browser view

5 Related works

Several transformations of AADL models have been proposed in the past decades to enable different functional and non-functional analyses and code generation capabilities. MARTE, an UML profile for modeling real-time systems, has been extended in [20] with AADL-like concepts. A similar approach has been considered in [21] to define the ExSAM profile, an AADL extension of SysML with the objective of exploiting the advantages of both languages. Unlike our approach of merging component types and implementations, two different stereotypes extending “Block” has been defined. AADL refinement and extension relations are mapped to UML realization and generalization relations, though they are semantically non-equivalent. We argue that our simplification of the DSL and handling of these relations by in-lining and resolution provides better usability and semantic point of views. Furthermore, that work does not propose any mechanism for handling property sets.

Several transformations into AADL from different models have also been considered with the goal of reusing existing AADL analyses. A transformation of MARTE design into AADL design, that focuses on thread execution and communication semantics, has been proposed in [22]. The study in [19] introduces a transformation from Capella physical architecture to AADL software architecture with the goal of implementing a development process from system definition to software design. Complementarily, our approach of transforming AADL into SCADE Suite aims to develop a seamless development process from software design to implementation and code generation.

The Stood tool for AADL [23] provides similar capabilities to SCADE AADL: import and export of AADL designs, graphical representation, and connection with the HOOD language. Extension relations can be defined using pragmas, rather than in-lined like in SCADE AADL. We argue that extension should be represented graphically by replicating elements of the super class in its children; a feature that will be implemented in future versions of SCADE AADL.

6 Conclusion and future work

This paper presented how the respective strengths of SCADE and of the AADL standard can be combined:

- AADL complements the SCADE Suite language for embedded software design.
- The powerful customization capabilities of SCADE Architect for domain specific languages is used to turn SCADE Architect into a full-fledged editor for AADL models, offering intuitive features such as automated replica for instance-based modeling. The openness of the SCADE tooling and APIs makes it possible to easily develop custom import/export utilities.

- Automated synchronization between SCADE Architect and SCADE Suite can be used in both top-down or bottom-up workflows to ensure consistency between the thread or subprogram interfaces defined in AADL for the system view, and in SCADE Suite for the software component implementation.

Besides support of the AADL standard, using the same DSL and system-software synchronization means, the SCADE Avionics Package includes support of the FACE™ Technical Standard [18] from the Open Group FACE™ Consortium and the support of an original avionics systems modeling guideline [5] detailing the way software messages carry functional data on a platform.

The three views complement each other:

- AADL focusing on systems analysis (performance, safety, …)
- FACE focusing on component reusability
- SCADE Avionics modeling focusing on ICD generation

These are supported by three distinct meta-models, but respective models represent different views of the same system. On-going work aims at reconciling and synchronizing these views in a single comprehensive framework.

7 References

1. "Architecture Analysis and Design Language (AADL), SAE standards: http://standards.sae.org/as5506/.
2. “Systems Modeling Language (SysML)”, OMG: http://www.omgsysml.org/.
3. Open Source AADL Tool Environment (OSATE), Carnegie Mellon Software Engineering Institute: https://wiki.sei.cmu.edu/aadl/index.php/Osate_2.
4. ANSYS SCADE products, http://www.ansys.com/products/embedded-software.
5. “The SAE AADL standard: A basis for model-based architecture-driven embedded systems engineering”, P. Feiler, B. Lewis, and S. Vestal. Workshop on Model-Driven Embedded Systems, May 2003.
6. “Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language (SEI Series in Software Engineering)”, Peter H. Feiler and David P. Gluch. Addison-Wesley, 2012.
7. “Challenges in validating safety-critical embedded systems”, P. Feiler. AEROTECH Congress. SAE, Nov 2009.
8. “AADL In Practice”, Julien Delange: http://www.aadl-book.com.
9. “Enforcing Software Engineering Tools Interoperability: An Example with AADL Subsets”, Vincent Gaudel, Alain Plantec, Frank Singhoff, Jérôme Hugues, Pierre Dissaux, and Jérôme Legrand. IEEE International Symposium on Rapid System Prototyping, October 2013 (Montreal, Canada).
10. “Joint Common Architecture Demonstration Shadow Architecture Centric Virtual Integration Process”, Final Technical Report, prepared for Army AMRDEC. Adventium Labs.
11. “SCADE 6: A Model Based Solution For Safety Critical Software Development”, François-Xavier Dormoy. ERTS 2008.
12. “A Conservative Extension of Synchronous Dataflow with State Machines”, J.L. Colaço, B. Pagano, M. Pouzet. EMSOFT’05.
13. “Benefits of Model Based System Engineering for Avionics Systems”, Thierry Le Sergent, François-Xavier Dormoy, Alain Le Guennec. ERTS 2016.
14. “Data Based System Engineering: ICDs management with SysML”, Thierry Le Sergent, Alain Le Guennec. ERTS 2014.
15. “Efficiently and Effectively Creating Software Components that Align with the FACE™ Technical Standard – The SCADE Experience”, Guilherme Goretkin, Bernard Dion, Thierry Le Sergent, Alain Le Guennec. US Air Force FACE TIM Conference - March 2017.
16. “Optimized Safety-Critical Embedded Display Development with OpenGL SC”, V. Rossignol - SAE AeroTech 2011.
17. “Optimized Model-Based Verification Process to Comply with DO-178C/DO-331 Objectives”, Pierre Vincent - ESSS 2015.
18. “Future Airborne Capability Environmennt (FACE)”. http://www.opengroup.org/face.
19. “Model Driven Engineering with Capella and AADL”, Bassem Ouni, Pierre Gaufillet, Eric Jenn, Jérôme Hugues. ERTSS 2016, January 2016.
20. “MARTE: Also an UML Profile for Modeling AADL Applications”, M. Faugere, T. Bourbeau, R. de Simone and S. Gérard. ICECCS 2017.
21. “Extending SysML with AADL Concepts for Comprehensive System Architecture Modeling”, R. Behjati, T. Yue, S. Nejati, L. Briand and B. Selic. Springer 2011, vol. 6689.
22. “From UML to AADL: an Explicit Execution Semantics Modelling with MARTE”. M. Burn, M. Faugère, J. Delatour and T. Vergnaud. ERTSS 2008.
23. “Stood 5.3 AADL User Manual”. P. Dissaux. 2011.