

AADL Guidelines To Implement and Verify Applications

Thomas VERGNAUD
thomas.vergnaud@cnes.fr

21st March 2007

Abstract

One of the purposes of the Architecture Analysis & Design Language is to enable the production of source code or formal models from architecture descriptions. Therefore, one unique semantics must be define and associated with AADL constructions, so that descriptions can effectively be turned into executable applications, and formal models exactly correspond to these applications. This document outlines such a semantics, and indicates how the standard annexes could be connected with it.

1 Current State of Affair

Analysis and code generation from architectural descriptions require the definition of rules concerning the way to design architectures. In order to efficiently process AADL descriptions, we need to define the complete semantics of these descriptions. This should cover two main aspects:

- component behavior;
- data semantics.

1.1 The Main Standard

The AADL standard provides some of such rules, by defining precise semantics for the components and the behavior of the threads. However, it is not sufficient to completely cover all the modeling semantics.

1.2 The Behavior Annex

The behavioral annex provides some additional constructions to describe the communications within AADL threads. It can be used to indicate when data should be read or sent, and under which conditions.

It also provides constructions to describe sequences of operations within threads and subprograms.

Besides these behavior constructions, the annex specifies how to associate semantics to AADL data types. This done by the definition of particular data components, e.g. *Integer*.

1.3 The Programming Language Annex

The programming language annex defines a property set to specify the semantics of data components.

It relies on the thread execution semantics described in the main standard by state diagrams to support the execution of applications. However, it does not describe the management of communications: they are supposed to be handled by source code—either generated or written by hand.

1.4 Conclusion

The different parts of the AADL standard deal with architecture semantics. Some notions are described (e.g. thread lifecycle), some others have several—and different—definitions (e.g. data types), others are not covered at all (e.g. inter-thread communications).

Hence there is need for guidelines to help users in designing architectures. These guidelines should describe “meaningful” architectural constructions and the associated semantics, so that everyone—architects and toolmakers—would rely on a unique interpretation of the AADL descriptions.

Such guidelines would be meant to help users write AADL when aiming at verifying or generating applications. High level architectures described in early stages of architectural design would not have to follow them. Hence, system designers would still be able to easily describe architectures using the AADL.

2 Specifying Data Types

There are two ways of specifying the semantics of data components: either by using properties or by defining a set of components with particular names (e.g. `Integer`).

Using properties is the better approach, since this provides more flexibility, especially when dealing with component extension: the properties are kept in the extending component, while the name changes. In addition, properties are more convenient regarding the characterization of components. They can be combined to provide fine description of the data types:

- data semantics: integer, boolean, float, etc.
- data size
- data structure: simple type, array, list, etc.

Hence the semantics of the data components should be specified using properties, as it is already described in the AADL wiki. This does not prevent, however, from defining AADL packages with pre-defined data component declarations that could be used within descriptions.

3 Specifying Semantics on Execution Flows

In order to help users in writing AADL descriptions that could be processed for code generation or analysis, it is required to define a common set of legal syntactic constructions and associated semantics.

3.1 Lifecycle of the Threads

The execution scheme of the AADL threads is described in the standard. It describes the different steps of a thread lifecycle. It is not connected to syntactic constructions: the scheme always applies to all kinds of threads, whatever their call sequences and features.

3.2 Communications Between Components

In order to allow pertinent architectural descriptions, precise semantics should be defined regarding the execution flows within AADL components. These execution flows mainly consist of actions performed inside components and communications (i.e. actions performed outside the components).

Communication functionalities are modeled by ports, subprograms as features and data accesses. The semantics of these communication means must be defined so that users could model architectures precisely. That is, legal AADL constructions must be listed, and associated with their counterpart in term of actual implementation. We can identify three categories of communications:

- ports model message passing;
- subprogram as features model remote procedure calls or remote method invocations;
- data accesses model shared memory.

Subprograms model remote procedure calls if they are features of threads; they model method invocation if they are features of data components. In this later case, the AADL data components correspond to classes, as defined in the object oriented programming paradigm.

Table 1 summarizes the interpretation of the AADL thread interfaces.

3.3 Communication Instant

One of the main issues of the modeling of communications is to be able to state *when* the communications actually occur. In order to ease the understanding of the modeling process in AADL, it is preferable to apply the same, general rules for the communication instants, whatever the communication paradigm is considered. This concerns message passing and share memory; communications of remote procedure calls are performed according to the call sequences involved.

Two semantic schemes can be chosen to specify when the communications occur:

- as soon as the data are transmitted from the called subprogram to the feature, through the associated connection;
- at the end of the thread execution cycle.

From an analysis point of view, it is easier to consider that data are sent or updated at the end of thread cycles. Data of out event data ports are stored in queues and sent at the end of the cycle. Shared data are only updated at the end of the cycle.

From an execution time point of view, the most efficient semantics is to state that communications occur *as soon as possible*. Data are sent to other threads as soon as they are generated by the AADL subprograms they are connected to; they are read when a subprogram is called with the

message passing	<pre> thread source_thread features msg : out event data port a_data; end source_thread; thread destination_thread features msg : in event data port a_data; end destination_thread; </pre>
remote procedure call	<pre> thread client_thread features rpc : requires subprogram access a_subprogram; end client_thread; thread server_thread features rpc : subprogram a_subprogram; end server_thread; </pre>
shared memory	<pre> thread a_thread features mem : requires data access a_data; end a_thread; </pre>

Table 1: Communication paradigms in AADL

corresponding parameters. Shared data are accessed whenever subprograms use them as parameters. This approach is especially relevant to model communications of background threads, that may not end.

Both approaches are interesting. They correspond to two possible configurations for the underlying runtime. Users should be permitted to use both, depending on the modeling context. The selection of the communication synchronism is to be done through a property that would apply to features, threads, processes, etc.

```

property set runtime is
  communication_policy : inherit enumeration (immediate, deferred) =>
    deferred applies to (port, data, thread, thread group, process,
      system);
end runtime

```

This property impacts the semantics of the AADL models, as we can see in the following example:

```

thread a_thread
features
  out1 : out event data port a_data
    {runtime::communication_policy => immediate;};
  out2 : out event data port a_data
    {runtime::communication_policy => deferred;};
  in1 : in event data port a_data;
end a_thread;

subprogram a_subprogram

```

```

features
  in1 : in parameter a_data;
  out1 : out parameter a_data;
end a_subprogram;

thread implementation a_thread.i
calls {
  call1 : subprogram a_subprogram;
  call2 : subprogram a_subprogram;}
connections
  in_cnx1 : parameter in1 -> call1.in1;
  in_cnx2 : parameter in1 -> call2.in1;
  out_cnx1 : parameter call1.out1 -> out1;
  out_cnx2 : parameter call1.out1 -> out2;
  out_cnx3 : parameter call2.out1 -> out1;
  out_cnx4 : parameter call2.out1 -> out2;
end a_thread.i;

```

Listing 1: Immediate and deferred communications

During each execution cycle, the thread expects two incoming messages from `in1`; the first is passed to `call1`, and the second to `call2`. The thread sends three data messages: The first two correspond to immediate emissions of connections `out_cnx1` and `out_cnx3` by the feature `out1`; the third one is the deferred emission by the feature `out2`, at the end of the thread's execution.

3.4 Data Management in Execution Flows

Communication instants are related to connections between parameters of subprogram calls and thread features. In addition to these constructions, data subcomponents have to be considered; they model local variables, or buffers.

Data subcomponents connected to ports or shared data correspond to particular situations, since they are not driven by any call sequence. The values of all data subcomponents connected to ports or shared data should be implicitly set when the thread is triggered, and be propagated into the out features at the end of the thread execution. Data readings and sendings is to be done in the same order as the connection declarations.

Such constructions are mandatory to model intermediate variables. If we consider the listing 1, we can see that the first message received on `in1` is always passed to `call1`. What if we want to pass it to `call2` instead? The use of intermediate data subcomponent allows such a manipulation, as we can see in the following model:

```

thread implementation a_thread.i
subcomponents
  buffer : data a_data;
calls {
  call1 : subprogram a_subprogram;
  call2 : subprogram a_subprogram;}
connections
  in_cnx1 : parameter in1 -> buffer;
  local_cnx1 : parameter buffer -> call2.in1;
  in_cnx2 : parameter in1 -> call1.in1;
  out_cnx1 : parameter call1.out1 -> out1;
  out_cnx2 : parameter call1.out1 -> out2;

```

```

out_cnx3 : parameter call2.out1 -> out1;
out_cnx4 : parameter call2.out1 -> out2;
end a_thread.i;

```

Listing 2: Communications with data subcomponents

In this situation, the local data first gets the received messages. Therefore, `buffer` gets the first incoming message. The call sequence is then executed, and `call1` gets the second message. The data in `buffer` is then passed to `call2`.

In the case of shared access to data components, it is mandatory to specify the locking policy for data manipulation. This implies the specification of critical sections in AADL call sequences. Such critical sections have to be easy to specify; hence they should apply to subprogram calls. Therefore, a property, called `atomic`, should be introduced. Atomic subprogram calls lock *all* the shared data they access. This way, it is not possible to manage fine-grain locking policy; however, atomic subprograms provide sufficient flexibility and remains simple enough, so that users could manipulate them easily.

4 Connection with Standard Annexes and Standard Properties

Call sequences describe linear behavior within threads or subprograms. Hence, it is not possible to model execution control, such as 'if' statements. Multiple call sequences should be allowed within threads and subprograms; they would describe all the potential execution scenarios of the component.

The coordination of the call sequences would then be described either by source code implementation, specified using AADL properties, or with the behavioral annex.

4.1 Connection with the Programming Language Annex

The current version of the programming language annex provides some API elements to control the execution of the AADL threads. However, it does not deal with communications.

The annex should be completed with the specification of an API to manipulate inputs and outputs, as well as locking data accesses. The programming language annex would then define a standard API that all AADL runtimes would have to implement. This API would contain primitives to lock and unlock accesses to shared data:

- `get_resource`
- `release_resource`

and also provide functions to manipulate (event) data ports:

- `get_data`
- `read_data`
- `send_data`
- `send_event`

Unlike `get_data`, `read_data` does not extract the data from the incoming queue, thus allowing to simply read values without “consuming” them. This can be useful to select the call sequence to execute according to incoming values. `read_data` and `get_data` correspond to the same action if they apply to data ports. `send_data` and `send_event` let the application send data or events on (event) data ports. The data or event will be actually sent by the runtime according to its communication policy (described in section 3.3).

The API to read and write the data subcomponents or shared data would consist of providing them as variables. Call sequences are to be translated in procedures or functions named after the AADL entities.

Mappings should be defined to describe the translation of AADL constructions into runtime API, and to define the exact syntax of the API primitives for each programming language. Such mappings would typically correspond to IDL mappings for OMG CORBA. It would then be possible for users to write source code that would control the execution of the call sequences and perform inputs and outputs. The execution of the threads in themselves would remain under total control of the runtime, configured from the AADL description.

The purpose of defining such an API and mappings is to provide a dual approach to implement applications in AADL: either by generating code from AADL constructions (i.e. call sequences) or by allowing users to directly write source code. This would allow users to rapidly write application prototypes by simply associating code to AADL threads, or use a more strict design approach with call sequences and subprograms, coordinated by source code. The source code provided by the user would then be specified by the standard property `Compute_Entrypoint`.

4.2 Connection with the Behavior Annex

The AADL constructions provide simple execution semantics: subprograms in call sequences fetch data from ports or shared memory and execute; threads then send data to other entities. Though such a standard behavior is likely to fit most situations, users may want to have finer control on the execution flow and the inputs/outputs. For example, users may want to describe multiple executions of a given call sequence without having to directly provide source code. They may also want a complete control on locks of data accesses.

Providing such functionalities using the plain AADL syntax would result in complex constructions that would be difficult to understand. The behavior annex can provide fine grain control on AADL constructions, the same way source code would do; it is more formal, though, and thus facilitates execution analysis.

Behavior descriptions made using the annex could then be translated into source code that would rely on the API defined by the programming language.

5 Classification of AADL Properties

The current version of the AADL standard provides a large set of properties. Since they are all defined in a single set, it may be difficult for users to easily identify which ones are relevant for a given modeling activity.

Hence, it would be interesting to split the property set `AADL_Properties` and create several dedicated property sets that would gather all the properties related to a given topic. It would facilitate the creation of new property sets for specific domains, e.g. connection with languages for formal verification.

In the following subsections, we classify the different standard properties into several categories, corresponding to different activities related to AADL descriptions.

5.1 Runtime Configuration & Architecture Deployment

One of the primary purposes of the AADL is to describe executable systems. To do so, properties are defined to specify periods, bindings, etc. Such information can be organized in two property sets: one to describe the deployment of the software components on the platform components, the other one to configure the runtime.

Deployment properties mainly deal with bindings of processes, connections, etc. They also cover the specifications of protocols. The properties of this set are:

- Allowed_Memory_Binding
- Allowed_Memory_Binding_Class
- Actual_Memory_Binding
- Available_Memory_Binding
- Allowed_Processor_Binding
- Allowed_Processor_Binding_Class
- Actual_Processor_Binding
- Available_Processor_Binding
- Allowed_Subprogram_Call
- Allowed_Subprogram_Call_Binding
- Actual_Subprogram_Call
- Actual_Subprogram_Call_Binding
- Server_Subprogram_Call_Binding
- Allowed_Connection_Binding
- Allowed_Connection_Binding_Class
- Actual_Connection_Binding
- Allowed_Connection_Protocol
- Aggregate_Data_Port
- Allowed_Access_Protocol
- Connection_Protocol
- Not_Collocated
- Required_Connection

The properties related to the configuration of the runtime cover aspects such as dispatch and queue protocols, thread priorities, periods, release time, etc. They describe all the parameters of the runtime that executes the applications. The properties of this set are:

- Active_Thread_Handling_Protocol
- Active_Thread_Queue_Handling_Protocol
- Allowed_Dispatch_Protocol
- Concurrency_Control_Protocol
- Dequeue_Protocol
- Device_Dispatch_Protocol
- Dispatch_Protocol
- Memory_Protocol
- Overflow_Handling_Protocol
- Period
- Priority
- Provided_Access
- Queue_Processing_Protocol
- Queue_Size
- Required_Access
- Runtime_Protection
- Scheduling_Protocol
- Synchronized_Component
- Thread_Limit
- Urgency
- Offset
- Atomic
- Communication_Policy

A property to specify thread priorities is to be added to the existing standard properties. In addition, a property to specify the release time of thread must be added, to indicate that at each period, threads may start after a given delay. This is useful in case of communications between threads.

5.2 Time & Memory Analysis

The runtime properties are used to configure the execution of the applications. Yet, they are not sufficient to address analysis and simulation. Other sets gather the required properties to characterize the dimensions—both temporal and spatial—of the architecture.

Properties for time analysis cover all the execution times. The properties of this set are:

- Activate_Deadline
- Activate_Execution_Time
- Actual_Latency
- Allowed_Period
- Client_Subprogram_Execution_Time
- Clock_Jitter
- Clock_Period
- Clock_Period_Range
- Compute_Deadline
- Compute_Execution_Time
- Deactivate_Deadline
- Deactivate_Execution_Time
- Deadline
- Expected_Latency
- Finalize_Deadline
- Finalize_Execution_Time
- Initialize_Deadline
- Initialize_Execution_Time
- Latency
- Load_Deadline
- Load_Time
- Process_Swap_Execution_Time
- Propagation_Delay
- Read_Time
- Recover_Deadline

- Recover_Execution_Time
- Startup_Deadline
- Subprogram_Execution_Time
- Thread_Swap_Execution_Time
- Transmission_Time
- Write_Time

Memory analysis is done using properties that specify code size, stack size, etc. The properties of this set are:

- Actual_Throughput
- Allowed_Message_Size
- Assign_Time
- Assign_Byte_Time
- Assign_Fixed_Time
- Base_Address
- Device_Register_Address
- Expected_Throughput
- Source_Code_Size
- Source_Data_Size
- Source_Heap_Size
- Source_Stack_Size
- Throughput
- Word_Count
- Word_Size
- Word_Space

Time analysis properties are to be used with runtime and deployment properties to perform scheduling analysis. Memory analysis properties should be used with deployment properties to perform memory footprint verification.

Both sets provide properties to *characterize* the components. Unlike the runtime property set, they do not deal with configuration. Hence, these properties should be defined in the component declarations.

5.3 Semantics for Data Types

In order to perform code generation or behavior analysis, semantics of data components must be defined. The corresponding property set—taken from the SAE wiki—provide properties to describe data types. The properties of this set are:

- Data_Type
- Integer_Range
- Real_Range
- Enum_Values
- Constant_Integer
- Constant_Real
- Constant_Boolean
- Constant_String
- Data_Structure
- Dimension
- Data_Classifier

These properties can be used in conjunction with these in the memory analysis set to specify both data semantics (integer, float, etc.) and size (32 bits, etc.)

5.4 Connection with Programming Languages

The data type annex allows the specification of data semantics, that is, the static part of the architecture. The dynamic part consists of the description of the algorithms that control the application components—threads and subprograms. The properties of this set are:

- Activate_Entrypoint
- Compute_Entrypoint
- Deactivate_Entrypoint
- Finalize_Entrypoint
- Initialize_Entrypoint
- Recover_Entrypoint
- Hardware_Description_Source_Text
- Hardware_Source_Language
- Source_Language

- Source_Name
- Source_Text
- Supported_Source_Language
- Type_Source_Name

the source code property set provides different properties to associate source code with the components. Other similar property sets could be defined for various formalisms (Petri Nets, etc.).

6 Interpretation of AADL Modes

In order to have a complete set of guidelines to describe AADL architectures, it is mandatory to specify how AADL modes are translated into actual applications. AADL modes correspond to architectural configurations; mode switches correspond to configuration switches.

Such configuration switches have to be managed by the AADL runtime. Therefore, they can only be interpreted at the level of the AADL threads and processes. As a consequence, no mode should be used within subprograms.

AADL modes are to be interpreted as state machines implemented within the runtime of each application node, i.e. each AADL process and the associated AADL threads, or the devices. Since AADL systems are immaterial, the configurations associated with their modes are implicitly managed by the contained processes, devices and threads.

7 Conclusion

The AADL syntax allows for the description of many architectural configurations. The standard and the existing annexes do not currently associate precise semantics to every construction.

In order to create a consistent design process with the AADL, it is mandatory to define which constructions are legal regarding the production of actual systems, and which are only meant to be used in the preliminary stages of modeling.

Such semantic definition should not be part of the AADL standard, since it deals with a specific use of the AADL—the production of applications. The existing annexes that are related to this usage of the AADL—the behavior annex and the programming language annex—do not completely cover the modeling semantic issues.

Therefore, there is a need for a *modeling* annex that would describe how to write AADL in order to describe architectures that can be handled by code generators or formal verification tools. This annex would describe the AADL constructions we mentioned in this document. The programming language annex would describe the standard API that all AADL runtimes should provide. Mappings between AADL and programming languages should be defined for all the constructions described in the modeling annex. The behavior annex would specify constructions that refine the ones that are described in the modeling annex. It would specify what source code must be generated from behavior description; such a source code would comply with the programming language annex.