SAE Aerospace
An SAE International Group

AEROSPACE STANDARD

SAE AS5506A

2015-01-27

ARCHITECTURE ANALYSIS & DESIGN LANGUAGE (AADL) V2 CODE GENERATION ANNEX DOCUMENT

This Architecture Analysis & Design Language (AADL) standard document was prepared by the SAE AS-2C Architecture Description Language Subcommittee, Embedded Computing Systems Committee, Aerospace Avionics Systems Division.

SAE Technical Standards Board Rules provide that: "This report is published by SAE to advance the state of technical and engineering sciences. The use of this report is entirely voluntary, and its applicability and suitability for any particular use, including any patent infringement arising therefrom, is the sole responsibility of the user."

SAE reviews each technical report at least every five years at which time it may be reaffirmed, revised, or cancelled. SAE invites your written comments and suggestions.

Copyright © 2004 SAE International

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of SAE.

TO PLACE A DOCUMENT ORDER: **Tel: 877-606-7323 (inside USA and Canada)**
Tel: 724-776-4970 (outside USA)
Fax: 724-776-0790
Email: custsvc@sae.org

SAE WEB ADDRESS: **http://www.sae.org**

TABLE OF CONTENTS

Code Generation

Normative

A.1 Scope

(1) The AADL allows the end user to model Distributed Real-Time and Embedded systems. Such systems would run atop an AADL runtime, eventually built over a complete runtime system (Ada, Java), or a real-time operating system or RTOS (with a C/POSIX API, etc.). Services provided by the AADL runtime can be manipulated by model entities (like subprogram implementation). In this context, guidelines and specifications are required to build the code referenced by such entities.

(2) The purpose of this annex is to enable the mapping of AADL model entities onto programming languages, and the mapping of the AADL runtime services onto these languages.

(3) This document targets the Ada and C languages. Other languages are expected to be regular and to follow similar considerations. We retained Ada 2012 and C11 language specifications, as they are the latest revisions of these two ISO standards and use their corresponding short name, without mention of the actual version being used.

(4) This document only addresses how the user can insert its own code to interact with the AADL runtime and elements from the models. It details how to map AADL identifiers (section A.3), packages (section A.4), data types (section A.5), subprograms (section A.6), AADL runtime services (section A.6.3)

(5) It addresses neither the actual implementation of the AADL runtime nor the use of actual underlying operating system services made by the AADL runtime.

(6) Different methods of implementation can be contemplated for supporting AADL concepts at source-code level. Among the solutions, one can pick: “manual coding” where the designer does AADL-to-code translation manually; “API based”, where the designer manipulates an API representing the AADL runtime; “compilation based”, where part of the code is generated from the AADL model. In this document, we aim at being generic to let implementation decide how to combine code and the AADL runtime, and perform specific optimizations.

(7) The objective of this document is to provide a portable definition of coding guidelines to map AADL concepts and model entities onto a programming language, so that source code can be included and manipulated by different tools.

A.2 Structure of the document

(8) The document is structured as follows: for each AADL concept, a mapping is proposed for both the Ada and C languages.

(9) The definition of the following rules obeys to the following rationale:

a. **Naming/Mapping**: one needs to name the AADL entities of its model in the target programming language. This is presented in section A.3 that lists naming conventions for mapping AADL identifiers, and in section A.4 that lists how to map an AADL hierarchy of packages.

b. **Data consistency**: one needs to manipulate data types. This document relies on the Data modeling annex to provide support for data types. Section A.5 illustrates how to map AADL data types definition onto the target language.

c. **Functional routines execution**: one needs to execute subprograms. Section A.6 illustrates how AADL subprograms are mapped.

d. **Interaction within distributed environments**: one needs to interact with other entities. Section A.6.3 shows how to use AADL runtime services.

(10) This Annex document also completes for C and Ada the mappings of AADL runtime services defined in the AADL core document, or data types defined using the “Data Modeling Annex Document; and provides language-specific packages and files.

A.3 Naming conventions

(11) Depending on the mapping requirements, one need to map either component type/implementation onto language constructs, e.g. for mapping AADL data component types onto their corresponding data type definition in source code, or to map component classifier onto corresponding references (e.g. variables, etc). This is discussed in the next sections.

(12) AADL identifiers (such as name of subcomponents, component types or implementations) are converted into language identifiers. A method of implementation should use identifiers from the declarative model whenever possible, as these represent the user view on the model.

(13) If the target language supports a namespace mechanism, it is recommended to use this mechanism to avoid the use of the fully qualified name.

(14) To avoid name clashing with language-specific keywords or to follow language-specific syntactic rules, the following rules are defined:

A.3.1 Ada mapping

(15) To respect the Ada rules of identifier construction (AARM, 2.3 (2)), some characters that may appear in AADL entity names must be replaced. Here is a list:

- "." must be replaced by underscores "_".
- Two consecutive underscores "__" must be replaced by "_U_".

(16) The string "AADL_" must prefix each identifier that clashes with an Ada keyword. If the concatenation provokes a clashing between identifiers, we add another concatenation of the same string until there is no longer name clashing.

(17) AADL string properties referencing to an Ada source code element (like `Compute_Entrypoint_Source_Text`) should be fully qualified Ada name.

A.3.2 C mapping

(18) To respect the C rules for identifiers (C RM 6.4.2.1), all identifiers derived from AADL identifiers are turned to lowercase.

(19) The string "`aadl_`" must prefix each identifier that clashes with a C keyword. If the concatenation provokes a clashing between identifiers, we add another concatenation of the same string until there is no longer name clashing.

(20) AADL string properties referencing to a C source code entity (like `Compute_Entrypoint_Source_Text`) should be the name of the corresponding entity.

(21) C requires special rules for mapping some constructs, these are listed after: enumeration type in section A.5.

(22) An implementation method is allowed to define additional rules in case of symbol conflicting with the ones defined by the underlying operating systems.

(23) Note: Such clash may occur if the implementation of the AADL runtime requires the use of C function from the C standard library (e.g. `read`), and the user defines also a subprogram called "read".

A.4 Mapping of AADL packages

A.4.1 Ada mapping

(24) AADL packages and Ada packages differ in visibility rules. There are no nested packages in AADL, whereas Ada favor them. To avoid introducing unwanted visibility to parent package, AADL packages are mapped onto Ada packages in a flat hierarchy.

AADL hierarchy is mapped onto a single name, where dots are replaced with one consecutive underscore. All identifiers are turned to lower case. For instance, the identifier `Foo::Bar` is mapped onto `foo_bar` and `Foo::Bar::Baz` onto `foo_bar_baz`.

A.4.2 C mapping

(25) AADL packages have no equivalent in C. AADL hierarchy is mapped onto a single name, where dots are replaced with two consecutive underscores. All identifiers are turned to lower case. For instance, the identifier `Foo::Bar` is mapped onto `foo__bar` and `Foo::Bar::Baz` onto `foo__bar__baz`.

A.5 Mapping data types components

(26) The AADL proposes several ways to define data types: either one uses the Data Modeling annex, or relies on AADL properties applied to data component types or implementation. See discussion in the Data Modeling Annex document, part of AS5506A document.

A.5.1 Mapping of Data_Model

(27) The Data Modeling annex of AADLv2 standard defines properties so that the system designer can define its own data. These properties are defined in the AADL property set `Data_Model`.

(28) An implementation method may provide a tool to map AADL data component type definitions onto the destination language.

(29) An implementation method may restrict the set of data types supported by the runtime to reflect actual hardware limitations (e.g. no floating point) or coding restrictions (e.g. no types of unbounded size).

A.5.2 Mapping of Base_Types

(30) The Data Modeling annex defines base types to be manipulated by the end user, in the AADL package `Base_Types`.

(31) The implementation method shall provide an implementation of the AADL package `Base_Types` in the destination language, or mechanism to generate it. The user may directly reference entities from this implementation in its own source code.

(32) An implementation method may restrict the content of `Base_Types` to the types whose properties are supported in the `Data_Model` property set.

(33) An Ada package mapped from `Base_Types` is provided for Ada in Annex 1.1.

(34) A C header file mapped from `Base_Types` is provided for C in Annex 1.2.

A.5.3 Mapping of data components

(35) Each data component denoting a basic type (Boolean, Character, Enum, Float, Fixed, Integer, String) is mapped to an equivalent data type in the target language. The name of the type follows rules on identifiers mapping. Its type derives from the properties of the AADL component.

(36) Mapping of the Based_Types package is provided in Annex 1

(37) Complex data components (Array, Struct, Union) are mapped to the corresponding construct in the target language. The name of the type follows rules on identifier mapping. Its type derives from the properties of the AADL component.

a. The following array definition

```
data One_Dimension_Array
 properties
   Data_Model::Data_Representation => Array;
   Data_Model::Base_Type => (classifier
(Base_Types::Integer));
   Data_Model::Dimension => (42);
 end One_Dimension_Array;
```

would be mapped to

```
-- Ada
type One_Dimension_Array
     is array (1 .. 42)of Base_Types.Integer_Type;
/* C */
typedef base_types__integer_types one_dimension_array [42];
```

b. The following struct definition

```
data A_Struct1
properties
   Data_Model::Data_Representation => Struct;
   Data_Model::Base_Type =>
```

```
      (classifier (Base_Types::Float),
       classifier (Base_Types::Character));
          Data_Model::Element_Names => ("f1", "c2");
        end A_Struct1;
```

would be mapped to

```
      -- Ada
      type A_Struct1 is record
        F1 : Base_Types.Float;
        C2 : Base_Types.Character;
      end record;
      /* C */
      typedef struct{
        base_types__float f1;
        base_types__character c2;
      } a_struct1;
```

c. The following union definition

```
       data A_Union1
        properties
          Data_Model::Data_Representation => Union;
          Data_Model::Base_Type =>
            (classifier    (Base_Types::Float),        classifier
(Base_Types::Character));
          Data_Model::Element_Names => ("f1", "f2");
        end A_Union1;
```

would be mapped to

```
        -- Ada
```

```
  type A_Union_1_Flag  is (F1_Flag, F2_Flag);

  type A_Union1 (A : A_Union_1_Flag) is record
     case A is
        when F1_Flag =>
           F1 : Float;
        when F2_Flag =>
           F2 : Character;
     end case;
  end record;
/* C */
typedef union {
     base_types__float f1;
     base_types__character f2;
} a_union1;
```

(38) C enum requires a special rule for mapping. C does not allow multiple use of the same enumerator. In this case, the name of the enumerator is prefixed by the name of the AADL data components type.

a. The following definition

```
  data An_Enum
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("foo", "bar");
end An_Enum;
```

would be mapped to

```
-- Ada
  type An_Enum is (Foo, Bar);

/* C */
typedef enum { an_enum_foo, an_enum_bar } an_enum;
```

(39) Complex data components that reference other data components are mapped onto data structures of the target language: Ada record types (resp. C structure definitions). Each field defining identifier is mapped from the subcomponent name given in the data component implementation with the rules on identifiers mapping. The type of the field is the Ada (resp. C) type mapped from the data corresponding component. The name of the type follows rules on identifiers mapping. Its type derives from the properties of the AADL component.

A.5.4 Other cases

(40) The system designer may decide not to use the Data Modeling annex. In this case, he may use the `Source_Language`, `Type_Source_Name` and `Source_Text` properties to specify the definition of its data type in the target language. In this case, the user may reference directly this type, without further refinements.

a. The following definition

```
data C_Type
properties
  Source_Language => (C);
  Source_Text => ("types.h");
  Type_Source_Name => "the_type";
end C_Type;
```

it is expected the user provides a C source file named “types.h” where a type “the_type” is defined.

A.6 Mapping of AADL subprograms

(41) AADL subprograms denote library elements that may later be used by threads. In this section, we review mapping rules for subprograms without dependencies to the AADL runtime.

(42) We illustrate how the Ada mapping may incorporate C or Ada subprograms, and how the C mapping may incorporate C subprograms.

(43) The actual implementation of the mapped subprograms depends on the nature of the corresponding AADL subprogram component. Subprogram components can be classified in many ways depending on the value of the `Source_Language`, `Source_Name` and `Source_Text` standard AADL properties, the existence or absence

of call sequences in the subprogram implementation. There are three kinds of subprogram components: empty subprograms, opaque subprograms and pure call sequence subprograms.

A.6.1 Ada mapping

(44) In this section, we discuss the mapping of AADL subprograms onto an Ada AADL runtime. AADL subprograms can be either Ada code, but also C or other formalisms.

(45) AADL subprograms are mapped onto Ada procedures. In case of data-owned subprograms, they are managed in the related generated package. The parameters of the procedure are mapped from the subprogram features with respect to the following rules:

- The name of the subprogram is derived from the name of the subprogram classifier
- The parameter name is mapped from the parameter feature name
- The parameter type is mapped from the parameter feature data type
- The parameter orientation is the same as the feature orientation (“in”, “out” or “in out”).
- The order of parameters is the same as the corresponding AADL subprogram component.

(46) **Empty subprograms** correspond to AADL subprogram types or implementations for which there is neither `Source_Language` nor `Source_Name` nor `Source_Text` values nor call sequences. Such kind of subprogram components is an empty placeholder for future implementation. It should evolve to one of the two other kinds in the production AADL model.

a. The AADL snippet below is an example of an empty subprogram:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
```

b. An Ada mapping for this subprogram could be:

```
procedure sp (e : in message; s : out message) is
     pragma Unreferenced (e…);
```

```
begin
   null;
end sp;
```

(47) **Opaque subprograms** are AADL subprograms for which the Source_Language property indicates the programming language of the implementation (C or Ada). The Source_Name property indicates the name of the target language subprogram implementing the AADL subprogram:

a. For Ada subprograms, the value of the Source_Name property is the fully qualified name of the subprogram (e.g. My_Package.My_Spg). If the package is stored in a file named according to the Ada compiler conventions, there is no need to give a Source_Text property for Ada subprograms. Otherwise the Source_Text property is necessary for the compiler to fetch the implementation files.

b. For C subprograms, the value of the Source_Name property is the name of the C subprogram implementing the AADL subprogram. The Source_Text is mandatory for this kind of subprogram and it must give one of the following information: the path (relative or absolute) to the C source file that contains the implementation of the subprogram; the path to one or more precompiled object file(s) that implement(s) the AADL subprogram; the path to one or more precompiled C libraries that implement(s) the AADL subprogram.

c. These properties can be used together, for example one may give the C source file that implements the AADL subprogram, an object file that contains entities used by the C file and a library that is necessary to the C sources or the objects. In this case, the code generation consists of creating a shell for the implementation code. In the case of Ada subprograms, the generated subprogram renames the implementation subprogram (using the Ada renaming facility).

d. Here is a mapping example

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
```

```
properties
   Source_Language => Ada;
   Source_Name => "Repository.Sp_Impl";
end sp.impl;
```

e. The generated code for the sp.impl component is:

```
with Repository;
-- ...
procedure sp_impl (e : in message; s : out message)
  renames Repository.Sp_Impl;
```

f. The code of the `Repository.Sp_Impl` procedure is provided by the designer and must conform to the `sp.impl` signature as defined in the architecture. The coherence between the two subprograms will be verified by the Ada compiler. The fact that the hand-written code is not inserted in the generated shell allows this code to be written in a programming language other than Ada. Thus, if the implementation code is C we have this situation:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
properties
   Source_Language => C;
   Source_Name => "implem";
   Source_Text => "code.c";
end sp.impl;
```

g. The `Source_Name` value is interpreted as the name of the C subprogram implementing the AADL subprogram.

h. Path information to actual source file can be added in `Source_Text`, or left under the control of the implementation method.

i. The generated code for the sp.impl component is:

```
procedure sp_impl (e : in message; s : out message);
pragma Import (C, sp_impl, "implem");
```

j. This approach will allow us to have a certain flexibility by separating the generated signature from the hand-written code. We can modify the AADL description without affecting the hand-written code (the signature should not be modified of course).

(48) **Pure call sequence subprograms**: in addition to the opaque approach, which consists of delegating all the subprogram body writing to the user, AADL allows to model subprogram as a pure call sequence to other subprograms. Example:

```
subprogram spA
features
   s : out parameter message;
end spA;
subprogram spB
features
   s : out parameter message;
end spB;
subprogram spC
features
   e : in parameter message;
   s : out parameter message;
end spC;
subprogram spA.impl
calls {
   call1 : subprogram spB;
   call2 : subprogram spC;};
connections
   cnx1 : parameter call1.s -> call2.e;
   cnx2 : parameter call2.s -> s;
end spA.impl;
```

a. In this case, the subprogram connects together a number of other subprograms. In addition to the call sequence, the connections clause

completes the description by specifying the connections between parameters. The pure call sequence model allows the generation of skeleton code referring to other subprograms: the calls in the call sequence correspond to Ada procedure calls and the connections between parameters correspond to the possible intermediary variables. The Ada code generated for the subprogram `spA.impl` is:

```
procedure spA_impl (s : out message) is
  cnx1 : message;
begin
  spB (cnx1);
  spC (cnx1, s);
end spA_impl;
```

b. Note that in case of pure call sequence subprograms, the AADL subprogram must contain only one call sequence. If there is more than one call sequence, it's impossible - in this case - to determine the relation between them.

A.6.2 C mapping

(49) In this section, we discuss the mapping of AADL subprograms onto a C AADL runtime. We review some options available.

(50) AADL subprograms are mapped onto C functions. In case of data-owned subprograms, they are managed in the related generated package. The parameters of the procedure are mapped from the subprogram features using the following rules:

- The parameter name is mapped from the parameter feature name previously generated using the data mapping rules in section A.5.
- The parameter type is mapped from the parameter feature data type
- The parameter orientation of the C subprogram follows the feature orientation: by copy for "in" parameters, by reference for "out" or "in out".
- The order of parameters is the same as the corresponding AADL subprogram component.

(51) **Empty subprograms** correspond to AADL subprograms for which there is neither `Source_Language` nor `Source_Name` nor `Source_Text` values nor call sequences. Such kind of subprogram components is empty placeholders for future implementation.

a. The AADL snippet below is an example of an empty subprogram:

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
```

b. An C mapping for this subprogram could be:

```
void sp (message e, message* s) {
   /* TO BE IMPLEMENTED */
}
```

(52) **Opaque subprograms** are AADL subprograms for which the Source_Language property indicates the programming language of the implementation (C or Ada). The Source_Name property indicates the name of the subprogram implementing the subprogram:

a. For C subprograms, the value of the `Source_Name` property is the name of the C subprogram implementing the AADL subprogram. The `Source_Text` is mandatory for this kind of subprogram and it must give one of the following information: the path (relative or absolute) to the C source file that contains the implementation of the subprogram; or, the path to one or more precompiled object files that implement the AADL subprogram; or, the path to one or more precompiled C library that implement the AADL subprogram.

b. These information can be used together, for example one may give the C source file that implements the AADL subprogram, an object file that contains entities used by the C file and a library that is necessary to the C sources or the objects. In this case, the code generation consists of creating a shell for the implementation code. Here is a mapping example

```
subprogram sp
features
   e : in parameter message;
   s : out parameter message;
end sp;
subprogram implementation sp.impl
```

```
properties
   Source_Language => C;
   Source_Name => "C_sp_impl";
   Source_Text => "sp.c";
end sp.impl;
```

c. The generated code for the sp.impl component is:

```
void sp_impl (message e, message* s) {
   C_sp_impl (e, s); /* Call user code */
}
```

(53) **Pure call sequence subprograms:** In addition to the opaque approach, which consist of delegating all the subprogram body writing to the user, AADL allows to model subprogram as a pure call sequence to other subprograms. Example:

```
subprogram spA
features
   s : out parameter message;
end spA;
subprogram spB
features
   s : out parameter message;
end spB;
subprogram spC
features
   e : in parameter message;
   s : out parameter message;
end spC;
subprogram spA.impl
calls {
   call1 : subprogram spB;
   call2 : subprogram spC;};
connections
   cnx1 : parameter call1.s -> call2.e;
```

```
    cnx2 : parameter call2.s -> s;
end spA.impl;
```

a. In this case, the subprogram connects together a number of other subprograms. In addition to the call sequence, the connections clause completes the description by specifying the connections between parameters. The pure sequence call model allows to generate complete code : the calls in the call sequence corresponds to C procedure calls and the connections between parameters correspond to the possible intermediary variables. The C code generated for the subprogram `spA.impl` is:

```
void spA_impl (message* s) {
   message cnx1;
   spB (&cnx1);
   spC (cnx1, &s);
}
```

b. Note that in case of pure call sequence subprograms, the AADL subprogram must contain only one call sequence. If there is more than one call sequence, it's impossible - in this case - to determine the relation between them.

A.6.3 Management of port variables

(54) Port variables represent a way to access the port of a thread or a subprogram. Those are the only place where user's code can interact with ports.

(55) To take into account multiple instances, the AADL runtime introduces a specific opaque type called an AADL context.

(56) One specific AADL context type is generated for each thread or subprogram component type or implementation. Its name is defined as follows

a. For Ada: `<component_identifier>_Context`

b. For C: `__<component_identifier>_context`

(57) This opaque type is a record whose members are the ports available in this particular context, but also access to data. The name of these members follows identifiers mapping rules defined in this annex.

(58) The Code_Generation::Convention property controls the generation of specific structures to manage port variables.

a. Legacy convention does not generate this information. It is the default value

b. AADL convention generates port contextual structure

(59) A method of implementation may infer the actual value of the Convention property based on the model information.

(60) The Code_Generation property set is present in Annex 3

A.7 Using AADL services inside subprograms

A.7.1 Using AADL ports to communicate

(61) AADL ports allow subprograms to send or receive events or data to other entities in the model. Ports are accessed through runtime services as defined in section A.9 of the standard AADLv2 document. These services are defined as AADL subprograms, but some implementation details exist, mostly in the way ports are defined.

(62) The following provides a solution to these implementation details in a way that is portable and consistent with the concept of port variables.

(63) AADL defines mechanisms to let subprograms interact with the environment through ports and access to data components. This section illustrates how to use these mechanisms

(64) One instance of the context type is passed as parameters to each user's subprograms that need to interact with ports. This parameter is used as parameter specifying the port variable to use.

a. Here is an example, using AADLv2

```
thread Operator_T
features
  Gear_Cmd: out event port;
properties
  Dispatch_Protocol  => Periodic;
  Period             => 10 Sec;
  Compute_Entrypoint_Source_Text => "On_Operator";
```

```
    source_text        => ("flight-mgmt.c");
  end Operator_T;
```

and the corresponding C implementation

```
  void on_operator (__operator_t_context *self) {
  /* … */
  __aadl_send_output (self->gear_cmd, &request);
 }
```

(65) Note: in this example, the AADL convention is implicitly enforced as the subprogram implicitly requires access to port variable, and there is no way to deduce subprogram output parameters.

(66) An AADL model may connect `in` or `out` subprogram parameters to thread ports. In this case, there is no need to use explicitly AADL runtime services. A method implementation is allowed to generate directly the corresponding code from the AADL model if the subprogram follows the Legacy convention.

a. Let us consider the following AADL model

```
subprogram Do_Ping_Spg
features
  Data_Source : out parameter Simple_Type;
properties
  Code_Generation::Convention => Legacy;
end Do_Ping_Spg;

thread P
features
  Data_Source : out event data port Simple_Type;
end P;

thread implementation P.Impl
calls
Mycalls: {
  P_Spg : subprogram Do_Ping_Spg;
```

```
};
connections
  parameter P_Spg.Data_Source -> Data_Source;
 -- Out parameter of P_Spg is passed to the port
 -- variable automatically
end P.Impl;
```

then, it is sufficient to implement Do_Ping_Spg this way

```
procedure Do_Ping_Spg (Data_Source : out Simple_Type);
```

and have the method of implementation generates the corresponding glue code to propagate the out parameter to the corresponding port variable.

Note: This allows the seamless integration of existing code in an AADL model, for instance for opaque subprograms.

A.7.2 Subprograms attached as entrypoints

(67) In some cases, the user may want to integrate source code as entrypoint attached to threads in event (data) ports using the `Compute_Entrypoint` standard property and its several derivations.

(68) The first parameter is the AADL context information, discussed in section A.6.3

(69) In the case of an `event port`, the signature of the corresponding subprogram adds no further parameter.

(70) In the case of an `event data port`, the signature has an additional unique parameter derived from the type of the port.

a. Here is an example, using AADLv2

```
thread HCI_T
features
  Stall_Warning  : in  event data port Ravenscar.Integer
    {Compute_Entrypoint_Source_Text =>
"Manager.On_Stall_Warning";};
  Engine_Failure : in  event      port
    {Compute_Entrypoint_Source_Text =>
"Manager.On_Engine_Failure";};
end HCI_T;
```

b. And the corresponding Ada and C declarations

```
-- Ada
package Manager is
   procedure On_Stall_Warning
     (Ctx: HCI_T_Context; Stall_Warning :
Ravenscar_Integer);
   procedure On_Engine_Failure (Ctx : AADL_Context);


/* C */
void on_stall_warning
     (__hci_t_context ctx, ravenscar_integer
     stall_warning);
void on engine_failure (__hci_t_context ctx);
```

A.7.3 Accessing data components

(71) Data components can support subprogram access as a way to model accessors to its internal data. Corresponding user subprograms have additional parameters that represent the internal data to interact on. See the following example:

```
subprogram Update
features
   this : requires data access POS.Impl;
end Update;
subprogram implementation Update.Impl
properties
   source_language => Ada95;
   source_name     => "Toy.Update";
end Update.Impl;


data Position_Internal_Type
properties
   Data_Model::Data_Representation => Integer;
```

```
end Position_Internal_Type;

data Position
features
   Update : provides subprogram access Update.Impl;
properties
   Concurrency_Control_Protocol => <..>;
end Position;

data implementation Position.Impl
subcomponents
   Field : data Position_Internal_Type;
properties
   Data_Model::Data_Representation => Struct;
end Position.Impl;
```

(72) This is mapped as the following in Ada

```
procedure Read (Field : in out POS_Internal_Type);
```

(73) The AADL runtime shall handle calls to this subprogram using the concurrency protocol mandated by the user. There is no need for explicit call to Get_Ressources and Release_Resources runtime services.

Note: an implementation method is allowed to support only a subset of allowed mechanisms, depending on the support from the target operating system.

A.7.4 Managing modes

(74) Mode changes are triggered through specific ports. A user subprogram may trigger one by sending an event to the corresponding port.

(75) A user subprogram may use the AADL runtime service `Current_System_Mode` to determine is current mode of operation.

Annex 1 Mapping of Base_Types

(76) The AADL `Base_Types` package proposes a list of general utility data types.

Annex 1.1. Ada mapping

(77) This package can be mapped onto Ada using the types provided by either the Standard package (types without size) or Interfaces as defined by the Ada 2012 reference manual.

(78) In order to support previous versions of Ada, an implementation method is allowed to select a different mapping that preserves the semantics of types.

```
with Interfaces;

package Base_Types is

  type AADL_Boolean is new Standard.Boolean;

  type AADL_Integer is new Standard.Integer;

  type Integer_8 is new Interfaces.Integer_8;

  type Integer_16 is new Interfaces.Integer_16;

  type Integer_32 is new Interfaces.Integer_32;

  type Integer_64 is new Interfaces.Integer_64;

  type Unsigned_8 is new Interfaces.Unsigned_8;

  type Unsigned_16 is new Interfaces.Unsigned_16;

  type Unsigned_32 is new Interfaces.Unsigned_32;

  type Unsigned_64 is new Interfaces.Unsigned_64;

  type AADL_Natural is new Standard.Integer; -- XXX incomplete range?

  type AADL_Float is new Standard.Float;

  type Float_32 is new Interfaces.IEEE_Float_32;

  type Float_64 is new Interfaces.IEEE_Float_64;

  type AADL_Character is new Standard.Character;

end Base_Types;
```

Annex 1.2. C mapping

(79) The AADL `Base_Types` package proposes a list of general utility data types. This package can be mapped onto C11 using the types provided by either `stdint.h` or `stdbool.h` standard header files.

(80) In order to support more ancient C compilers, an implementation method is allowed to select a different mapping that preserves the semantics of types, or one that is a superset of the intended type.

(81) For instance, `Base_Types::Natural` cannot be represented in C, as there is no native positive only numeric types.

```
/* C mapping of AADL package base_types */
#ifndef __BASE_TYPES_H__
#define __BASE_TYPES_H__

#include<stdint.h>
#include<stdbool.h>

typedef bool     base_types_boolean;

typedef int8_t   base_types_int8;
typedef int16_t  base_types_int16;
typedef int32_t  base_types_int32;
typedef int64_t  base_types_int64;

typedef uint8_t  base_types_uint8;
typedef uint16_t base_types_uint16;
typedef uint32_t base_types_uint32;
typedef uint64_t base_types_uint64;

typedef float    base_types_float32; /* As per IEEE 754 */
typedef double   base_types_float64;
typedef char     base_types_character;

#endif  /* __BASE_TYPES_H__ */
```

Annex 2 AADL runtime services

(82) The AADL core standard defines two sets of runtime services in section A.9. Application runtimes services declare service subprograms that can be called by the application source text directly. The second set of services is concerned with services internal to the AADL runtime, and is not the subject of this annex.

(83) We recall that the following API is available to the user:

a. `Send_Output`: explicitly cause events, event data, or data to be transmitted through outgoing ports to receiver ports.

b. `Put_Value`: allows the source text of a thread to supply a data value to a port variable.

c. `Receive_Input`: explicitly requests port input on s incoming ports to be frozen and made accessible through the port variables.

d. `Get_Value`: allows access to the current value of a port variable.

e. `Get_Count`: determine whether a new data value is available on a port variable, and in case of queued event and event data ports, how many elements are available to the thread in the queue.

f. `Next_Value`: provides access to the next queued element of a port variable as the current value. A `NoValue` exception is raised if no more values are available.

g. `Updated` allows the source text of a thread to determine whether input has been transmitted to a port since the last `Receive_Input` service call.

(84) To favor optimization, code safety and reliability, the proposed mapping does not enforce any convention on the organization of the code: AADL API function may be placed in any relevant package. This allows for multiple implementation of the AADL API, with one dedicated instance per AADL model entities. The only recommended practice is to preserve the name of the functions and of its parameters.

Annex 3 Code_Generation property set

(85) The following property set is defined to control various aspects of the code generation:

```
property set Code_Generation_Properties is

      Convention: enumeration (AADL, Legacy) => Legacy

            applies to (subprogram);

      --  Under the Legacy convention, no context information

      --  for port variables is generated.

      --  The AADL convention generates such information

      Parameter_Usage: enumeration (By_Value, By_Reference)

           applies to (data access, parameter);

      Return_Parameter: aadlboolean => false applies to (parameter);

      -- if true, out parameter is actually a return parameter
```

```
end Code_Generation_Properties;
```