



From AADL to code, and code to AADL



Data modeling and Programming language annexes

■ Objective

- Provide guidelines to integrate code in a full AADL runtime
- Code generation of part of the AADL runtime and helper functions
- Integrate existing code seamlessly

■ Two annex documents

- Data modeling annex: how to define types more precisely
- Programming language annex: how to interface with AADL runtime services
 - Note: the implementation of the AADL runtime is not addressed

■ Keep efficiency expected by users

- Implementation prototype available as part of Ocarina, for both C and Ada, and both AADLv1 and part of AADLv2



Current status

■ Both documents rely on ongoing work at Telecom ParisTech

- Since 2004 IST-ASSERT then Flex-eWare
- Integrated in Ocarina AADL toolsuite
 - C and Ada code generators, for various runtimes
- Concepts are validated and stable

■ Reviewers wanted ;)



Data types in AADLv2



AADLv2 and user types

- One can already model some types in AADL
- User types => AADL data components
- Standard properties allow you to say a few things

```
data C_Type -- An AADL data component
properties -- representing a
  Source_Language => C; -- C object
  Source_Text => ("types.h"); -- defined in "types.h"
  Type_Source_Name => "the_type"; -- whose name is "the_type"
end C_Type;
```

■ Additional properties:

- Source_Data_Size, Concurrency_Protocol (constants), Base_Address, Priority (PCP), ...



Data modeling annex: rationale

■ Complement AADLv2 standard property sets

- To define more precisely data types
- To allow for finer analysis: automatic resource dimensioning from precise definition; code generation; traceability

■ One property set + one package library

- Data_Model property set: new properties
 - Initial_Values, typical types: integers, floats, enum, union, ..
- Base_Types package: well-known types
 - Integer_8, Float_64, ...

■ Status: stable as of April 2009 meeting

■ Integrated in Ocarina 2.0w

- Tested in ASSERT and Flex-eWare (IDL3 to AADL mapping)

Data modeling annex: examples

- **2-dimension array -> automatic size computation based on processor information (padding, word size, ...)**

```
data Two_Dimensions_Array
properties
  Data_Model::Data_Representation => Array;
  Data_Model::Base_Type => (classifier (Base_Types::Integer));
  Data_Model::Dimension => (74, 75);
end Two_Dimensions_Array;
```

- **Enum + representation information -> encoding of values**

```
data An_Enum
Properties
  Data_Model::Data_Representation => Enum;
  Data_Model::Enumerators => ("foo", "bar");
  Data_Model::Representation => ("00", "11");
end An_Enum;
```



Programming language integration



Integrating source code

■ One can define

```
subprogram sp
```

```
features
```

```
  e : in parameter message;
```

```
  s : out parameter message;
```

```
properties
```

```
  Source_Language => Ada95;
```

```
  Source_Name     => "Repository.Sp_Impl";
```

```
end sp.impl;
```

■ Question: what is the associated source code ?

■ Solution: combine data modeling annex and define mapping rules → programming language annex

Mapping of data types

■ Depending on combination of properties, derive actual data type implementation

- From existing source-level data type (*Type_Source_Name*)
- From Data_Model properties

```
data Float_32 extends Float  
properties
```

```
  Data_Model::IEEE754_Precision => Simple;
```

```
  Source_Data_Size => 4 Bytes;
```

```
end Float_32;
```

```
#include<stdint.h>
```

```
#include<stdbool.h>
```

```
typedef bool    base_types_boolean;
```

```
typedef int8_t  base_types_int8;
```

Mapping of subprograms parameters

■ Define mapping of subprograms (à-la CORBA)

- Ada: use in, out, inout + reference to data types
- C: pass parameters by value or reference depending on C conventions + reference data types

```
subprogram sp
features
  e : in parameter message;
  s : out parameter message;
properties
  Source_Language => Ada95;
  Source_Name     => "Repository.Sp_Impl";
end sp.impl;

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

```
subprogram Do_Ping_Spg
features
  Data_Source : out parameter Simple_Type;
properties
  source_language => C;
  source_name     => "user_ping";
end Do_Ping_Spg;

#include <types.h>
#include <subprograms.h>
void user_do_ping_spg
  (simple_type* data_source);
```

Combining subprograms and threads

■ Parameters are connected to thread ports

```
subprogram Do_Ping_Spg
```

```
features
```

```
  Data_Source : out parameter A_Type;
```

```
properties
```

```
-- ...
```

```
end Do_Ping_Spg;
```

```
thread implementation P.Impl
```

```
calls {
```

```
  P_Spg : subprogram Do_Ping_Spg;
```

```
};
```

```
connections
```

```
  parameter P_Spg.Data_Source ->  
  Data_Source; -- ...
```

■ No need for the user to write anything special

- Generated code would take the output value and send it using AADL runtime services (dequeue event, ...)
- Or write wrapper to dequeue the event

Combining subprograms and threads (cont'd)

■ Compute_Entrypoints are used

```
thread HCI_T
features
  Stall_Warning : in event data port Ravenscar.Integer
    {Compute_Entrypoint => "Manager.On_Stall_Warning"};
  Gear_Req      : out event port;
-- ...
end HCI_T;
```

■ Simply pass a “reference” to the data to user’s code

- “reference” to leave optimization options opened
- Generated code interacts with the AADL runtime, as there is no need to express out ports in this case
- or direct user interaction with the AADL runtime

Interaction with the AADL runtime

- **AADL subprograms can be used by multiple threads**
 - Need to preserve context information, data integrity
- **Solution: defines a context for each thread, passed as parameter: opaque container `__aadl_ctx` defined as a struct**
 - Implementation defined, only agreement on members name
 - This context holds “references” to the actual buffer variable
 - Allows for finer optimizations using indirections

```
thread HCI_T
```

```
features
```

```
  Stall_Warning : in event data port Ravenscar.Integer
```

```
    {Compute_Entrypoint => "on_dummy_in"};
```

```
  Gear_Req      : out event port;
```

```
-- ...
```

```
end HCI_T;
```

```
void on_dummy_in (__aadl_ctx self) {  
  /* .. */  
  __aadl_send_output  
    (self->gear_req, &request);}
```

Interaction with data components

■ Define accessors to manipulate data components

data position

features

Update : **provides subprogram**

access Update.Impl;

end position;

data implementation position.Impl

subcomponents

Field : **data** A_Type;

properties

Data_Model::Data_Representation => Struct;

end position.Impl;

■ Code generated handles concurrent access,

■ Manual code interacts through the *Get_Ressource* service

Open issue #1

■ Additional legality rules or constraints ?

- Is that over-specification ? Do we want to generate the_type in types.h, or does it already exist ?

```
data C_Type
```

```
properties
```

```
Data_Model::Data_Representation => Array;
```

```
Data_Model::Base_Type => (classifier (Base_Types::Integer));
```

```
Data_Model::Dimension => (74, 75);
```

```
Source_Language => C;
```

```
Source_Text => ("types.h");
```

```
Type_Source_Name => "the_type";
```

```
end C_Type;
```

■ Ocarina has its own set, other implementations add others

- Call for a constraint annex ? Leave it as opened ?



Open issue #2

■ How to organize source code

- How many files ? Which criteria ? AADL package ? Comp. type ?
- Required for `#include` or Ada's with ..
 - C++ CORBA did not, opened many interoperability issues

■ How to compile code ?

- Outside the scope ?

■ How to configure runtime ?

- Buses ? OS parameters ?
 - E.g. RTEMS requires specific macros for the number of threads, mutexes
- Could be handled by code generation, what for manual code ?
- Additional `Initialize_Entrypoint`, e.g. for process or processor ?

Open Issue #3

■ About modes

- If explicitly modeled using port communication, we rely on standard mechanisms
- What about runtime services ?
 - What is implementor-specific ? Actual representation ? Way to name modes ?
 - Is it something about mode naming in the BA ?

```
subprogram Current_System_Mode  
features
```

```
  ModeID: out parameter <implementor-specific>; -- ID of the mode  
end Current_System_Mode;
```