Current implementation status

- **C implementation tested on OSEK, POSIX and ARINC653**
  - Get_Value/Next_Value/Put_Value
  - Get_Count/Updated
  - Receive_Input/Send_Output
  - Await_Dispatch
  - Current_System_Mode/Set_System_Mode
    - Limited to modes in processes, threads (issue with thread groups in OSATE: no modes in more than 2 layers of coposition)
    - Not on OSEK
  - Await_Mode
- **Not yet implemented**
  - Set_Error_Code/Get_Error_Code
    - Global variable? → not confident with the « thread safeness » of the service.
    - Should be stored in the thread context?

Principle

- **Provide an AADL definition of the RTS:**
  - It can be used in the standard to defined these RTS
  - It can be used in RAMSES to refine AADL models towards code generation

- **Provide a reference implementation in C:**
  - Make sure the existing RTS are sound, and/or define their usage limitations
  - Test to find limitations in the RTS definition and/or the standard (core and annexes)

Integration in RAMSES

- **RAMSES has been deeply reworked to rely on RTS presented hereafter.**
- **Parts in red in this presentation are blocking and require decisions from the comittee**
  - Impact on the core standard:
    - features of subprograms
  - Impact on the code generation annex: consistency rules and minor modifications.
- **More generally, the adoption of a standardized set of precisely defined RTS is necessary**

Detailed specification example: Get_Value()

AADL model

```
subprogram GetValue
        prototypes
                MESSAGE_TYPE: data;
        features
                port_reference: requires data access PortReferenceType;
                MESSAGE_ADDR: in parameter MESSAGE_TYPE
                                {Code_Generation_Properties::Parameter_Usage => By_Reference;};
                status: out parameter StatusType {Code_Generation_Properties::Return_Parameter => true;};
        properties
                Programming_Properties::Source_Name => "Get_Value";
                Programming_Properties::Source_Text => ("aadl_runtime_services.h",
                                                        "aadl_ports_standard.c", "aadl_error.c");
end GetValue;
```

C function (signature)

```
StatusType Get_Value(port_reference_t * dst_port, void * dst);
```

Detailed specification example: Next_Value()

AADL model

```
subprogram NextValue
        prototypes
                MESSAGE_TYPE: data;
        features
                port_reference: requires data access PortReferenceType;
                MESSAGE_ADDR: in parameter MESSAGE_TYPE
                                {Code_Generation_Properties::Parameter_Usage=>By_Reference;};
                status: out parameter StatusType {Code_Generation_Properties::Return_Parameter => true;};
        properties
                Programming_Properties::Source_Name => "Next_Value";
                Programming_Properties::Source_Text => ( "aadl_runtime_services.h", "aadl_ports_standard.c",
                                                         "aadl_error.c");
end NextValue;
```

C function (signature)

```
StatusType Next_Value(port_reference_t * dst_port, void * value);
```

Detailed data type definition: example for StatusType

- **AADL definition**

```
data RuntimeDataType
properties
  Programming_Properties::Source_Text =>
        ("aadl_runtime_services.h");
end RuntimeDataType;

data StatusType extends RuntimeDataType
properties
    Programming_Properties::Source_Name
        =>  "error_code_t";
end StatusType;
```

- **Corresponding C source code**

```
typedef enum error_code_t
{
  OK,
  EMPTY_QUEUE,
  FULL_QUEUE,
  LOCK_ERROR,
  OUT_OF_BOUND,
  INVALID_PARAMETER,
  INVALID_SERVICE_CALL,
  MISSING_INPUT,
  INIT_HYPERPERIOD
} error_code_t;
```

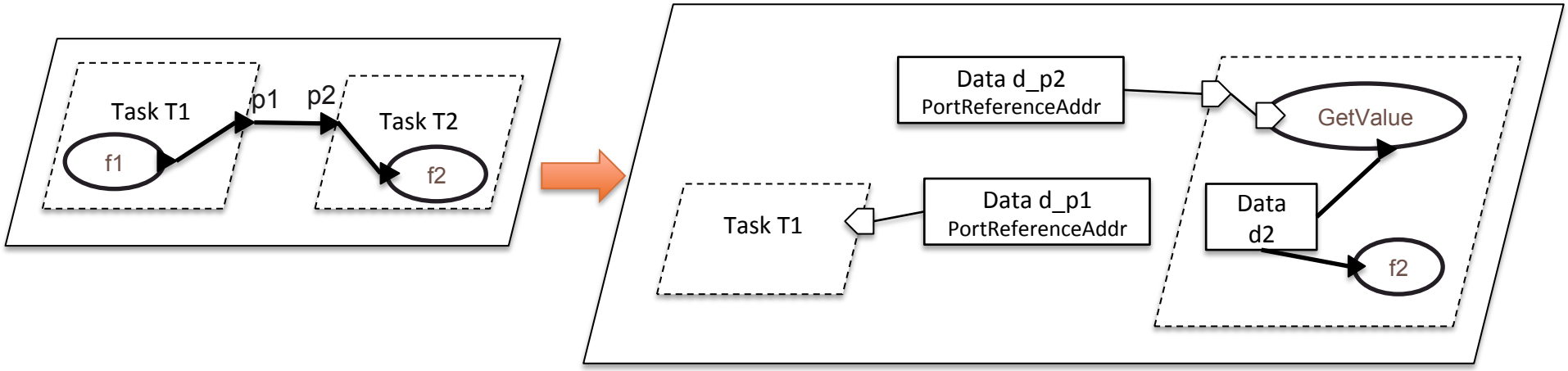
- **Could be extended for standardization**

General remarks on the RTS usage

- **User code can only use Set_System_Mode/ Get_System_Mode to manipulate processes modes**
  - No control on system-wise modes; no need (so far) to control threads or subprograms modes.
  - Calling Set_System_Mode in user code may hide mode transitions and/or mode transition conditions: implemented in source code but not explicitly represented in AADL

- **Integrating the RTS in a generated code is « easy » if the code generation *Convention* uses le « Legacy » property value:**
  - Calls to the RTS are embedded in the generated code;
  - Port values are directly passed as parameters to the user code

generated code with Legacy convention



- **Generate C pseudo-code**

```
void thread_entrypoint() {
        …
        data_type_of_d d;
        error_code_t err = Get_Value(&d_p2, &d);
        if(!err)
                f2(d);
}
```

General remark on the RTS usage

- **User code can only manipulate AADL features if the AADL convention style is selected**
  - Integrating such user code is more difficult in the current verion of AADL and the code generation annexe:
    - Requires to define programming interfaces, data types, and code generation rules so as to ***define what a developper can write in a subprogram source code***
    - Some limitations in the AADL standard have to be addressed

Issues related to the core AADL standard and code generation annexe

- **Connection of subprogram parameters with**
  - Data port → OK if integration style is legacy
    - See previous example
  - Event [data] port → LIMITED if integration style is legacy
    - Example1: dequeue only when the recipient subprogram is called (i.e. access to the 3rd element of the queue must be modelled with 3 calls to the recipient subprogram) **??? → If yes, it should be specified!**
    - Example2: how to send a mode change request on an event port from the source code of a subprogram? **Note: it should be written that only parameters/data accesses are allowed in AADL subprograms definition if the Legacy convention is used!**
    - Example 3: how to know from the source code if an event was received?

This is why the AADL convention was defined, but…

AADL Convention style: subprogram call

- **Take an example from RAMSES demo: the logging code reads value from input ports to write it on the robot screen:**

```
subprogram Log
        features
                angle : in parameter data_types::Int;
        properties
                Source_language => (C);
                Source_text => ("logging.h", "logging.c");
                Source_name => "logData";
                Code_Generation_Properties::Convention
                => AADL;
end Log;
```

```
#include "gtypes.h"

int angle_array[ANGLE_LOG_SIZE];

int iter = 0;
void logData(__log_context * ctx)
{
 char prefix[15] = "Ang=";
 char txt[80];

 int i, angle;
 int ret = Get_Value(ctx->angle, &angle);
 display_goto_xy(0, 6);
 if(!ret) {
  angle_array[0]=angle;
  itoa(angle, txt, 10);
  strcat(prefix,txt);
 }
 for(i=1;i<ANGLE_LOG_SIZE;i++) {
  ret = Next_Value(ctx->angle, &angle);
  if(!ret) {
   angle_array[i] = angle;
   ...
```

AADL Convention style: subprogram call

- **Take an example from RAMSES demo: the logging code reads value from input ports to write it on the robot screen:**

```
#include "gtypes.h"

int angle_array[ANGLE_LOG_SIZE];

int iter = 0;
void logData(__log_context * ctx)
{
  char prefix[15] = "Ang=";
  char txt[80];

  int i, angle;
  int ret = Get_Value(ctx->angle, &angle);
  display_goto_xy(0, 6);
  if(!ret) {
    angle_array[0]=angle;
    itoa(angle, txt, 10);
    strcat(prefix,txt);
  }
  for(i=1;i<ANGLE_LOG_SIZE;i++) {
    ret = Next_Value(ctx->angle, &angle);
    if(!ret) {
      angle_array[i] = angle;
      …
```

These are based on rules developers should know; e.g. defined (or to be updated) in the Code Generation Annex:

ctx->angle : angle is a field of ctx
- ctx is the subprogram context data type
- angle is defined as a PortReferenceAddr (reference to a PortReferenceType variable)
*… **This is good is the in parameter of log is connected to a port but not so good if connected to a parameter, a data subcomponent…***

However, the data type of the angle field should not depend on how it is connected (otherwise the source code becomes « instance-model dependent »).

Issues related to the core AADL standard and code generation annexe

- **Should a subprogram parameter be part of the subprogram context data type?**
  - If yes, what is the data type of a subprogram parameter field in a subprogram context data type?
  - If no, can parameters be connected to ports? If yes, we fall on the same limitations as before w.r.t the Legacy convention
- **Proposal:**
  - Add [in | out | inout] [event|event data | data] ports to subprogram classifiers → ***Change request in the core standard***
  - Add a rule stating that parameters can only be connected to parameters or data subcomponents if the AADL convention is selected → ***Change request in the Code Generation Annex***
  - Add a rule stating tha only parameters and data accesses can be used if the Legacy convention is used → ***Change request in the Code Generation Annex***
  - In the Legacy convention, add a description of the expected behavior when
    - Input parameter of a subprogram is connected to an input event data port (call to Get_Value for the first call and Next_Value for the subsequent calls?)
    - Input parameter of a subprogram is connected to an empy in event data port, or a unfresh input data port

Code generation annex

- **The code generation annex states that the context data type is constructed as follows:**

  __<component_identifier>_context

- **Application for the logging example:**

  subprogram Log

  …

  end Log;

  void logData(__log_context * ctx)

Developper responsibility to ensure name unicity

Write code « more independently » from the AADL model

- **However, it would be safer and easier for the source code developper to use (if defined) the source_name to create context**

  void logData(__logData_context * ctx)

- **Note: force source_text definition if AADL convention is used? Same with Legacy?**
  - Avoid issues with aadl subprogram type vs implementation vs extension

Scheduling table representation

- **How is this related to runtime services?**
  - We are implementing code generation for immediate port connections using a static scheduler…. Implementation of runtime services can just check that the scheduling had writer/producer executed before the reader/consumer…
  - The core standard is incomplete to that respect:
    - defines a static property value for the scheduling_protocol
    - defines a frame_period property
    - BUT the scheduling table content cannot be defined
  - Proposal: adopt the scheduling table property of the ARINC653 annexe as a core standard property (add in code, remove from ARINC653 annexe)

```
--
-- The Execution_Slot type represents a record to specify a time
-- slot/frame of a processor
--
Execution_Slot : type record (
                    Computation_Unit : reference (processor);
                    Start_Time : Time;
                    End_Time : Time;);


--
-- Execution_Slots represents all the execution slots for the thread
--
Execution_Slots : list of Execution_Slot applies to (thread, thread group);
```

About exceptions in BA

- RTS must define return error code, but they are not real exceptions… !

Conclusion

- **Proposal: adopt and adapt the aadl_runtime.aad model to standardize AADL runtime services**

- **Integration of these runtime services in RAMSES**
  - mature for Legacy convention
  - requires decisions from the committee for the AADL convention
  - Experimented for what seems not to require more discussions

- **Path to follow is an iterative process: answer current issues, test, rework…**
  - Requires fixes for continuing towards the next step

Thank you for your attention

etienne.borde@telecom-paristech.fr

- **Subprogram inheritence**
  - Parameter list (and thus order)
- **Relative path to source**
  - Rule = from the declarative aadl model
- **Code generation annex**
  - Is_Reference (data)