AADL2RTOS Build System User Guide

The AADL2RTOS build system allows generation of BRTOS system images from AADL models and C/C++ code. The build information is derived from the structure of the AADL model (threads and connections) and properties that describe the location of the source code used for the build process.

1 AADL Support for System Build Information

For the most part, the information required for the BRTOS system build process comes from property sets in AADL. When possible, we will use standard AADL property sets to provide this information; however in some cases, due to expectations of the Tower build system or BRTOS, we will define new properties using the property set SMACCM_SYS.

The types of information required can be split into several categories:

1. Type declarations for information exchanged between threads and devices.

2. Port declarations for describing the input/output interfaces supported by a thread, process, or system.
    - All ports will be associated with a type
    - For input event and event-data ports, the properties include:
        - the name and file location of *event handlers,* which will be the user-level entrypoints invoked when the event occurs.
        - the size of the queue used to store incoming events, and a queuing strategy in case the queue fills.

3. Thread declarations for describing the different tasks supported by BRTOS.
    - Each thread will be associated with a *dispatch protocol,* which can be *periodic, aperiodic,* or *sporadic*.
    - For periodic threads, properties include:
        - The name and file location of an *event handler,* which is a user-level entrypoint invoked once per period.

4. System-implementation declarations describing the platform and referenced libraries
    - The output directory for the system image
    - TBD: there are a variety of things we could store here, spanning the gamut between a pseudo-makefile and loader with:
        - the gcc compiler flags,
        - linker flags
        - tool version information,

        - referenced c files (for library functions),
        - loader memory addresses for PX4,
        - interrupt mappings for the PX4 and the current hardware configuration
    - Alternately, we could make this a reference to the files with this information and store only:
        - The locations of any external makefiles necessary to build the BRTOS, Ivory, and handwritten code.

2 Type Declarations

All type declarations shall follow the conventions of the AADL data modeling annex. This annex is available under SVN at:
svn_smaccm/architecture/AADL-spec/ Revised_BehaviorAnnex_Draft_20121031.docx

3 Port Declarations

Port declarations are split along two axes: whether the ports are input or output ports and whether the port is a *data, event,* or *event-data* port. Data ports do not require any associated properties.

Event and Event-data input ports cause thread dispatch, and so require properties as documented below:

`SMACCM_SYS::Compute_Entrypoint_Source_Text : list of string =>` This property provides the name(s) of the entrypoint function(s); given multiple entrypoint functions, it calls them sequentially in the order in which they are provided. This property is an extension of the standard AADL `Compute_Entrypoint_Source_Text` property, which allows only one entrypoint per event.

`SMACCM_SYS::Compute_Entrypoint_Header : list of string =>` This property provides the name of the header files containing the entrypoint function(s) in Compute_Entrypoint_Source_Text.

`Source_Text: list of string =>` This property provides the names of the source files containing the entrypoint function(s) in Compute_Entrypoint_Source_Text.

`Queue_Size: aadl_integer =>` This property provides the size of the queue used to buffer events (and their associated data, for event-data queues)

Related to scheduling, but not necessary for system build, is the following properties:

`Compute_Execution_Time : Time_Range =>` This describes the expected execution time for the event. Note that this is the execution time of *all* handlers invoked for the event.

TODO: In addition, we may want to add an additional property describing the sporadic dispatch rate for the handler.

An example of an input port is the following:

```
signalCh: in event port
{
    Compute_Entrypoint_Source_Text => "exec_signalCh_threadB";
   SMACCM_SYS::Compute_Entrypoint_Header =>
     "tower_task_loop_fooBarSinkTask_24.h";
   Source_Text => "tower_task_loop_fooBarSinkTask_24.c"
   Queue_Size => 2;
   Compute_Execution_Time             => 0 ms .. 5 ms;
};
```

Output ports cause output to other tasks to occur, so contain properties describing the name of the output function and its location, as follows:

`SMACCM_SYS::CommPrim_Source_Text : string =>` This property provides the name of the function that will be used to write to the output port.

`SMACCM_SYS::CommPrim_Source_Header : string =>` This property describes the header containing the `CommPrim_Source_Text` function used for writing to the port.

TODO: Who generates the header? It would seem that the natural place for this is in the AADL code; but it could alternately be the client responsibility.

NOTE: It would also be possible to use a naming convention for the output functions. A reasonable one would be “aadl2brtos__” ^ <thread_name> ^ “__” <port_name>, if we disallow the aadl2brtos prefix elsewhere.

4 Thread Declarations

AADL threads are bound to BRTOS tasks. Each thread has a number of input and output ports that comprise its interface, and a *dispatch protocol* that defines how often it will run. The dispatch protocols describe how and when the thread will be dispatched. The currently supported dispatch protocols are `Periodic`, where the thread runs once per period (as specified by the Period property), `Aperiodic`, where the thread runs because of an occurrence of an input event (as defined by the input event ports), and `Hybrid`, where the thread is dispatched once during its period and also input events. In order to dispatch from input events, the associated input event port must have a dispatch function (see Section 3) .

The following properties are used during code generation:

`Dispatch_Protocol : {Periodic, Sporadic, Aperiodic, Timed, Hybrid, Background} =>` The dispatch protocols describe how and when the thread will be dispatched. The currently supported dispatch protocols are `Periodic`, `Aperiodic`, and `Hybrid.`

`Period : Time =>` For periodic and hybrid threads, the period defines the dispatch interval for the thread. For aperiodic threads, it is ignored.

`Source_Stack_Size : Size =>` This parameter defines the stack size for the thread.

`SMACCM_SYS::Compute_Entrypoint_Source_Text : list of string =>` This property provides the name(s) of the entrypoint function(s) for periodic dispatch; given multiple entrypoint functions, it calls them sequentially in the order in which they are provided. This property is an extension of the standard AADL `Compute_Entrypoint_Source_Text` property, which allows only one entrypoint per event. If the dispatch protocol is aperiodic, this property is ignored.

`SMACCM_SYS::Compute_Entrypoint_Header : list of string =>` This property provides the name of the header files containing the entrypoint function(s) in Compute_Entrypoint_Source_Text.

`Source_Text: list of string =>` This property provides the names of the source files containing the entrypoint function(s) in Compute_Entrypoint_Source_Text.

`Initialize_Entrypoint_Source_Text : list of string =>` This property provides the name of the initializer function for the thread. This property is optional.

`Compute_Execution_Time : Time_Range =>` This property describes the best and worst case execution times for the entrypoint function. Note that if multiple entrypoint functions are used, this time is the aggregate time for dispatch of all functions.

An example of a thread declaration is as follows:

```
thread Thread_B
    features
      datap: in data port dt.rec;
      signalCh: in event port
       { … aadl event channel properties here.
       };

    properties
      Dispatch_Protocol => Aperiodic;
      Period => 30 ms;
      Source_Stack_Size => 64 bytes;
      Source_Text => ("src/rigel_task.c");
      Initialize_Entrypoint_Source_Text => "init_barSourceTask";
end Thread_B;
```

5 Process Declarations

Processes in AADL define protected memory spaces. As BRTOS does not support memory protection, processes have no real meaning when generating BRTOS code, other than to “house” threads (threads must be immediate children of processes) and to describe how threads communicate using *connections*. Connections wire together output ports from one thread to input ports of another thread. Note that the process itself has an interface, so thread outputs can propagate to process outputs and process inputs can be routed to thread inputs.

In AADL, “fan out” from a single emitting process to several receiving processes is allowed. Currently “fan in” from several emitting processes to one receiving process is not supported, though this could be added for event ports.

An example process declaration is:

```
process system_proc
end system_proc;

process implementation system_proc.Impl

    subcomponents
             A : thread Thread_A ;
             B : thread Thread_B ;
             systick : thread systick_signal ;

    connections
      port systick.chEmitter -> A.signalCh;
      port systick.chEmitter -> B.signalCh;
      port A.foo_data -> B.signalCh;

end system_proc.Impl;
```

6 Expected Function Signatures

The interactions between the RTOS, the AADL glue code, and the client code are ultimately through C functions. There are only a handful of different signatures necessary to create this interface. The signatures are described below. All signatures include a THREAD_ID parameter that identifies the currently executing thread. The reason for this parameter is that AADL distinguishes thread *implementations* from thread *instances.* The same implementation may have multiple instances. We expect the client to write entrypoints on a per-thread-implementation basis. However, the different thread instances have different connections, so the port functions must know which instance is reading or writing data. The THREAD_ID parameter allows correct routing. In addition, if the client needs to initialize thread-instance data, this parameter allows mapping from a thread instance to thread instance data.

6.1 Port Function Signatures

For writing to output ports, the signature of the provided AADL glue code is as follows:

```
void sample_write_function(THREAD_ID tid, sample_type *to_write);
```

Where sample_type is the AADL type of the port. The provided function will copy the data into a shared variable, so the information stored in to_write is owned by the caller.

For reading from input ports, the signature of the provided AADL glue code is as follows:

```
void sample_read_function(THREAD_ID tid, sample_type *to_read);
```

Where sample_type is the AADL type of the port. The provided function will copy the input data into the address pointed to by to_read, overwriting the previous contents of this address.

NOTE: In the current semantics we have defined state that the contents of the input ports will not change during the execution of a thread; calling a reader function at multiple instants will always return the same value. The input values are copied into thread-local storage immediately prior to dispatch.

6.2 Thread Function Signatures

For thread initialization functions, the expected signature to be provided by the client is:

```
void sample_initialize_thread(THREAD_ID tid);
```

This initialize function will be called once for each *thread instance* that is bound to a particular *thread implementation.* These initializer functions are not expected to be thread safe.

For entrypoint functions, we have a choice in terms of interface: we could either include the current state of the input environment via input parameters to the function, or we could provide only the thread id and rely on the client to query the current input environment using the port 'read' functions. Either signature is straightforward to support; however, if we pass input parameters to the function, then ordering becomes a concern; we could pass the parameters will be passed in the order that they are in the AADL file, but there is the possibility for mis-identification of parameters, especially if the AADL file is modified but the client code is not. Therefore, we have chosen to implement entrypoints as single parameter functions, and require that the client call read() functions to extract the input values.

```
void sample_entrypoint(THREAD_ID tid);
```

7 System Implementation Declarations

There are a variety of properties that may be useful at the "top level" system implementation for the BRTOS generation. The set of these properties is still under consideration and will be guided by our experience in using the system build tool.

8 Build Process

TBD. Description here of how to invoke aadl2rtos and the different phases of system build.