AADL Trusted Build System User Guide

Mike Whalen  
Kristin Mead  
University of Minnesota  
5/20/2016

# Version History

|  |  |  |
| --- | --- | --- |
| Date | Version | Information |
| 2/3/2014 | 0.1 | Initial version of trusted build document. Original title: AADL2RTOS Build System User Guide.docx |
| 5/21/2016 | 1.0 | Renamed: AADL Trusted Build System User Guide.docx. Added:   * Information on different thread types: {active, passive, external}. * Type specifications: AADL types and external types * Shared memory resources: managed and “raw”. * “Raw” RPCs * Using trusted build in CAmkES build process |

Table of Contents

[1 Version History 2](#_Toc451520771)

[2 Introduction 5](#_Toc451520772)

[3 Overview and Example of the Build Process 6](#_Toc451520773)

[4 Running Example 8](#_Toc451520774)

[5 Support for Constructs in AADL 9](#_Toc451520775)

[5.1 Processes 10](#_Toc451520776)

[5.1.1 Properties 10](#_Toc451520777)

[5.1.2 Example 10](#_Toc451520778)

[5.1.3 Discussion 10](#_Toc451520779)

[5.2 Threads 11](#_Toc451520780)

[5.2.1 Properties 11](#_Toc451520781)

[5.2.2 Example 12](#_Toc451520782)

[5.2.3 Discussion 13](#_Toc451520783)

[5.3 Types 13](#_Toc451520784)

[5.3.1 Base Types 13](#_Toc451520785)

[5.3.2 Arrays 15](#_Toc451520786)

[5.3.3 Structures 15](#_Toc451520787)

[5.3.4 External Types 16](#_Toc451520788)

[5.3.5 Naming types in implementation code 16](#_Toc451520789)

[5.4 Ports 16](#_Toc451520790)

[5.4.1 Properties 17](#_Toc451520791)

[5.4.2 Examples 17](#_Toc451520792)

[5.4.3 Interrupt Ports and Managing ISRs 18](#_Toc451520793)

[5.4.4 Discussion 19](#_Toc451520794)

[5.5 Shared Data 19](#_Toc451520795)

[5.6 RPC Declarations 20](#_Toc451520796)

[5.7 System Implementation AADL Declarations 20](#_Toc451520797)

[6 Function Signatures in C 21](#_Toc451520798)

[6.1 Port Function Signatures 22](#_Toc451520799)

[6.2 Thread Function Signatures 22](#_Toc451520800)

[7 Schedulability Analyis 22](#_Toc451520801)

[8 Install Process 22](#_Toc451520802)

[9 Build Process 23](#_Toc451520803)

# Introduction

The AADL trusted build system allows generation of system images in a variety of operating systems: currently eChronos, CAmkES/seL4, and VxWorks 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.

The goal of the trusted build system is to support portable, high assurance image generation between multiple operating systems. For the most part, this goal has been achieved, but certain aspects of system development, notably drivers, still involve some OS-specific differences. In addition, the performance and safety characteristics of different operating systems can differ substantially: for example, eChronos and VxWorks are RTOSes that do not provide memory protection, while CAmkES/seL4 is a microkernel that does provide memory protection. This means that context switches are faster in the RTOSes, and the mechanisms and performance characteristics of AADL communications differ substantially between the OSes. For accurate schedulability analysis, some knowledge of the operating system is required. OS-specific aspects of the trusted build process will be highlighted within the document.

In addition, the trusted build system **does not** perform the final system build. Each of the operating systems have a final build step that is OS-specific, and that is expected to be under the control of the OS builder. Instead, what is provided is an extendable Makefile, and all OS configuration, manifest, glue code, and C code in the locations expected by the operating system. One aspect of the trusted build system is that it expects to be used with *external* components which may involve additional build steps outside the control of the AADL model (for example, the Linux VM in the air team Phase 2 architecture). In this document, for each operating system, we will discuss how one could construct a final build using the provided outputs of the trusted build tool, and ideas of how this process could be enriched with external components. In the end, however, the full system build process is outside the scope of this document.

The AADL trusted build system supports a relatively small subset of the AADL language; AADL allows a very wide range of system descriptions and parameters to be added to a model, and it would be difficult and labor-intensive to construct a high-assurance build system that supports the full range of AADL constructs. We will describe which AADL component type, implementation, and instances will be supported, and describe the set of built-in properties that can be supported.

For the most part, the information required for the 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, we will define new properties using the property set SMACCM\_SYS. In addition, we add a fairly important extension to AADL: the idea of *monitors* that can perform computation over shared data via RPCs, the trusted build system extends AADL in a handful of important ways.

# Overview and Example of the Build Process

An overview of the build process is shown in Figure 1. Although this example is for the CAmkES operating system, the ideas are similar across all supported operating systems. The user-provided information is on the left, the AADL-generated artifacts are in the middle, and the OS-specific make steps are on the right. On the left, we have the user-provided AADL files (top left) and C files (bottom left). From the AADL files, we generate the OS-specific configuration information. For CAmkES, this is a set of CAmkES *components* that implement the *threads* in the AADL model. Each *component* contains a set of C files (and libraries) which define the behavior of the component. To construct a *component* we use the AADL model to define a configuration file for the component that defines its interface to the external world, and we information as to the locations of C files relevant to the component to copy the files to the appropriate location for CAmkES.

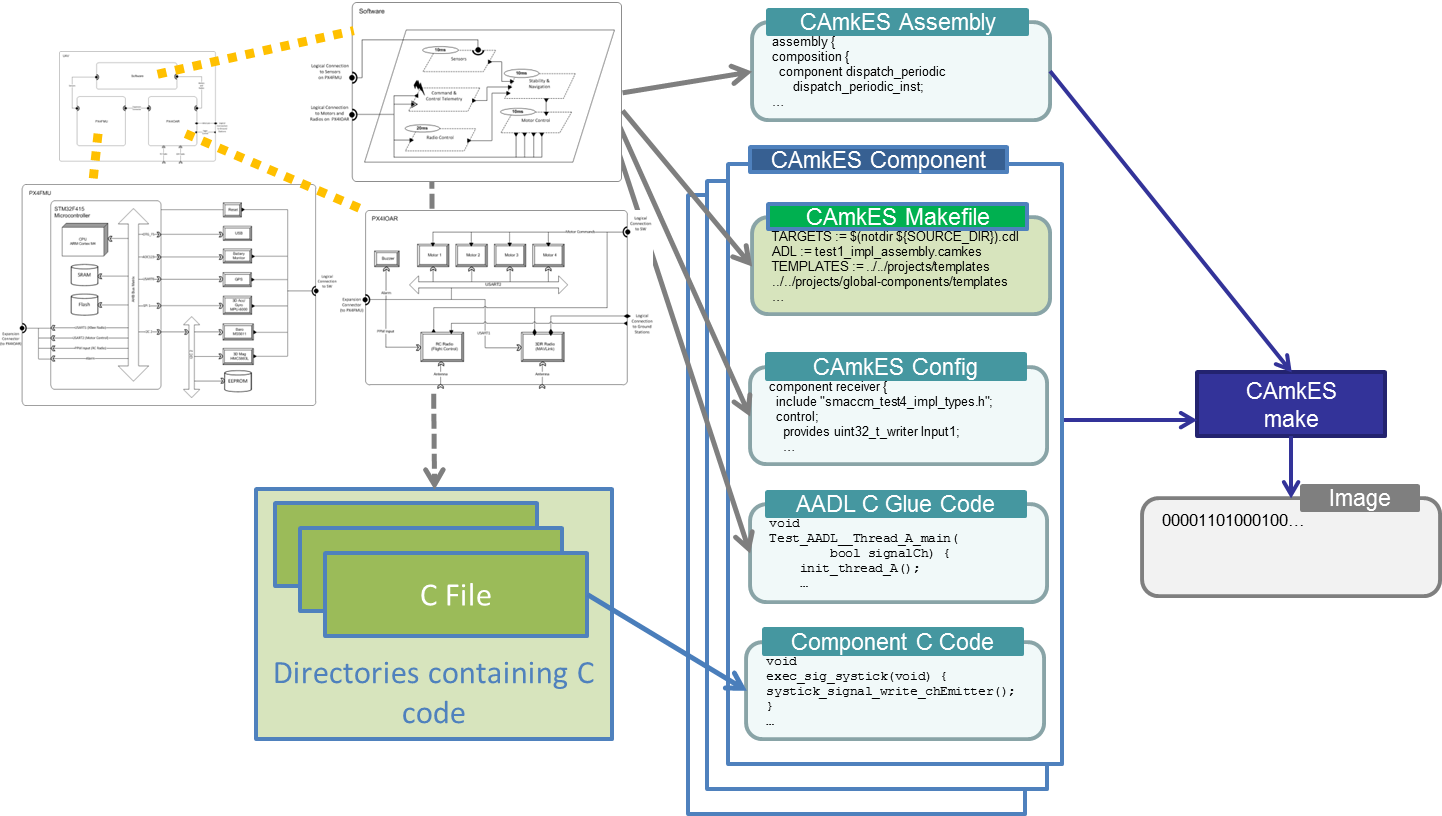


Figure 1: Overview of the build process (CAmkES OS)

Given this information, we generate information sufficient to build a system image, given the model description in AADL. However, because we allow *external* threads in the model, which are considered “outside” of the standard system build process, this may not be sufficient information to build the system image. We therefore allow extension of the generated Makefile (the color green indicates user provided / user-extendable) for a component to add additional information.

If no “external” information is required, then the generated Makefiles are sufficient to build a system image in the target operating system using the OS-provided generation tools. Continuing to use CAmkES, for example, there is a CAmkES make tool that uses the Makefiles for each component to build the system image.

# Running Example

In order to illustrate the generation process, we will use a very simple running example that shows several of the supported features of the code generation process. This example can be found in the SMACCM github site (<https://github.com/smaccm/smaccm>) in the models/Trusted\_Build\_Test/test7 directory, and is shown in Figure 2. The example demonstrates communication between two threads using scalar reads and writes to an output data port. The sender thread updates an output value on port Output1 every 500 ms, which is consumed by a receiver thread on port Input1 every 1000 ms.

-- AADL Test 7: thread communication using scalar read/write.

**package** test7

**public**

**with** Base\_Types;

**with** SMACCM\_SYS;

**with** Data\_Model;

**thread** sender

**features**

Output1: **out** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::CommPrim\_Source\_Text => "ping\_Output1";

};

**properties**

Dispatch\_Protocol => Periodic;

Period => 500 ms;

Source\_Text => ("user\_code/user\_sender.c");

Compute\_Entrypoint\_Source\_Text => "periodic\_ping";

Priority => 10;

Stack\_Size => 256 bytes;

Compute\_Execution\_Time => 10 us .. 50 us;

**end** sender ;

**thread** receiver

**features**

Input1: **in** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::CommPrim\_Source\_Text => "ping\_Input1";

};

**properties**

Dispatch\_Protocol => Periodic;

Period => 1000 ms;

Source\_Text => ("user\_code/user\_receiver.c");

Compute\_Entrypoint\_Source\_Text => "periodic\_ping";

Priority => 8;

Stack\_Size => 256 bytes;

Compute\_Execution\_Time => 10 us .. 50 us;

**end** receiver ;

**process** proc

**end** proc;

**process** **implementation** proc.Impl

**subcomponents**

s: **thread** sender;

r: **thread** receiver;

**connections**

s\_to\_r : **port** s.Output1 -> r.Input1;

**end** proc.Impl;

**system** test7

**end** test7;

**system** **implementation** test7.impl

**subcomponents**

tl: **process** proc.Impl;

**properties**

SMACCM\_SYS::OS => CAmkES;

SMACCM\_SYS::HW => QEMU;

**end** test7.impl;

**end** test7;

Figure 2: Example of Dataport Communications between two threads.

The example illustrates many of the features in the trusted build tool. First, *threads*, which define an independent sequence of execution. In AADL, thread execution occurs at *entrypoints*, which define functions that are invoked in response to some external stimulus. In this example, both threads are invoked due to periodic dispatch. Threads have a *priority,* which defines their precedence for scheduling by the operating system. In the trusted build tool, a higher priority number indicates higher precedence (e.g., *sender*, with priority 10, will run before *receiver*, with priority 8, if both are runnable).

Threads are grouped into *processes,* which defined protected address spaces. As we will describe in more detail below, the concept of a *process* is somewhat superfluous in the current trusted build tool because the three back-end operating systems do not have a real notion of a process: eChronos and VxWorks do not support protected address spaces and CAmkES requires that each component (thread) in the system has its own address space. Processes allow one to specify the “wiring” between thread ports in AADL. Finally, the set of processes in the model (in this case only one) are grouped into a *system implementation*, where it is possible to specify the desired operating system and hardware target for the trusted build tool. In this case, we are targeting CAmkES/seL4 and we will run on the QEMU virtual machine.

AADL attempts to make a strong separation between component *types, implementations,* and *instances*, which you can see in this model. For the most part, for code generation, the component *types* do not contain a significant amount of information.

# Support for Constructs in AADL

In this section, we describe the means by which we support the various component types, implementations, and instances defined in AADL. Note that an overview of the various features of AADL is outside the scope of this document. For such an overview, we refer the user to the *Model-Based Engineering with AADL: An Introduction to the SAE Architecture Analysis and Design Language* book by Peter Feiler and David Gluch, and the AADL standard: SAE 5506B: *Architecture Analysis and Design Language.*

## Types

All type declarations follow the conventions of the AADL Base\_Types package and Data\_Model package (described in the data modeling annex). The data modeling annex is available from the SAE ([www.sae.org](http://www.sae.org)) bundled as part of the AADL Behavioral Annex.  
  
In the subsections below, we provide a list of the supported types. In the trusted build system, we support *internal types,* which are declared within the AADL model and *external types* that are declared externally (usually a C header file). External types are treated as opaque within the AADL model. For internal types, the trusted build system generates a header file: smaccm\_<model\_name>\_types.h, which contains the definitions of all of the user-defined types. This location of this file is dependent on the particular target OS.

### Base Types

AADL supports a range of basic types, as described in the Base\_Types package. The currently supported types are as follows:

**data** Integer\_8 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Signed;

Data\_Size => 1 Bytes;

**end** Integer\_8;

**data** Integer\_16 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Signed;

Data\_Size => 2 Bytes;

**end** Integer\_16;

**data** Integer\_32 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Signed;

Data\_Size => 4 Bytes;

**end** Integer\_32;

**data** Integer\_64 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Signed;

Data\_Size => 8 Bytes;

**end** Integer\_64;

-- Unsigned integer of various byte sizes

**data** Unsigned\_8 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Unsigned;

Data\_Size => 1 Bytes;

**end** Unsigned\_8;

**data** Unsigned\_16 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Unsigned;

Data\_Size => 2 Bytes;

**end** Unsigned\_16;

**data** Unsigned\_32 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Unsigned;

Data\_Size => 4 Bytes;

**end** Unsigned\_32;

**data** Unsigned\_64 **extends** Integer

**properties**

Data\_Model::Number\_Representation => Unsigned;

Data\_Size => 8 Bytes;

**end** Unsigned\_64;

  **data** Float\_32 **extends** Float

**properties**

    Data\_Model::IEEE754\_Precision => Simple;

    Source\_Data\_Size => 4 Bytes;

**end** Float\_32;

**data** Float\_64 **extends** Float

**properties**

    Data\_Model::IEEE754\_Precision => Double;

    Source\_Data\_Size => 8 Bytes;

**end** Float\_64;

In the current trusted build tool suite, strings are not supported directly, but these can be implemented as arrays of Unsigned\_8 or Unsigned\_16 elements.

### Arrays

AADL has a somewhat clunky syntax for arrays and structures. They can be defined via data declarations as follows:

**data** array\_type

**properties**

Data\_Model::Data\_Representation => Array;

**end** array\_type;

**data implementation** array\_type.a

**properties**

Data\_Model::Base\_Type => (**classifier** (Base\_Types::Unsigned\_32));

Data\_Model::Dimension => (4);

**end** array\_type.a;

The first declaration states that the implementations of the base type array\_type will be arrays. The second provides the structure of the type. In this example, we have a four element array of Unsigned\_32 integers. Any defined type can be used as the Base\_Type of the array. **NB**: the data model supports direct definition of multi-dimensional arrays by allowing the “dimension” property to have multiple elements, e.g.,

**data** **implementation** array\_type.mult

**properties**

Data\_Model::Base\_Type => (**classifier** (Base\_Types::Unsigned\_32));

Data\_Model::Dimension => (4,5);

**end** array\_type.mult;

However, this format is not directly supported in the trusted build system. The same effect can be achieved by iterated definitions:   
  
**data** **implementation** array\_type.row

**properties**

Data\_Model::Base\_Type => (**classifier** (Base\_Types::Unsigned\_32));

Data\_Model::Dimension => (4);

**end** mult\_array\_row;

**data** **implementation** array\_type.mult

**properties**

Data\_Model::Base\_Type => (**classifier** (mult\_array\_row));

Data\_Model::Dimension => (5);

**end** mult\_array;

### Structures

Structures are defined in a two step process in AADL. First a data declaration is provided that declares something *is* a structure, followed by an implementation that contains the fields of the structure:

**data** a\_struct

**properties**

Data\_Model::Data\_Representation => Struct;

**end** a\_struct;

**data** **implementation** a\_struct.impl

**subcomponents**

field1 : **data** Base\_Types::Float;

field2 : **data** Base\_Types::Float;

**end** a\_struct.impl;

This example defines a structure with two “float” fields named field1 and field2.

### External Types

It is possible to define types externally in C and leave them opaque within the AADL model. This is done by declaring the type to be external and providing a reference to the location where the type is defined:

**data** can\_message

**properties**

SMACCM\_SYS::Is\_External => **true**;

SMACCM\_SYS::CommPrim\_Source\_Header => "canDriverTypes.h";

**end** can\_message;

### Naming types in implementation code

Each user-defined data implementation is assigned a name conjoining the base type and implementation type. For example for the structure implementation, the type would be: a\_struct\_impl. If a user wishes to override the default type name in the generated C code, this can be specified by using the “C\_Type\_Name” property:   
  
**data** **implementation** a\_struct.impl

**subcomponents**

field1 : **data** Base\_Types::Float;

field2 : **data** Base\_Types::Float;

**properties**

SMACCM\_SYS::C\_Type\_Name => "My\_type\_name";

**end** a\_struct.impl;

## Ports

Port declarations describe the standard mechanism by which threads communicate. The trusted build tool only supports unidirectional ports, so each port is either an *input* port or an *output* port. Ports are then split between queued and unqueued varieties. *Data ports* do not perform queueing of data (updates can be missed) and provide a “last value written” semantics when they are read. *Event data ports* provide a queueing semantics, so each update is enqueued (up to a queue size limit). In addition, these ports can cause a *dispatch* of the target thread through the use of an *entrypoint* (similar to the way that periodic threads are dispatched). *Event ports* describe data-less *signals* to be sent to a destination.

Ports are routed to each other through the use of *connections* (see examples in the example of a process in Section 5.1.2).

### Properties

Different kinds of ports have different properties, as described below:

SMACCM\_SYS::CommPrim\_Source\_Text : string  
**Relevant to:** all output ports

This property describes the name of the function to be generated for the thread to write to the output port. The AADL generator creates a default function name by conjoining the thread name with the port name, but this property allows the user to override the default name.

Compute\_Entrypoint\_Source\_Text : string  
SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text : list of string  
**Relevant to:** input event and event data ports.   
These properties allow dispatch to one (or in the SMACCM\_SYS version, multiple) entrypoint functions based on the arrival of an event. The entrypoint function(s) should take a single input matching the type of the port (void for event ports). These properties are essentially the same as the thread properties used for periodic dispatch.

Compute\_Execution\_Time : Time\_Range   
**Relevant to:** input event and event data ports

This property describes the expected execution time for the entrypoints that process the event. Note that this is the total execution time of \*all\* entrypoints invoked for the event. This property is used for schedulability analysis, but is not strictly necessary for system build.

Queue\_Size: aadl\_integer   
**Relevant to:** inputevent data ports  
This property provides the size of the queue used to buffer event messages (and their associated data, for event-data queues). Messages are queued FIFO, and if the buffer is full, then the event broadcast from the sender fails. This will be described in more detail in Section 6.

SMACCM\_SYS::Sends\_Events\_To: aadlstring

**Relevant to:** input event and event data ports.

This property is used for schedulability analysis and communication with monitors. It is a list of sets, where each set gives a list of ports and the maximum number of messages that will be emitted to that output port. For example: given the declaration SMACCM\_SYS::Sends\_Events\_To => "{ {1 Output1}, {1 Output2, 2 Output3} }"; for a single dispatch, the port entrypoint function could emit one output message to port Output1 or, alternately, one output message to port Output2 and two messages to port Output3.  
  
If monitors are used, then this property is *required*. Otherwise, it is *optional.*

Source\_Text: list of string   
**Relevant to:** input event and event data ports  
This property provides the names of the source files containing the entrypoint function(s) in Compute\_Entrypoint\_Source\_Text. These properties are essentially the same as the thread properties used for periodic dispatch.

### Examples

Examples of input and output ports are:

signalCh: **in** **event** **port**

{

Compute\_Entrypoint\_Source\_Text => "exec\_signalCh\_threadB";

Source\_Text => "tower\_task\_loop\_fooBarSinkTask\_24.c"

Queue\_Size => 2;

Compute\_Execution\_Time => 0 ms .. 5 ms;  
};

Input1: **in** **event** **data** **port** Base\_Types::Unsigned\_32 {

Queue\_Size => 3;

};

Output1: **out** **event** **data** **port** Base\_Types::Unsigned\_32

{

SMACCM\_SYS::CommPrim\_Source\_Text => "ping\_Output1";

};

Output2: **out** **data** **port** a\_struct.impl {

SMACCM\_SYS::CommPrim\_Source\_Text => "write\_Output2";

};

### Interrupt Ports and Managing ISRs

The current AADL specification does not really define how interrupts are to be managed for code generation. In the trusted build tool, we have chosen to implement them as a special port type, designated by a set of additional properties. This port is designed to be left unconnected, and will trigger an error during build if it is connected to a output port.

Following standard practice, we split interrupt handling into first and second level handlers. In each of the operating systems that we target, the first-level handler runs immediately in a special execution context (for eChronos or VxWorks, the context of the current thread; for CAmkES, a special high-priority thread in the component of interest). The second level handler runs through the standard dispatch mechanism of the designated thread.

For trusted build, it is possible to make a thread an ISR handler by adding an ISR port, which is a specialized input event port. It is most straightforward to present using an example:

Input1: **in** **event** **port** {

Source\_Text => ("user\_code/user\_sender.c",

"user\_code/qemu\_clock\_driver.c",

"user\_code/clock\_driver.h");

SMACCM\_SYS::Is\_ISR => **true**;

SMACCM\_SYS::First\_Level\_Interrupt\_Handler => "timer\_flih";

Compute\_Entrypoint\_Source\_Text => "timer\_slih";

SMACCM\_SYS::Signal\_Name => "systick";

SMACCM\_SYS::Memory\_Pages => ("mem", "0x53F98000:0x1000");

SMACCM\_SYS::Signal\_Number => 27;

SMACCM\_SYS::Sends\_Events\_To => "{{2 Output1}}";

};

An ISR port is designated by the AADL property Is\_ISR; it can only be associated with an input event port. It supports several additional properties that can be used to configure the hardware to support interrupt processing, as described below.

First\_Level\_Interrupt\_Handler: string   
This property provides the name of the first-level interrupt handler to run in response to the interrupt.

Compute\_Entrypoint\_Source\_Text : string  
SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text : list of string  
For ISR ports, these properties define the second-level interrupt handlers. They are managed the same way as any other dispatcher for an input port.   
  
SMACCM\_SYS::Signal\_Name : string;  
SMACCM\_SYS::Signal\_Number : int;

Depending on the OS, interrupts can be identified by name, number, or both. This property describes the ISR in terms of the interrupt signal name. Consult the relevant OS/HW guide for the appropriate mechanism for identifying the signal. Both the name and number properties are *optional* for certain OS/HW combinations and *required* for others (though one or the other is always *required*).

SMACCM\_SYS::Memory\_Pages : list of string ;

This property is really a list of string pairs that each define a name / memory region that is necessary for the ISR. These are used to describe the hardware memory regions that provide data / read data for the ISR, usually corresponding to some memory mapped IO region. A memory region is defined by a base address and size: "0x53F98000:0x1000". The location and size are separated by a colon ‘:’, so the example above starts at 0x53F98000 and is 4k (0x1000) bytes long.

Source\_Text: list of string   
SMACCM\_SYS::Sends\_Events\_To: aadlstring

These elements have the same definition as for other ports.

### Discussion

It is not requiredthat input event ports have entrypoints. If they do not have entrypoints, then they do not trigger a dispatch. The queued values for the input port can be read by another entrypoint (say, a periodic dispatch entrypoint).

ISR threads are simply “normal” threads that have an ISR input port. The second level interrupt handler dispatch is handled like any other dispatch. Therefore it is possible to have a hybrid thread that both responds to interrupts and is periodically dispatched. This can be useful for (say) watchdog timers related to hardware, but should be used with care, as it can be difficult to analyze for schedulability.

## Shared Data Access

The trusted build tool supports shared data as a “raw” use of OS resources. For port-based communication, the middleware uses mutexes to ensure that communication between threads is free from race conditions and deadlocks. Shared data provides fast access to shared data buffers between threads, but the application is responsible for ensuring that data is consistent in the face of potential multi-threaded reads/writes of the shared data. This consistency can be enforced through the use of user-defined mutexes or through some application-defined signaling scheme.

Shared data is defined at the process level, and accessed at the thread level.

### Example

**process** **implementation** proc.Impl

**subcomponents**

s: **thread** sender;

r: **thread** receiver;

shared: **data** a\_struct\_wrapper.impl {

SMACCM\_SYS::CAmkES\_Owner\_Thread => "sender";

};

**connections**

conn1: **data** **access** shared -> s.buff1;

conn2: **data** **access** shared -> r.buff2;

**end** proc.Impl;

This example shows how shared data is facilitated by process declarations. Note the CAmkES\_Owner\_Thread declaration: this is an example of OS specific information that requires some management at the AADL level. In the VxWorks and eChronos operating systems, it is possible to construct shared memory simply by declaring the memory in a system visible location, so this declaration is unnecessary. However, because of the memory management scheme used by CAmkES, it is necessary to declare an “owner” thread where the memory will be resident.

To use the shared data, one declares threads with read or write access to the region:

**thread** sender

**features**

buff1 : **requires** **data** **access** a\_struct\_wrapper.impl

{

Access\_Right => READ\_WRITE;

};

**properties**

// thread properties here…

**end** sender ;

-- Example of a periodically dispatched "Active" thread.

**thread** receiver

**features**

buff2 : **requires** **data** **access** a\_struct\_wrapper.impl

{

Access\_Right => READ\_WRITE;

};

**properties**

// thread properties here…

**end** receiver ;

The AADL code generator provides a named pointer to the shared memory region whose name matches the declaration name used for the data access.

## Shared Subprogram Declarations

The trusted build system supports the use of “raw” RPCs between threads. To perform RPCs, one declares subprograms and subprogram groups that can be invoked by other threads. An example is shown in Figure 3. The example is excerpted from /models/Trusted\_Build\_Test/test\_rpc\_native.addl.

**subprogram** add\_two\_numbers

**features**

A: **in** **parameter** Base\_Types::Unsigned\_32;

B: **in** **parameter** Base\_Types::Unsigned\_32;

result: **out** **parameter** Base\_Types::Unsigned\_32;

**end** add\_two\_numbers;

**subprogram** subtract\_two\_numbers

**features**

A: **in** **parameter** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::By\_Reference => **true**;

};

B: **in** **parameter** Base\_Types::Unsigned\_32;

result: **out** **parameter** a\_struct.impl;

**end** subtract\_two\_numbers;

**subprogram** **group** add\_subtract\_interface

**features**

add: **provides** **subprogram** **access** add\_two\_numbers;

subtract: **provides** **subprogram** **access** subtract\_two\_numbers;

**end** add\_subtract\_interface;

**thread** sender

**features**

add\_subtract: **requires** **subprogram** **group** **access** add\_subtract\_interface ;

**properties**

-- several properties from the “sender” thread

**end** sender ;

**thread** receiver

**features**

add\_subtract: **provides** **subprogram** **group** **access** add\_subtract\_interface;

**properties**

-- several properties from the “receiver” thread.

**end** receiver ;

**process** proc

**end** proc;

**process** **implementation** proc.Impl

**subcomponents**

s: **thread** sender;

r: **thread** receiver;

**connections**

s\_to\_r : **subprogram** **group** **access** s.add\_subtract -> r.add\_subtract;

**end** proc.Impl;

Figure 3: Example using RPCs

AADL provides a standardized interface to call subprograms AADL thread boundaries. AADL’s subprogram parameter mechanism is similar to Ada’s parameter mechanism. Parameters can either be *in, out,* or *inout.* By default inputs are pass-by-value and outputs are pass-by-reference. It is possible to override the input mechanism by setting the SMACCM\_SYS::By\_Reference parameter to ‘true’.

The thread providing subprogram group access uses the “provides” declaration and the user of the subprogram group uses the “requires” form.

In the case of VxWorks and eChronos, these just become procedure calls (there is no separate address space) while in CAmkES/seL4 these are translated to remote calls to procedures (interfaces) between components.

AADL subprograms do not allow return values to be specified for subprograms, so as a convention, if the parameter list contains an out parameter called ‘result’, the trusted build tool will map this parameter to the return value for the native code function. Otherwise, the subprogram will be mapped to a function returning ‘void’.

## Threads

Threads are central to the trusted build code generation approach. Depending on the target operating system, AADL threads are bound to *tasks* or (for CAmkES) *components*. Each thread has a number of input and output *ports* that comprise its interface, and a *dispatch protocol* that defines when it will execute. For threads that are periodically dispatched, the user should define an *entrypoint* function that will be called when the period elapses.

### Properties

The following properties are used during code generation:

Compute\_Entrypoint\_Source\_Text : string   
This property is the standard AADL dispatch property; it provides the name of the function to be invoked when a periodic dispatch occurs. If the dispatch protocol is Sporadic or Aperiodic, this property is ignored.

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 Sporadic or Aperiodic, this property is ignored.

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.

Dispatch\_Protocol : {Periodic, Sporadic, Aperiodic, Hybrid }

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), Sporadic, 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.

SMACCM\_SYS::External\_Mutex\_List: list of aadlstring  
SMACCM\_SYS::External\_Semaphore\_List: list of aadlstring  
If a thread wishes to use “raw” RPCs or shared memory access, it can request mutexes or semaphores to be created on its behalf. The developer provides the list of names, and the middleware creates the resources for the user with the specified names. Semaphores are created with an initial count of 1. Note that the interactions with mutexes/semaphores are OS-specific; this property only ensures that they are created. This feature is only for use by advanced users.

External\_Semaphore\_List: list of aadlstring applies to (thread, system implementation);

Initialize\_Entrypoint\_Source\_Text : string  
This optional property provides the name of the initializer function for the thread.

SMACCM\_SYS::Is\_External : Boolean  
This property affects the code generation process for the thread. It is to allow integration with threads that do not follow the AADL scheduling protocols, or require control of the thread *main* function, which is usually controlled by the middleware. **The trusted build system** **does not generate middleware or OS configuration for external threads**, and it is expected that an external thread will follow the conventions of the aadl-generated communications functions. External threads are only for advanced users.

Period : Time

For periodic and hybrid threads, the period defines the dispatch interval for the periodic entrypoint. For sporadic threads, it defines the minimum inter-arrival time between events.

Priority : aadlinteger

This parameter defines the priority of the thread. Higher numbers correspond to higher precedence for threads.

SMACCM\_SYS::Sends\_Events\_To: aadlstring

This property is used for schedulability analysis and communication with monitors. It is a list of sets, where each set gives a list of ports and the maximum number of messages that will be emitted to that output port. For example: given the declaration SMACCM\_SYS::Sends\_Events\_To => "{ {1 Output1}, {1 Output2, 2 Output3} }"; for a single dispatch, the periodic dispatcher could emit one output message to port Output1 or, alternately, one output message to port Output2 and two messages to port Output3.  
  
If monitors are used, then this property is *required*. Otherwise, it is *optional.*

Source\_Stack\_Size : Size

This parameter defines the stack size for the thread (usually measured in bytes).

Source\_Text: list of string   
This property provides the names of the source / header files to be associated with the thread. At minimum, this list should contain the files containing the entrypoint function(s) in Compute\_Entrypoint\_Source\_Text. Files that are listed here will automatically be copied to the OS-specific location necessary for system build; it is possible to add additional files into components outside the AADL build system, but it is the developer’s responsibility to ensure these are copied to the appropriate OS-specific location.

Thread\_Type: enumeration (Active, Passive) => Active

This property is used to construct *monitors,* discussed in Section YYY below. A monitor is denoted by a *Passive* thread.

### Example

An example of a thread declaration is as follows:

**thread** Thread\_B

**features**

datap: **in** **data** **port** dt.rec;

signalCh: **in** **event** **port**

{

SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text => ("ping\_received");

Queue\_Size => 8;

};

**properties**

Dispatch\_Protocol => Sporadic;

Period => 30 ms;

Source\_Stack\_Size => 4096 bytes;

Source\_Text => ("src/example\_task.c");

Initialize\_Entrypoint\_Source\_Text => "init\_barSourceTask";

Compute\_Execution\_Time => (0ms .. 5ms);

**end** Thread\_B;

This example shows a thread that is sporadically, rather than periodicially, dispatched. It is dispatched by a signal that arrives on the signalCh input port. By setting the Period to 30, we are stating that the minimum inter-arrival time of inputs on the signalCh inputs to be 30 ms. We will discuss input ports in more depth in Section 5.4.

### Discussion

Threads are the “workhorses” of the trusted build system. For each thread, middleware code is generated, with name smaccm\_<thread\_name>.c, and smaccm\_<thread\_name>.h, to manage the thread. As will be discussed in Section 6, the middleware provides a framework that the user-level code can “plug into” to provide entrypoints that will be invoked at the appropriate time.

Legacy or special purpose code may not be compatible with the middleware framework. For those instances, we provide *external* threads, which are expected to follow AADL conventions for communication but are externally managed. As these are outside the control of the middleware, care must be taken to ensure that they are “well behaved” and that the system is schedulable in the presence of these threads.

## Monitors

An important concept in concurrent programming is that of *monitors*. These involve a set of procedures that operate over some amount of shared state. Different monitor concepts are supported in a range of programming languages including Java and C#. They are also the main abstraction in CAmkES, and are essential to the Ivory/Tower configuration of tasks, in which a relatively small number of threads communicate through a large number of fine-grained monitors that manage shared resources.

AADL does not have a built-in notion of monitors. In the trusted build tool, we have added a monitor concept to AADL called *passive threads*. Passive threads are declared as threads but do not have their own activation thread; instead, they execute in the context of another thread in response to some input stimulus. The input stimuli that can trigger a monitor are RPCs, or messages at an input event or input event data port. A *passive thread* is declared as an AADL thread but it has an additional property: Thread\_Type => Passive. See the example in Section 5.6.1 for a complete example.

Monitors can use all of the standard AADL communication primitives (event, event-data, dataport, shared data, and RPC), and have some special support in the trusted build tool to improve schedulability analysis and to prevent deadlock. When port-based communication mechanisms are used, the outgoing messages are *queued* until the completion of the monitor. This is done for two reasons: first, monitors are a *shared resource*, so their execution must be considered *blocking time* for the calling thread if a lower priority thread may be using the monitor. By queueing the outgoing results in thread-local queues, the maximum blocking time is that of the shared monitor, rather than the *chain* of monitor calls. The idea is demonstrated in Figure 4.

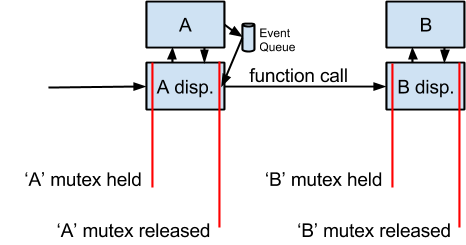


Figure 4: Locking and queueing scheme used for AADL monitors.

The use of *passive threads* in AADL allows a sequence of computations to be initiated by an active thread and then several computations over shared resources can be performed by monitors. An example of this is shown in Figure 5. In this figure, we have two threads (*A* and *B*) and four monitors (*C, D, E,* and *F*) that all communicate using event-data ports. The event-data port from *A* has *fan-out* to *C* and *D*. The black lines describe the static connections in the AADL model and the red and blue lines, respectively, describe the sequence of computations that occur when the threads execute.

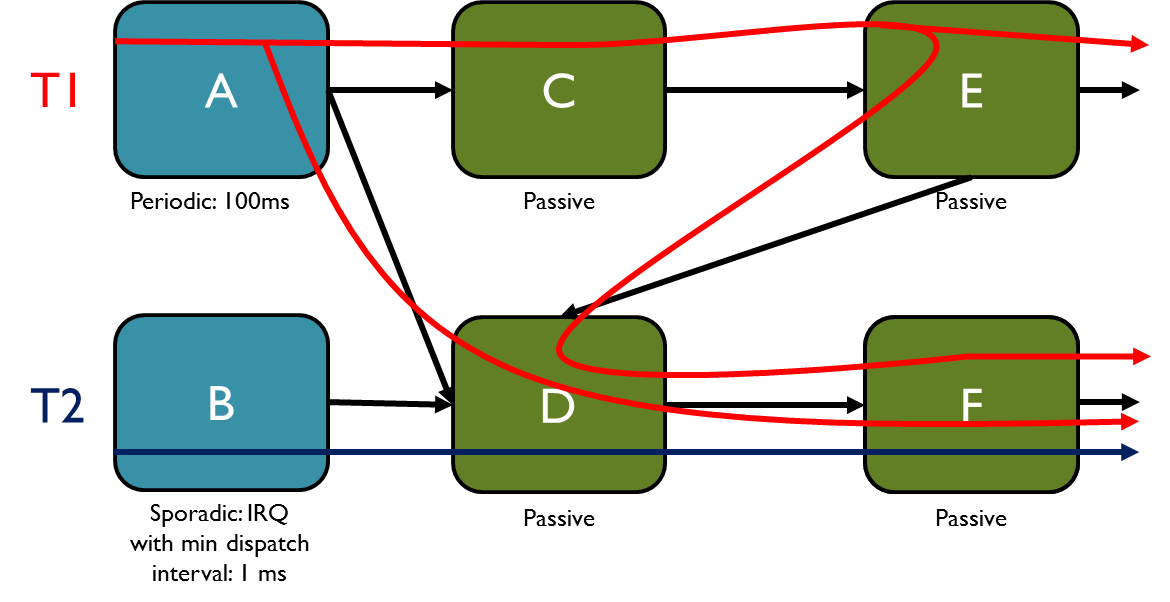


Figure 5: AADL architecture involving two threads and four monitors.

Depending on the desired architectural interactions, monitors can describe systems that are both highly concurrent and utilize a large number of shared resources.

### Example

An example system which contains one active and two passive threads is shown in Figure 6. This example can be found in the SMACCM github repository under /models/Trusted\_Build\_Test/test1

**package** test1

**public**

**with** Base\_Types;

**with** SMACCM\_SYS;

-- Example of a periodically dispatched "Active" thread.

**thread** sender

**features**

Output1: **out** **event** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::CommPrim\_Source\_Text => "snd\_Output1";

};

**properties**

Dispatch\_Protocol => Periodic;

Period => 2 sec;

Source\_Text => ("user\_code/user\_sender.c");

SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text => ("periodic\_ping");

SMACCM\_SYS::Requires\_Time\_Services => **true**;

Priority => 10;

Stack\_Size => 4 KByte;

SMACCM\_SYS::Thread\_Type => Active ;

Compute\_Execution\_Time => 10 us .. 50 us;

SMACCM\_SYS::Sends\_Events\_To => "{{1 Output1}}";

**end** sender ;

-- Example of an aperiodically dispatched "passive" thread.

**thread** receiver

**features**

Input1: **in** **event** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text => ("ping\_received");

Source\_Text => ("user\_code/user\_receiver1.c");

SMACCM\_SYS::Sends\_Events\_To => "{{1 Output1}}";

};

Output1: **out** **event** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::CommPrim\_Source\_Text => "rec\_Output1";

};

**properties**

Dispatch\_Protocol => Aperiodic;

SMACCM\_SYS::Thread\_Type => Passive ;

Compute\_Execution\_Time => 10 us .. 50 us;

**end** receiver ;

**thread** receiver2

**features**

Input1: **in** **event** **data** **port** Base\_Types::Unsigned\_32 {

SMACCM\_SYS::Compute\_Entrypoint\_Source\_Text => ("ping\_received");

Source\_Text => ("user\_code/user\_receiver2.c");

};

**properties**

Dispatch\_Protocol => Aperiodic;

SMACCM\_SYS::Thread\_Type => Passive ;

Compute\_Execution\_Time => 10 us .. 50 us;

SMACCM\_SYS::Sends\_Events\_To => "{{}}";

**end** receiver2 ;

**process** proc

**end** proc;

**process** **implementation** proc.Impl

**subcomponents**

s: **thread** sender;

r1: **thread** receiver;

r2: **thread** receiver2;

**connections**

s\_to\_r1 : **port** s.Output1 -> r1.Input1;

s\_to\_r2 : **port** s.Output1 -> r2.Input1;

r1\_to\_r2 : **port** r1.Output1 -> r2.Input1;

**end** proc.Impl;

**system** test1

**end** test1;

**system** **implementation** test1.impl

**subcomponents**

tl: **process** proc.Impl;

**properties**

SMACCM\_SYS::OS => CAmkES;

SMACCM\_SYS::HW => QEMU;

**end** test1.impl;

**end** test1;

Figure 6: Example containing both Active and Passive Threads

In this example, the *active* thread (sender) sends a message two receiver threads, *receiver* and *receiver2*. The *receiver* thread also sends a message to *receiver2*.

### Implementation Efficiency

Monitors allow very efficient use of shared resources on eChronos and VxWorks. On CAmkES, the current implementation for message queueing is unfortunately relatively inefficient: queues are passed as in\_out parameters to RPC calls, which involves copying using the CAmkES RPC semantics. Therefore, if large amounts of data are being communicated, this may be problematic for schedulability. We are investigating an improved implementation using thread-specific dataports, but this has not yet been implemented.

## Processes

A Process in AADL describes a protected memory space. A process contains *threads*, and *connections,* which are mechanisms to connect thread *ports* to each other. A process may additionally have an interface consisting of ports, so thread outputs can propagate to process outputs and process inputs can be routed to thread inputs.

### Properties

There are currently no properties supported for processes in the trusted build tool.

### Example

An example process and process implementation is shown below.

**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;

The trusted build tool supports both “fan in” and “fan out” communications between ports; however, “fan in” communications can lead to difficulties in schedulability analysis

**process** proc

**end** proc;

**process** **implementation** proc.Impl

**subcomponents**

s: **thread** sender;

r: **thread** receiver;

shared: **data** a\_struct\_wrapper.impl {

SMACCM\_SYS::CAmkES\_Owner\_Thread => "sender";

};

**connections**

conn1: **data** **access** shared -> s.buff1;

conn2: **data** **access** shared -> r.buff2;

**end** proc.Impl;

The second example shows how shared data is facilitated by process declarations (for more information on shared data declarations, see Section 5.5). In the VxWorks and eChronos operating systems, it is possible to construct shared memory simply by declaring the memory in a system visible location. Because of the memory management scheme used by CAmkES, it is necessary to declare an “owner” thread where the memory will be resident.

### Discussion

The support for processes in the current trusted build tool is rudimentary. This is because none of the target operating systems have a concept of a process that is distinct from a thread. In eChronos and VxWorks, there is no memory protection provided by the OS, so the concept of a process truly does not exist. On the other hand, in CAmkES/seL4, each thread operates in its own protected memory space, so once again the notion of process is superfluous. For the purposes of trusted build, a process is merely an amalgam of threads. It is possible to structure one’s system as a set of communicating process or as a system containing a single process and all threads and the generation process and result will be the same.

For shared data, in the trusted build tool, processes define the shared data to be accessed by multiple threads.

## System Implementation AADL Declarations

The skeleton and glue code for eChronos is emitted from a top-level AADL system implementation. This system can contain additional declarations to support specific features of the operating system or to include “legacy” code whose thread and communication structures are not described in AADL into the build process.

### Properties

OS : enumeration (eChronos, CAmkES, VxWorks)   
The OS property describes the target operating system of the AADL file.

HW : enumeration (PIXHAWK, ODROID, x86, QEMU)

The HW property describes the target hardware platform for the AADL file. The currently supported platforms are *PIXHAWK –* an ARM7 core*, ODROID –* an ARM15 core*, x86 –* a generic x86 platform*,* and *QEMU –* for building against the QEMU virtual machine*.*

**[MWW: Current Stopping Point]**

# Function Signatures in C

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.

**NOTE:** In the current version of aadl2rtos, we do not support multiple thread instances for a given thread implementation. This could be added without substantial difficulty, but the Ivory/Tower backends would have to be modified, so this is not currently planned for the HACMS project.

**NOTE:** for a given thread or monitor, all functions that the user is expected to provide or call with respect to the AADL middleware are defined in a header file emitted by the generation process. Given a thread of name <thread\_name>, the generated header file containing the trusted build-related functions is smaccm\_<thread\_name>.h.

## Port Function Signatures

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

bool sample\_write\_function(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. The return value determines whether or not the read succeeds. Data port writes will always succeed, and event queue writes will succeed if the message buffer is not full.

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

bool sample\_read\_function(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.

## Thread Function Signatures

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

void sample\_initialize\_thread();

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();

# Schedulability Analysis

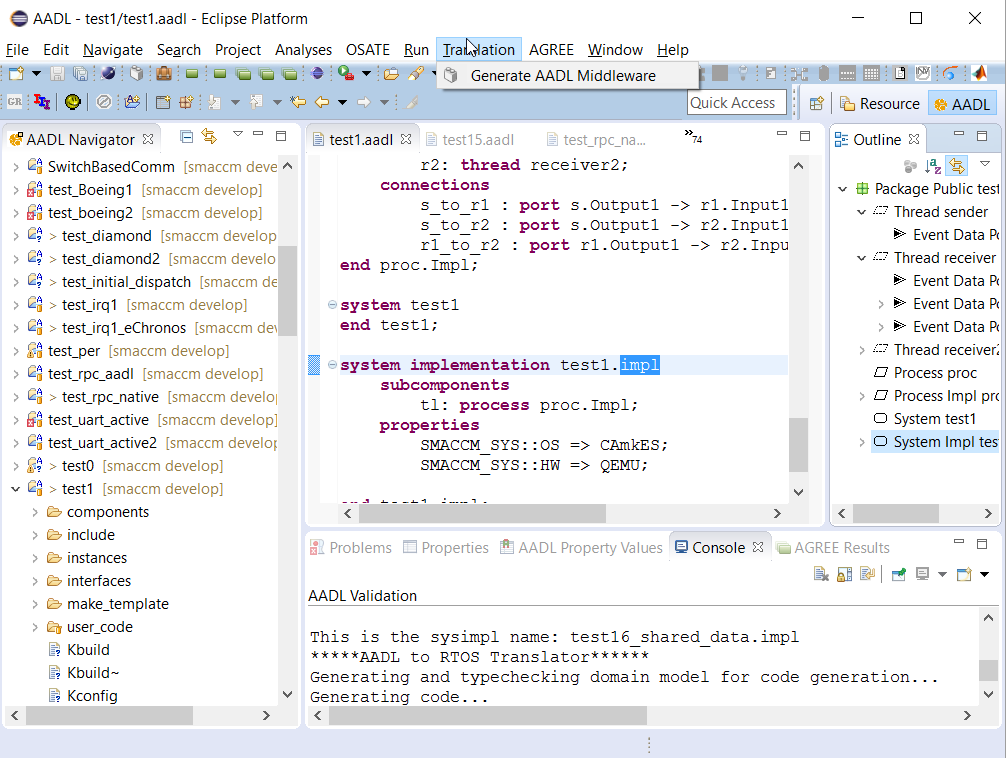
TBD

# Install Process

The install instructions are maintained in the SMACCM github repository in the README.md file.

# Build Process

To invoke aadl2rtos, all that is required is to choose the top-level system implementation that will form the “root” of the generated implementation, then choose the “Generate AADL middleware” menu item from the “Translation” menu in OSATE:



If successful, by default, the system will generate the middleware files in an OS specific location, as explained below. If errors occur, they will be displayed in the console. Usually errors occur because a property required by the build process has not been added to an AADL component that is under the root-level system implementation. For example, if a thread does not have a priority property, the system will return an error.

## eChronos

For eChronos, all files are generated in a subdirectory entitled ‘gen’ that is a child directory of the top-level AADL file, and will generate a eChronos .prx configuration file and Makefile in the same directory as the top-level AADL file. The .prx file contains the information required to generate the eChronos operating system implementation, and after the OS implementation is generated, the Makefile can build the system implementation. For more information on the eChronos build process, please see the eChronos documentation.

## CAmkES

For CAmkES, a set of CAmkES *components, interfaces,* and an *assembly* fileare generated corresponding to the AADL threads and monitors, processes, and system implementation. These components are stored in the directory structure as expected by CAmkES. An example is shown below:

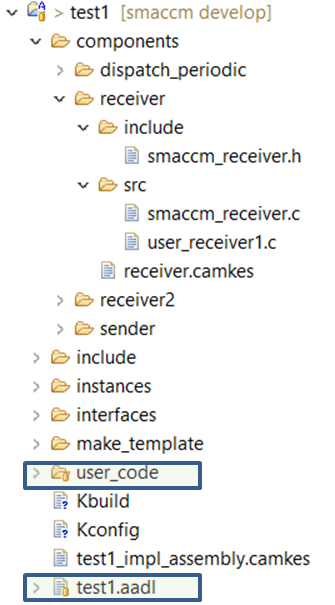


Figure 7: generated directory structure for the CAmkES back end.

This tree shows the generated code for the *test1* example. The files in boxes are provided by the user, and the other files are generated by the AADL middleware.

In the test1 example, there are three threads that are defined in the AADL model: *sender, receiver*, and *receiver2*. In order to manage periodic scheduling, the trusted build system constructs a periodic dispatcher component (dispatch\_periodic).

The layout of a component is shown for the *receiver* component. It includes an *include* directory with the autogenerated *smaccm\_receiver.h* file. This file describes the interface of interaction between user code and the middleware code. The *smaccm\_receiver.c* file in the src directory provides the implementation of all middleware functions. The *user\_receiver1.c* file is a user-provided file that is automatically copied into this location as part of the build process. In this particular example, it was originally stored in the user\_code directory, but the location for user code can be anywhere within the file system.

The *include* directory contains include files that are relevant to all components: this is where the header containing all AADL data types is stored. The *interfaces* directory describes the RPC interfaces for each of the components that will be used for CAmkES. The *make\_template* directory contains a template make file that is sufficient for models that do not contain external code. For more complex make processes, this file can be modified.

**NB:** In order to use the generated make file, it must be copied from the *make\_template* directory to the top-level directory. This was a conscious decision to prevent the trusted build system from accidentally overwriting a custom make file.

The *test1\_assembly.camkes* file describes the top-level system architecture from the CAmkES perspective, and the *Kbuild* and *Kconfig* files provide necessary information to the CAmkES build system.

For more information on building CAmkES applications, please see the Data61 CAmkES documentation.

## VxWorks

**TBD.**