AADL2RTOS Design Document

Version 2.0

Mike Whalen

University of Minnesota

1. Questions: Are all CAmkES interfaces point-to-point?
2. How is memory allocated for out parameters? Is it the responsibility of the caller or the callee?
3. How are in parameters passed? By value or by reference?
4. Is access to all “procedures” provided by a component serialized? In other words, suppose that a component provides procedure x and procedure y. If a function f from procedure x is dispatched, is it possible to dispatch function g from procedure y before f is completed? Another way of asking this is: does each procedure have its own thread? If so, how do we assign the priorities of the different component threads?

# Translating AADL User-Defined Types

# Translating AADL Threads

## User-Level API for Component Communication

## User-Level API for Thread Dispatchers

## Implementation in CAmkES

### AADL Thread to CAmkES component

Each AADL thread has several associated generated files in CAmkES. It makes changes to the file system in the following ways:

<TopLevelSysName>/

interfaces/

smaccm\_<ThreadImplName>\_dispatch\_types.h

smaccm\_<ThreadImplName>\_dispatchers.idl4

… files for other threads and user-defined types here…

components/

<ThreadImplName>/

smaccm\_<ThreadImplName>.camkes

smaccm\_<ThreadImplName>.h

smaccm\_<ThreadImplName>.c

…additional user-created files for the component here…

The interface directory additions are related to types introduced by the thread, and the components/<ThreadImplName>/ directory contains the files that describe the implementation of the component.

#### Thread Implementation .idl4 interfaces and types

AADL threads are mapped to CAmkES components in terms of their incoming dispatchers. A thread is dispatched either because of an incoming event (which can be an IRQ) or because its period has elapsed. A thread has a single CAmkES ‘procedure’ (interface) that contains all of its dispatchers that is stored in the ‘interfaces’ directory. The name of the interface will be <ThreadImplName>\_dispatchers.idl4

Associated with each procedure is a header file that defines the types for the outgoing information for each dispatch function (as defined below). The header will have the name <ThreadImplName>\_dispatch\_types.h

The interface for a dispatch function is as follows: it has an input that is mapped to the type of dispatcher: for a periodic dispatcher, the input contains the current system time in milliseconds. For an event data port dispatcher, the input matches the type of the incoming data. For an event port, there is no input. The dispatcher also has outputs representing the messages that may be generated by the component. For each output event port there is one output that represents the number of output messages emitted on that port. This output will be named <PortName>\_count. For each output event-data port, there are two output parameters: one integer that represents the number of emitted messages (named <PortName>\_count) and another that is an array of the specified data type: smaccm\_<ThreadImpl>\_<Dispatcher>\_<PortName>\_type that is named <PortName>\_data. The order of output parameters will match the order of the output ports in the file. The size of the arrays will match the maximum number of dispatches possible for these ports as specified in the AADL “Sends\_Events\_To” property. The array types will be specified in the <ThreadImplName>\_dispatch\_types.h file.

For each datatype used in a read/write or shared memory interface, we create a smaccm\_rw\_<type> interface containing read() and write() functions for the data type. Similarly, for non-dispatch queued data, we create these interfaces.

Native RPC interfaces will also be supported. For “requires” subprogram group access from AADL, an equivalent RPC procedure will be ‘used’. For “provides” access, an equivalent RPC procedure will be provided. The types and procedures for the subprogram group will be automatically generated. The use of these subprogram groups will be entirely up to the implementer. Note especially that for “provides” accesses, these accesses occur on a separate thread and care must be used to ensure correct use of shared resources.

#### Thread Implementation CAmkES components

The CAmkES component implementing a thread owns (provides) its dispatcher interface and the shared data interface for any data ports that it reads from. It uses the shared data interface for any data components that it writes to. In the case of fan-out, several interfaces may be “used” for the same outgoing port; all CAmkES communications are point-to-point.

#### Shared Data CAmkES components

AADL Shared data is represented by its own component. The component provides a smaccm\_rw\_<type> interface containing the read() and write() functions for the data type.

### Active Thread Dispatchers

For “active” threads, a dispatcher component will be constructed. This component manages the event-based interactions of the components to minimize the possibility of deadlock and the duration of resource contention within the system.

The dispatcher will maintain a data queue for each event/data interface associated with the active thread.

The queue will be accessed through a procedure that adds/removes data from the queue (the standard interface for non-dispatch components). The only slight change in the implementation for active thread queues is that it signals the control thread (via a semaphore) that a message is available.

The control thread wakes up when it is signaled and attempts to drain all message queues. Given a message, it calls the appropriate dispatcher within the associated “passive” component.

The CAmkES dispatcher component uses all passive components that could ultimately be dispatched by it.

### CAmkES Assembly

#### Composition

The assembly in CAmkES describes the top-level composition of the components within the model. It consists of a listing of the components used and their connections. In the current AADL translation, we currently expect to use only sel4RPC connectors.

Each dispatcher is connected to all passive component dispatchers to which it dispatches.

All reader/writer interfaces are directly connected between passive components.

Dispatchers connect to one another through queued send/receive interfaces.

#### Configuration

Each active thread is assigned a priority

All passive threads are assigned a nominal priority that will ensure interrupt from active thread (lower priority).

Any hardware-specific information will be added.