AADL Modeling Guidelines for CASE

July 20, 2020

Background

The Architecture and Analysis Design Language (AADL) has been engineered as a general-purpose system architecture modeling language. As a result, the language specification does not necessarily dictate the semantics of how the modeling artifacts in AADL are mapped to actual physical artifacts in the end systems. The way in which AADL-based analysis and code-generation tools interpret the language's modeling artifacts are domain specific, and are left to the tool developers.

The purpose of this document is to define a set of modeling guidelines for producing well-formed AADL models for use in the Collins CASE toolchain. The CASE toolchain is extensible, but is currently comprised of the following tools and technologies:

- **Cyber Requirements (TA 1)**
  - GearCASE (Charles River Analytics)
  - DCRYPPS (Vanderbilt / DOLL Labs)
- **Cyber Resiliency (TA 2)**
  - StairCASE (Collins)
  - AGREE (Collins)
  - Resolute (Collins)
  - SPLAT (Collins)
- **Formal Methods (TA 4)**
  - Sally (SRI)
- **Integration and Build (TA 5)**
  - BriefCASE (Collins)
  - HAMR (Kansas State University / Adventium)
  - CAmkES (Data 61)
  - seL4 (Data 61)

Due to the importance of preserving data flow contracts between the design and implementation, this document also details how the CASE system build toolchain, specifically HAMR (High-Assurance Modeling and Rapid Engineering for Embedded Systems) AADL-to-CAmkES translator interprets an AADL model and converts it into CAmkES source code, targeted for a specific hardware platform that is ready for

compilation. It is assumed the reader has familiarity with AADL, seL4, CAmkES, and the CASE program.

Checking Compliance with these Guidelines in OSATE

To understand whether a given AADL model complies with these guidelines, the Collins CASE tools include a *CASE_Tools* ruleset that can be used by the Resolint tool in OSATE.

Rules are identified in this document with a unique descriptive identifier a textual representation of the rule, and a problem type in the following format:

Rule **Rule_ID**	Rule	

The problem type is how the rule violation will be classified in the Problem's view of OSATE when Resolint is run on the ruleset. The three problem types are:

	Information
	Warning
	Error

Software Architecture Requirements

HAMR is currently designed to process AADL instance models rooted at a system implementation.[1] The model *must* contain a single processor-bound process that contains one or more thread subcomponents (see the section "Hardware Architecture and Binding Requirements" for an example).

Figure 1 below shows an example diagram from the CASE "Simple UAV" model of a process that contains four thread subcomponents.

[1] From inside Eclipse, the system build can also be generated by selecting a system implementation in the Outline view and invoking the HAMR tool.

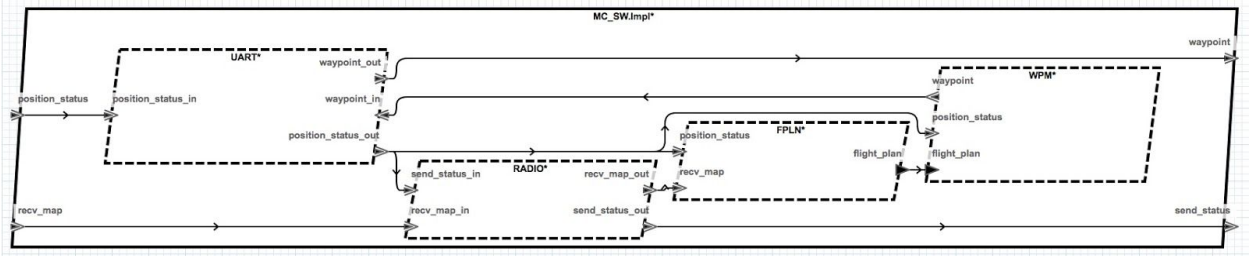

Figure 1: The primary software architecture model of the mission computer application in the "Simple UAV" example developed under the CASE program.

The model may contain additional processes, however HAMR will issue an error if more than one process contains thread or thread-group subcomponents. For example, the instance model rooted at CASE_Simple_Example_V3's UAS.UAS.Impl has two processor bound processes, GS::GS_SW.Impl and SW::SW.Impl, but will be accepted by HAMR as only the latter contains thread subcomponents. Both the `modes` and `end to end flows` capabilities in AADL are not currently supported.

rule **one_process**	Only one processor-bound process can contain thread or thread-group subcomponents	⛔

rule **one_thread**	A process must have at most one thread subcomponent.	⛔

rule **modes_ignored**	Modes will be ignored	⚠

rule **flows_ignored**	Flows and end-to-end flows will be ignored	⚠

The table below summarizes the mapping of the primary AADL software artifacts used by the CASE system build: systems, processes, threads, ports, and data components. Additional details are given in subsequent sections.

AADL Component	CAmkES Mapping
System	For the CASE program, system components are modeling artifacts that represent high-level collections of software components that share common hardware bindings. System components may represent an arbitrary decomposition of the software architecture, and are not necessarily directly mapped to seL4 or CAmkES components.
Process	Each process implementation represents a single seL4 instance (one-to-one mapping). Subcomponents defined within an AADL process are thus mapped to components hosted within its own seL4 instance. AADL ports attached to a process represent dedicated communication channels (usually hardware specific I/O) into and out of the seL4 instance.
Thread	Threads are the lowest level of software component in the CASE context. Each thread implementation maps to a single CAmkES partitioned component (one-to-one mapping). The system build toolchain does not support nested thread components. AADL ports attached to a thread represent dedicated communication channels into and out of the CAmkES space-partitioned component.
Port	Communications between CAmkES partitions are modeled in AADL as connections between port subcomponents of threads. The type of port dictates the type of communication implemented in CAmkES. See details under the Connections subsection below.
Data	Data components are associated with data ports, which represent the data types used in the CAmkES implementation. See details below.

Table 1: Summary HAMR AADL to CAmkES Component Mapping.

System Components

The top level implementation of the model must be a system component, in order to use the HAMR CAmkES translation tool. The AADL code sample below shows a top-level system with two system subcomponents.

```
system implementation UAV.Impl
      subcomponents
            MCMP: system MC::MissionComputer.Impl;
            FCTL: system FC::FlightController.Impl;
            SBUS: bus Serial.Impl;
      end UAV.Impl;
```

Thread Components

The AADL code example below shows an example of a process implementation with four thread subcomponents.

```
process implementation MC_SW.Impl
      subcomponents
            RADIO: thread RadioDriver.Impl;
            FPLN: thread FlightPlanner.Impl;
            WPM: thread WaypointManager.Impl;
            UART: thread UARTDriver.Impl;
end MC_SW.Impl;
```

The dispatch behavior of a thread can be specified using the *Thread_Properties::Dispatch_Protocol* property. HAMR currently supports only *Periodic* or *Sporadic* threads. If the dispatch protocol property is not provided then the thread is treated as sporadic and a warning will be issued. HAMR will issue an error if a dispatch protocol other than periodic or sporadic is specified.

rule **dispatch_protocol_specified**	Threads should have the dispatch_protocol property specified	

rule **valid_dispatch_protocol**	Threads can only specify a dispatch_protocol property of *periodic* or *sporadic*	

rule **complete_periodic_protocol**	If a thread has a Dispatch_Protocol property value of "Periodic" then it must have a valid Period and Compute_Execution_Time property values set. Also, if the thread subcomponents of a process are specified as Dispatch_Protocol "Periodic" then the process must have a seL4_Properties::Domain property value specified.	

rule **consistent_dispatch_protocol**	For all thread subcomponents of processes bound to the same processor (via the Actual_Processor_Binding property), the Dispatch_Protocol property value assigned to the threads must be identical. For example, if one thread bound to a processor is "Periodic"	

	then all threads bound to that processor must be "Periodic".	

Data Components

AADL data components can be used to specify the types of AADL features such as data ports. AADL includes a *Base_Types* package that provides data component declarations for basic types like signed/unsigned integers, floating-point numbers, booleans and strings. HAMR provides translation support for each of these, mapping them to appropriate C data types, except for the unbounded *Base_Types::Integer* and *Base_Types::Float* types. HAMR will issue an error if these two unbounded types are used. E.g., the subcomponent SW::Coordinate.latitude will cause HAMR to issue an error[2].

rule **bounded_integers**	Integer types must be bounded (cannot use Base_Types::Integer)	

rule **bounded_floats**	Float types must be bounded (cannot use Base_Types::Float)	

AADL allows data type classifiers to be left unspecified in the instance model, for example the data type classifier of a data port. In such cases, HAMR will use a placeholder classifier called *MISSING_TYPE* and issue a warning. For example, HAMR will attach the *MISSING_TYPE* classifier to SW::WifiDriver.gimbal_command

rule **data_type_specified**	Data types should be specified	

User defined structured/record types, array types, and enumeration types can be specified using data components as follows:

Records

HAMR identifies data component implementations that contain data subcomponents as record types (i.e. instead of using the *Data_Model::Data_Representation => Struct* property). HAMR

[2] A rewriter is used during HAMR development to convert *Base_Types::Integer* to *Base_Types::Integer_32* in order to allow non-conforming models to be processed.

will substitute the *MISSING_TYPE* and issue a warning if a subcomponent's type is not provided.

For example, SW::Command.Impl is a valid record type declaration

```
data Map
      -- The Map is a structure that contains a list of coordinates that
      -- encircle a region. In this implementation, we fix the size of
      -- the map to 4 waypoints.
      properties
            Data_Model::Data_Representation => Array;
            Data_Model::Base_Type => (classifier (Coordinate.Impl));
            Data_Model::Dimension => (4);
end Map;

data FlightPattern
      -- The Flight Pattern is an enumeration that defines how
      -- the UAV will fly through the sensing region to conduct
      -- surveillance.
      properties
            Data_Model::Data_Representation => Enum;
            Data_Model::Enumerators =>
                  ("ZigZag", "StraightLine", "Perimeter");
end FlightPattern;

data implementation Command.Impl
      subcomponents
            map: data Map;
            pattern: data FlightPattern;
end Command.Impl;
```

rule **subcomponent_type_specified**	Subcomponent types should be specified	

Arrays

Data components containing the property *Data_Model::Data_Representation => Array* are identified as array types. An array data component *must* contain the *Data_Model::Dimension* property providing the dimensions of the array. HAMR currently supports only one dimensional arrays. HAMR will issue an error if the dimension property is not provided, or if a multidimensional array is specified. The base type of an array can be specified using the *Data_Model::BaseType* property. HAMR will substitute the *MISSING_TYPE* and issue a warning if the base type is not provided.

For example, SW::Map is a valid array type declaration

```
data Map
      properties
            Data_Model::Data_Representation => Array;
            Data_Model::Base_Type => (classifier (Coordinate.Impl));
            Data_Model::Dimension => (4);
end Map;
```

rule **array_dimension**	Array dimensions must be specified	

rule **one_dimensional_arrays**	Arrays can only have one dimension	

rule **array_base_type**	The array base type should be specified	

Enums

Data components containing the property *Data_Model::Data_Representation => Enum* are identified as enumerated types. A non-empty list of enumerators for an enumeration data component must be defined using the Data_Model::Enumerators property. For example, SW::FlightPattern is a valid enumerated type declaration

```
data FlightPattern
      properties
            Data_Model::Data_Representation => Enum;
            Data_Model::Enumerators =>
                  ("ZigZag", "StraightLine", "Perimeter");
end FlightPattern;
```

rule **non-empty_enums**	Enumeration data components must be non-empty	

Connections

HAMR currently only translates connections between threads. Connections between thread components *must* be unidirectional, otherwise HAMR will issue an error.

rule **unidirectional_connections**	Connections between thread components must be unidirectional	⊗

Additional constraints are placed on component ports. Although AADL ports can be both *in* and *out*, HAMR requires ports to be unidirectional.

rule **unidirectional_ports**	Ports must be in or out, but not both	⊗

Furthermore, HAMR does not permit multiple incoming connections to a single port (fan-in).

rule **no_fan_in**	Multiple incoming connections to a single port are not allowed	⊗

A warning will be issued if the model contains ports that are not connected.

rule **ports_connected**	All ports should be connected	⚠

Ports and connections in AADL define the types of CAmkES communications that are deployed between components in the system build. In the simplest constructs, there is a straight one-to-one mapping for the AADL connection to seL4 communication channel: AADL event ports translate into seL4 notifications and AADL data ports translate into seL4 shared data channels.

Refer to the following table:

AADL Port Type	seL4 Communications	CAmkES Description
Event Data Port	Shared Data + Notification	For communications between (non-virtualized) CAmkES components, implemented as a shared data/notification communications pair. The data is queued within the shared memory. The sending component can only write to the shared data port and the receiving component can only read the shared data port.
Data Port	Shared Data	Implemented as a shared data connection, where the sending component can only write to the shared data port and the receiving component can only read the shared data port.
Event Port	Notification	Implemented as an emit/consumes pair.
Data Access	Shared Data	Not yet fully supported
Subprogram Group Access	RPC	Not yet fully supported.

Table 2: Summary of HAMR AADL to CAmkES Connection Mapping.

The Appendix gives examples of each AADL to CAmkES connection mapping, including AADL and CAmkES source code.

Component Behavior

HAMR supports the insertion of behavior code to components in the generated CAmkES output. The files containing user supplied source code for a component can be specified by attaching the file location directly to threads in the AADL model using the `Source_Text` property. The files, if they exist, will be copied into the corresponding CAmkES component's directory.

rule **threads_have_source**	Thread implementations must indicate location of source code or binary	⊗

Alternatively, a directory location can be provided to AHAMRCT and any C-source files contained in the directory will be copied to an *auxiliary code* directory that will be provided to every generated CAmkES component. The names of any functions declared in these source files must be unique across the entire system.

HAMR recognizes the properties in the following table as specifying behavior code. Each property is of string type and will contain the name of a function in the component's source file, which must conform to the corresponding signature (Table 3).

Property Name	Purpose	Applies To	Function Signature
`Initialize_Entrypoint_Source_Text`	Initialize component	Thread	`void` *functionName* `(const int64_t *in_arg);`
`SB_SYS::Compute_Entrypoint_Source_Text`[3]	Event callback	Event Data Port	`void` *functionName*`(const` *portType* `*in_arg);`

Table 3: Properties to Identify Component Behavior Entry Points in HAMR.

For example, the following AADL model was constructed to help illustrate how component behavior can be attached (the generated CAmkES code is available here).

```
thread sender
      ...
      properties
            ...
            Source_Text => ("user_code/user_sender.c");
            Initialize_Entrypoint_Source_Text => "sender_init";
            SB_SYS::Compute_Entrypoint_Source_Text => ("periodic_ping");
```

The entry point for a CAmkES component is a method generated by HAMR called `run`. The `Initialize_Entrypoint_Source_Text` property can be used to specify the method name containing initialization instructions that should be executed when the method is invoked. The required signature for the method is provided in the header file that HAMR generates for the component (e.g., from sb_sender.h). The header file also contains the signatures of the methods that can be used to interact with a component's

[3] SB_SYS is an AADL property set that is provided as an OSATE plugin contribution by HAMR

middleware. The following excerpt shows a portion of the generated C-code for the sender component (full listing is available at sb_sender.c).

```
void sb_entrypoint_sender_initializer(const int64_t * in_arg) {
  sender_init((int64_t *) in_arg);
}

int run(void) {
  CALLBACKOP(
      sb_timer_complete_reg_callback(sb_timer_complete_callback, NULL));
  {
    int64_t sb_dummy;
    sb_entrypoint_sender_initializer(&sb_dummy);
  }
  // Initial lock to await dispatch input.
  MUTEXOP(sb_dispatch_sem_wait())
  for(;;) {
    MUTEXOP(sb_dispatch_sem_wait())
    // Drain the queues
    If (sb_occurred_periodic_dispatcher) {
      sb_occurred_periodic_dispatcher = false;
      sb_entrypoint_sender_periodic_dispatcher(
            &sb_time_periodic_dispatcher);
    }
  }
  return 0;
}
```

After executing the optional initialization block, the method then waits on a dispatching semaphore that is posted at the arrival of external events; e.g., an incoming event for a sporadic thread or the start of a new period for a periodic thread. The names of the methods that should be invoked to handle a particular event can be specified using the `SB_SYS::Compute_Entrypoint_Source_Text` property, which should be attached to the component for periodic threads (e.g. sender), or to an event port for sporadic threads (e.g. receiver).

Hardware Architecture and Binding Requirements

For the CASE program, the hardware specifications in an AADL model are used only as references, and do not directly impact the generated CAmkES output, except that the process in the model targeted for CAmkES implementation must be bound to a hardware processor resource.

rule **processes_bound**	All processes must be bound to exactly one processor or one virtual processor	

Consider the example AADL code below, again taken from the CASE “Simple UAV” model. This example defines the hardware specification for the mission computer subsystem. In its system implementation, the processor binding property, e.g. `Actual_Processor_Binding`, specifies that the software process `PROC_SW` is bound to the hardware processor `PROC_SW`.

```
system MissionComputer
	features
		recv_map: in event data port;
		position_status: in event data port;
		waypoint: out event data port;
		send_status: out event data port;
		UARTA: requires bus access UAV::Serial.Impl;
		RFA: requires bus access UAS::RF.Impl;
end MissionComputer;

system implementation MissionComputer.Impl
	subcomponents
		RADIO_HW: device Radio.Impl;
		UART_HW: device UART.Impl;
		PROC_HW: processor MC_Proc.Impl;
		MEM_HW: memory MC_Mem.Impl;
		BUS_HW: bus MC_Bus.Impl;
		PROC_SW: process SW::MC_SW.Impl;
	connections
		bac1: bus access RADIO_HW.MCA <-> BUS_HW;
		bac2: bus access UART_HW.MCA <-> BUS_HW;
		bac3: bus access PROC_HW.MCA <-> BUS_HW;
		bac4: bus access MEM_HW.MCA <-> BUS_HW;
		bac5: bus access RADIO_HW.RFA <-> RFA;
		bac6: bus access UART_HW.UARTA <-> UARTA;
		c1: port recv_map -> RADIO_HW.recv_map_in;
		c2: port RADIO_HW.recv_map_out -> PROC_SW.recv_map;
		c3: port PROC_SW.send_status -> RADIO_HW.send_status_in;
		c4: port RADIO_HW.send_status_out -> send_status;
		c5: port PROC_SW.waypoint -> UART_HW.waypoint_in;
		c6: port UART_HW.waypoint_out -> waypoint;
		c7: port position_status -> UART_HW.position_status_in;
		c8: port UART_HW.position_status_out ->
			PROC_SW.position_status;
	properties
		Actual_Processor_Binding => (reference (PROC_HW))
```

```
                applies to PROC_SW;
            Actual_Memory_Binding => (reference (MEM_HW))
                applies to PROC_SW;
            Actual_Connection_Binding => (reference (BUS_HW))
                applies to c2,c3,c5,c8;
    end MissionComputer.Impl;
```

Virtual Machine Binding

Special consideration is made for processes that are hosted within virtual machines. The HAMR code generator translates processes bound to virtual machines into a CAmkES infrastructure that configures the virtual machine, its hosted (Linux) instance, and the necessary components that enable communications to and from the virtual machine. The same basic infrastructure is used for both communications between to separate virtual machine instances (within their own separate CAmkES components) and between a virtual machine and a CAmkES component not hosting a virtual machine. The section in the Appendix “Virtual Machine Communications” describes the communication infrastructure in detail.

Consider the following AADL code sample:

```
system implementation top.Impl
    subcomponents
        proc: processor proc.impl;
        vproc: virtual processor vproc.impl;
        vm : process vm_p.impl;
        ping : process ping_p.impl;
    connections
        vm_to_ping : port vm.enq -> ping.deq;
        ping_to_vm : port ping.enq -> vm.deq;
    properties
        Actual_Processor_Binding =>
            (reference (proc)) applies to vproc;
        Actual_Processor_Binding =>
            (reference (vproc)) applies to vm;
        Actual_Processor_Binding =>
            (reference (proc)) applies to ping;
end top.Impl;
```

As the example shows, binding is established from the process `vm` to the virtual processor `vproc` via the property `Actual_Processor_Binding`. Similarly, a virtual machine is represented in AADL as a **virtual processor**, and is bound to a physical

processor using the `Actual_Processor_Binding` property, as the virtual processor `vproc` is bound to the processor `proc` in the example.

HAMR recognizes the binding and automatically generates the build and source code infrastructure to implement the process hosted within the CAmkES virtual machine. The AADL specification allows other modeling approaches to represent hardware-software bindings, but the approach described here is the only approach for representing virtual machines that HAMR supports. For example, in AADL a virtual processor can be bound to a processor by instantiating the virtual processor as a subcomponent of the processor. The CASE tools do not support this representation.

rule **no_processor_subcomponents**	Processor subcomponents may be ignored	

rule **vm_host_one_process**	A virtual processor may host at most one process	

rule **vm_bound_to_one_processor**	A virtual processor may be bound to at most one processor	

rule **vm_no_dispatch_protocol**	A virtual processor should not have the Dispatch_Protocol specified. This property applied to virtual processors is a corner-case allowed by the AADL standard, but is outside the scope of the CASE program	

Timing Isolation through Domain Scheduling

Temporal partitioning can help maintain the desired availability, integrity, and confidentiality of mission-critical information flows. Temporal isolation prevents malicious actors with a foothold in one component from disrupting the operations of another component sharing resources on the hardware platform. In the simplest example, a component infected with malware can steal the processing cycles from another component, and thereby prevent the other component from completing its tasks in a timely manner. Early Integrated Modular Avionics (IMA) efforts established the need

for robust temporal partitioning, and specified several key properties that must be maintained, including resident duration on the processor, rate, latency, and jitter, accounting for context switch overhead, and strict control over cache state. Since then, researchers have identified issues in some scheduling approaches that previously claimed temporal partitioning, such as Rate Monotonic Analysis (RMA).

Our approach employs the domain scheduler in the stock seL4 implementation, and a special "tick-tock" pacing mechanism, built with the existing CAmkES communication infrastructure.

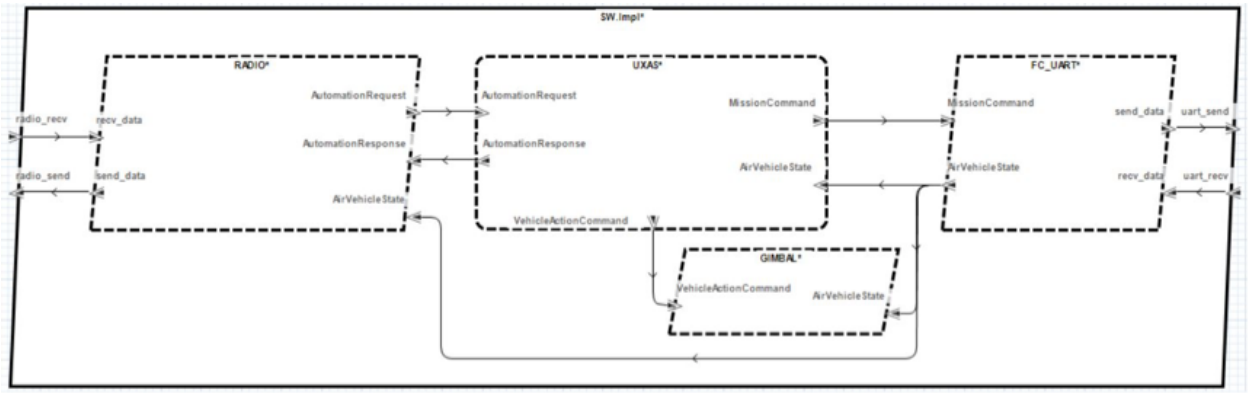

Figure 2: Portion of the CASE Phase 2 demo application

Consider Figure 2, which shows the CASE Phase 2 demonstration modeled in AADL. The application is a basic feed-forward design. The default configuration for this example has no temporal partitioning. In the default configuration of the UAV application without temporal isolation, all components are assigned to domain 0 (which by default receives all the CPU time). Periodic behavior is simulated with a busy-wait loop. Consequently, if one of the execution threads does not relinquish the processor, then the operations of the threads downstream may be disrupted. If the operations are disrupted enough, then the mission fails, or in the worst-case scenarios, the UAV asset is put at risk. Priority mechanisms on threads can alleviate some of these issues, but as described above, are insufficient to provide significant mission resiliency. The busy-wait loop slows down the output for demonstration display purposes; otherwise the processing chain would run-to-complete, only bounded by the machine's execution capability. However, the busy-wait loop is not calibrated to machine performance, so the perceived output behavior changes depending on the hardware platform.

Domains in the seL4 context are a way to group threads. Domains include temporal execution budgets, and a fixed, repeating schedule, which is defined independently of the thread components themselves. The domain scheduler in stock seL4 supports the

completely deterministic scheduling of domains specified during system design. By default, all threads are assigned to Domain 0. During design time, the system developer may explicitly assign threads to other domains. Threads can only execute during the scheduled times for the domains to which they are assigned. Within each domain slice (time that it is resident on the processor), the threads in that domain are released for execution according to the normal seL4 scheduler. For our temporal isolation approach, we assign each thread to a single domain.

This basic behavior is sufficient to schedule the CASE Phase 2 UAV example with a static schedule that is akin to an ARINC 653 partition schedule. One capability that is missing from the stock seL4 domain scheduler is a mechanism to synchronize the thread execution with the domain schedule. When seL4 threads execute under the (currently verified) stock seL4, they are not aware of when, within their domain slice, they are executed, i.e., there is no concept of “this period” or “wait until next period.” Therefore, it is difficult to force periodic behavior in the default case.

To provide that capability in the base seL4, we make use of a safety feature of the verified seL4 kernel: *domain-to-domain notifications (AADL events) are propagated outside of the domain slice*. We use this mechanism to provide a self-pacing signal that each thread uses to start itself at the beginning of its domain slice. This approach enforces the behavior of periodic thread launches at the start of each domain slice. To support this behavior, the HAMR infrastructure generator adds emitter “tick” and consumer “tock” event ports to each thread component. Also generated by the HAMR infrastructure is a CAmkES connection from “tick” to “tock.” When initialized, the thread emits a “tick” pacing signal to itself, that it will not consume until the next domain slice. Once completing its initialization, the thread waits on the “tock” signal. That “tock” request blocks until the next domain slice. During regular periodic execution, it follows then the same pattern: the thread blocks on the tock, does its work, emits a tick event, and blocks again. The sequence of events and the connections is shown for two major frames for a simple source-to-destination model in Figure 3.

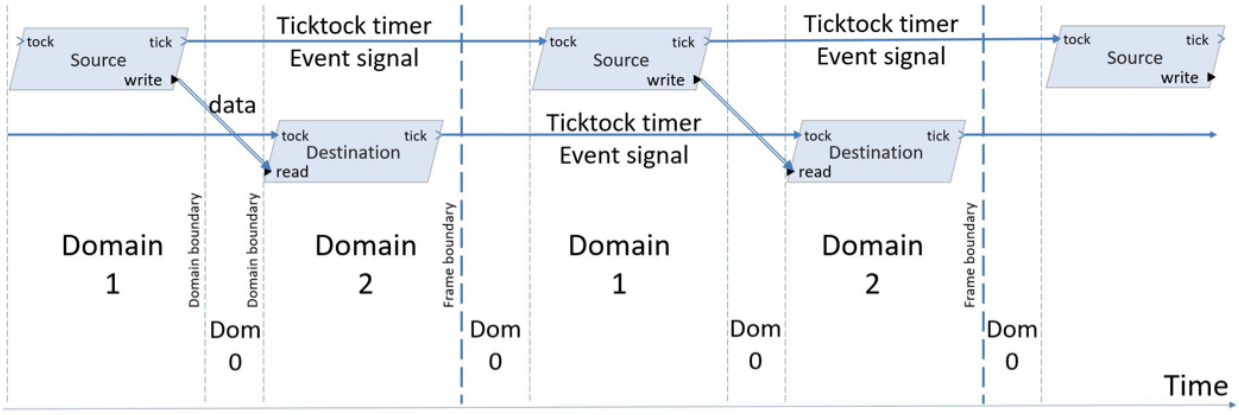

Figure 3: Tick-Tock timer approach for a simple source-destination model

This approach does not require modifications to application behavior code residing within the CAmkES component itself. Communications to and from the CAmkES component is the same as if the “tick-tock” mechanism is not part of the implementation.

The resulting system is resilient to component failures that could cause stalls leading to loss of processing availability. The transformed UAV example runs on QEMU, as well as and ODroid-XU4 and ODroid-C2 platforms. One particularly nice feature of this approach is that it is stable across architectures, and the timing is guaranteed by seL4. For example, when running directly on a hardware platform with hardware timers (e.g., the ODroids with their different processors and clock rates), the perceived output rate is the same, regardless of the processor clock rates. This benefit will aid testing, debugging, and certification efforts, since the behavior will be the same on bench tests as well as the embedded system.

Through a particular technique that utilizes the domain scheduler within seL4, HAMR is able to generate a system-build from AADL source that enforces time partitioning, as well as space partitioning. The following section describes how to construct the AADL source to use this time-scheduling technique. Other CASE resources describe the theory behind the seL4 domain scheduler and timing isolation.
Below is a list of constraints and properties required by HAMR that must be followed within the AADL system models in order to enact the described static cyclic scheduling paradigm on a seL4 platform.

- All threads are periodic (all threads should be declared with Periodic dispatch mode).
- There is one thread per process in the AADL model.
- Each AADL process is associated with a numeric domain identifier specified using the `CASE_Scheduling::Domain` property.

- The maximum value of the domain identifiers is specified in the AADL model (applies to system instance) using the `CASE_Scheduling::Max_Domain` property.
- `Timing_Properties::Compute_Execution_Time` (applies to each AADL thread type) specifies the duration for which the thread is scheduled. This property is used by HAMR to generate a hint in the auto-generated schedule skeleton. The value should match the slot duration value length in the seL4 domain schedule (this needs to be checked manually).
- `Timing_Properties::Period` (applies to each AADL thread type) specifies the period for each thread. Comments containing this information are included along with the HAMR generated skeleton for the schedule, helping the developer to verify that the periodicity of the thread implied by the written schedule matches the specified period in the AADL model (this needs to be checked manually).
- The schedule for the system is written as a domain schedule for seL4 (which is specified in a C data structure, processed by CAmkES). One future effort might be to develop a platform independent representation of the same information that would enable us to generate scheduling for Linux and JVM platforms.
- The tick duration for the underlying platform is configured using the `TimingProperties::Clock_Period`, applied to system instances. The slot durations within the schedule are specified in terms of ticks.

The period of the major frame of the schedule is specified using the `Timing_Properties::Frame_Period` property in the AADL model. The sum of the domain lengths specified in the domain schedule should equal this value. The frame period value is included in hints generated by HAMR to accompany an auto-generated schedule skeleton. Correspondence to the actual duration of the schedule must be checked manually.

rule **valid_sched_domain**	The value of a seL4_Properties::Domain property set on processes must be greater than or equal to 1. Domain zero is reserved for seL4/CAmkES operations and domain one is reserved for the Pacer component.	⚠

rule **consistent_sched_domain**	For all processes bound to the same processor (via the Actual_Processor_Binding property), the seL4_Properties::Domain property value assignments must be sequential and non-repeating.	⚠

rule **valid_compute_execute_time**	The Period property value assigned to a thread must be greater than or equal to the minimum time in the Compute_Execution_Time range value. All values for	⚠

	Period and Compute_Execution_Time property values must be greater than or equal to 2 ms. All values for Period and Compute_Execution_Time property values must be divisible by 2 ms, the default tick value for the seL4 platform.	

Utilizing the seL4 Domain Scheduler to Support the CASE Scheduling Approach

The *static schedule* defined by the system designer must satisfy the timing constraints on the computation or environment interactions of the system. For example, a thread for mission planning must execute quickly enough for flight controls, or satisfying jitter, throughput, and latency requirements on sensing and actuating actions in control loops.

On the seL4 platform, the static scheduling is realized using the seL4 domain-scheduling framework. The schedule itself is expressed as a C data structure used to configure seL4. The scheduling data structure for a simple producer (source) / consumer (destination) system is shown below.

```
// Copyright 2020 Adventium Labs
// This is a kernel data structure. You must compile this into your kernel.
// For example, you might modify the top level settings.cmake file to include
// it.
// camkes-project/projects/camkes/settings.cmake

#include <config.h>
#include <object/structures.h>
#include <model/statedata.h>

// An arbitrary hand generated schedule. The length is in seL4 ticks
// (2 ms default). This schedule should be generated from the AADL model
// using execution time and data flow latency specifications.
//
// This schedule is single-rate, 1Hz, run each thread within 200ms windows
// This will provide room to slot in other test examples without perturbing
//   this particular example.
// Fill space in with domain 0.
//           +
// 2 dest  |     -               -            -            -
// 1 src   |  -            -            -            -
// 0 dom0  |-- -- ----------- -- ----------- -- ----------- -- ---------
//         |_____________|_____________________________________________\time
//           seconds      1            2            3            4 /
//
// Major frame is 1 seconds, since destination has 1 second period
//
const dschedule_t ksDomSchedule[] = { // (1 tick == 2ms)
```

```
        { .domain = 0, .length = 100 }, //   all other seL4 threads, init, 200ms
        { .domain = 1, .length =   1 }, //   source           2ms
        { .domain = 0, .length =  99 }, //   domain0   198ms
        { .domain = 2, .length =   1 }, //   destination   2ms
        { .domain = 0, .length = 349 }, //   domain0     198ms + 500ms
                                    // + ____________________________
                                        ///  frame      1000ms
};
const word_t ksDomScheduleLength = sizeof(ksDomSchedule) / sizeof(dschedule_t);
```

Domains

Because we are leveraging the seL4 domain scheduler as the foundation of the CASE scheduling approach, the unit of schedulability is the *domain*. The CASE strategy uses domain 0 for infrastructure, and the rest for application threads (called *application* domains). At present, there is a one-to-one correspondence between AADL threads and application domains. Later we may be able to support multiple AADL threads in the same scheduling domain, but for CASE Phase 2 this is not supported.

We use the following convention for organizing the infrastructure domains:

- Domain 0 includes seL4 infrastructure threading for initialization, servicing interrupts, etc.
- Given a feed-forward design represented in a left-to-right layout in AADL, it is practical for the developer to visually inspect the schedule, if the domains are assigned in increasing order left-to-right. This is not required, but it has been handy to aid debugging and schedule visualization.

System developers do not need to configure the internal logic of the infrastructure domain. The CAmkES compilation process determines the logic of domain 0. However, the system designer will need to configure manually the time slots, following some straightforward guidelines. For application domains, the current convention is to simply consider the order in which AADL threads are intended to execute and select domain identifiers, starting at 1, according to that order. Note that the domain numbers are only identifiers – they do not influence the order of execution. The order of execution is completely determined by the order of entries in the domain schedule data structure. An AADL process is assigned to a domain using AADL model properties. This illustrated further in sections below.

Configuring the Number of Domains

The first step to develop the schedule is to establish the total number of domains. Currently this is specified manually in the CAmkES *cmake* file (not auto-generated by HAMR). Here is an example for the producer/consumer system: 2 application domains, 1 each for producer and consumer, and 1 infrastructure domain.

```
// set(KernelNumDomains 3 CACHE STRING "" FORCE)
```

Developing the Schedule Concept

Next, one would typically develop a “schedule concept,” an informal description of the schedule that can be shared across the development team. The inputs to developing the schedule concept include the end-to-end latency needs of the application and the information flow (dependences) reflected in the AADL model. The scheduling concept is developed and refined simultaneously with the specification of periods for each AADL thread, captured using an AADL thread property (see subsequent sections). Periods are also potentially determined by the end-to-end latency needs. See real-time scheduling textbooks for these concepts and process.[1]

Prior Figure 3 showed a visualization of a schedule concept for the procedure/consumer system. First, one aims to determine the relative ordering of the domains within the schedule. Following that, the specific time values for the slots in the schedule are determined.

The schedule for the system is cyclic, where each complete cycle is referred to as a *major frame*. The system designer specifies the schedule by configuring the ordering and timing properties of domains within the major frame. Within each major frame, a domain that has a thread rate that is the same as the major frame rate will typically be scheduled once. The exception to this convention is that domain 0 needs to be interleaved with application threads to handle interrupts and other kernel level services. Another exception is if a thread has a long duration and must be preempted to allow a higher rate thread to execute. For the remainder of the discussion we will focus on single-rate schedules for simplicity.

Next the developer will consider ordering implied by producer/consumer information and control flows. Analysis tools such as FASTAR automatically take information flow into account, as long as the AADL model follows the necessary modeling conventions (e.g., ARINC 653). One may manually accomplish the same for simple systems. For example, one may interleave application domains such as monitors to satisfy ordinal requirements (i.e., “monitor evaluates producer outputs before consumer reads inputs”).

The example producer/consumer schedule visualization above illustrates two major frames. In each frame, the application domains (domains 1 and 2) are each scheduled once, whereas domain 0 is scheduled multiple times, interleaved between the other domains.

When defining the schedule, domain 0 should always execute first. In the simple CASE scheduling methodology, the domain 0 time slot durations for will be determined by starting with a value that typically works, and then tweaking based on experience running the system (see additional guidelines at the end of this chapter).

For application domains, the ordering is typically determined by looking at the application data flow reflected in the AADL model. In our example, the producer component produces information that flows to the consumer. Therefore, we schedule the source thread before the destination thread, with domain 0 executions interleaved.

Defining the Schedule

After the schedule concept is developed, the system designer encodes the schedule in a data structure that is compiled into the seL4 kernel. An example schedule data structure for the producer/consumer system was shown above in Figure 3 (and repeated here).

```
const dschedule_t ksDomSchedule[] = { // (1 tick == 2ms)
       { .domain = 0, .length = 100 }, //   all other seL4 threads, init, 200ms
       { .domain = 1, .length =   1 }, //   source        2ms
       { .domain = 0, .length =  99 }, //   domain0       198ms
       { .domain = 2, .length =   1 }, //   destination   2ms
       { .domain = 0, .length = 349 }, //   domain0     198ms + 500ms
                                       // + ____________________________
                                      //   frame      1000ms
};
const word_t ksDomScheduleLength = sizeof(ksDomSchedule) / sizeof(dschedule_t);
```

The schedule is an array of domain and execution durations in the order of desired execution within the major frame. The domain fields are either 0 or refer to the CASE_Scheduling::Domain property associated with a process in the AADL model. The units on duration fields (.length) are seL4 *ticks*. The actual clock time for a tick is configured in TIMER_TICK_MS[2]. The default tick duration for verified seL4 kernels is 2 milliseconds.

It is good style to indicate the duration of each domain activation time in milliseconds (.length * tick duration) via comments in the code. It is also useful to indicate in comments the total duration (in ticks and seconds/milliseconds) of the major frame.

The following constraints[3] should hold for the completed data structure and AADL model:

- Each domain field should lie within the interval {0..CASE_Scheduling::Max_Domain}.
- Each domain from {0..CASE_Scheduling::Max_Domain} should be included at least once in the schedule.
- The slot duration (in milliseconds) written in comments should correspond to the Timing_Properties::Compute_Execution_Time property for the thread captured in the AADL model.
- The slot duration field value (in ticks) should be correctly computed from the millisecond duration in comments and the tick duration.
- The major frame duration written in comments (in milliseconds) should correspond to the Timing_Properties::Frame_Period property value written in the AADL model.
- The sum of the length fields (and associated milliseconds in comments) in the schedule should be equal to the major frame duration (in ticks, respectively milliseconds) written in comments.
- The time between the activation of an application domain in one cycle to the next activation of the domain (which may be in the next major frame) should be equal to the Period property of thread in the AADL model.

AADL Modeling of Timing and Domain Scheduling Information

The CASE scheduling approach provides strong temporal partitioning and predictability for user applications running on seL4, the primary execution context technology for the Collins CASE teams. To achieve alignment with the seL4 context, AADL modeling related to process and threads as well as AADL properties are restricted compared to the full expressive power available in AADL. The primary restrictions are:

- There is a single thread in each process (AADL normally allows multiple threads and thread groups per process).
- A CASE-specific property annotation explicitly associates a process to a scheduling domain (this concept is expressed using Virtual Machines in the ARINC 653 annex).

Below we provide an example-driven explanation of how models are structured and annotations are added to support the CASE scheduling approach.

Following the standard semantics of AADL, an AADL process represents a separate address space, a space partition unit whose boundaries are enforced at runtime (see Section 5.8 of the AADL standard). When using the seL4 CAmkES framework, a space

partition unit is represented as a CAmkES component. While AADL allows multiple threads per process, to simplify the verification argument, we currently restrict there to be exactly one AADL thread per process. Thus, an AADL process/thread pair will become both the unit of space partitioning and scheduling.

The AADL code below illustrates how processes and threads are declared in concert, with accompanying annotations, to support the CASE scheduling approach.

```
-- thread specifies unit of temporal execution; depending on scheduling
-- model this can provide temporal isolation.
thread source_thread
	features
		write_port: out data port Base_Types::Integer_8;
	properties
		Dispatch_Protocol => Periodic;
		Period => 1000ms;
			Compute_Execution_Time => 2ms .. 2ms;
			Source_Text => ("behavior_code/components/source/src/source.c");
			Initialize_Entrypoint_Source_Text =>
				"test_data_port_periodic_domains_source_component_init";
			Compute_Entrypoint_Source_Text =>

"test_data_port_periodic_domains_source_component_time_triggered";
end source_thread;

thread implementation source_thread.impl
end source_thread.impl;

-- process specifies boundary of spatial isolation
process source_process
	features
		write_port: out data port Base_Types::Integer_8;
	properties
		CASE_Scheduling::Domain => 1; -- source 1, destination 2
end source_process;

process implementation source_process.impl
	subcomponents
			source_thread_component: thread source_thread.impl;
	connections
			write_connection: port source_thread_component.write_port ->
write_port;
end source_process.impl;
```

Startup initialization often occurs in a different mode, then the schedule switches to operational mode. If partitions (corresponding to domains in this context) are restarted (e.g., suffered a failure and an external signal or watchdog causes it to reset), initialization runs within its regular domain slice. Initialization may take multiple frames

to complete, often understood as a temporary loss of availability of that function until it is again running in operational mode. If the system needs to transition to operational mode in fewer frames, then the system designer should allocate more time to the domain. This is a typical latency and efficiency trade. Latency can be a feature of platform resiliency. Efficiency is often driven by cost, so the ultimate trade is resiliency versus cost.

---

[1] There are many scheduling textbook examples, such as https://www.wiley.com/en-us/Real+time+Systems+Scheduling+1%3A+Fundamentals-p-9781848216655

[2] https://github.com/seL4/seL4/blob/master/config.cmake

[3] At the time of this writing, these constraints must be checked manually, although they may be automated in the future. HAMR currently generates some helpful hints when values used in the schedule can be derived from AADL model properties,

Appendix

The following is a set of basic AADL model examples that exercise specific communication types or transformations when converted to CAmkES application source code using the HAMR auto-generation tool.

Event Data Port

The event data port communication pattern is illustrated in Figure 2.

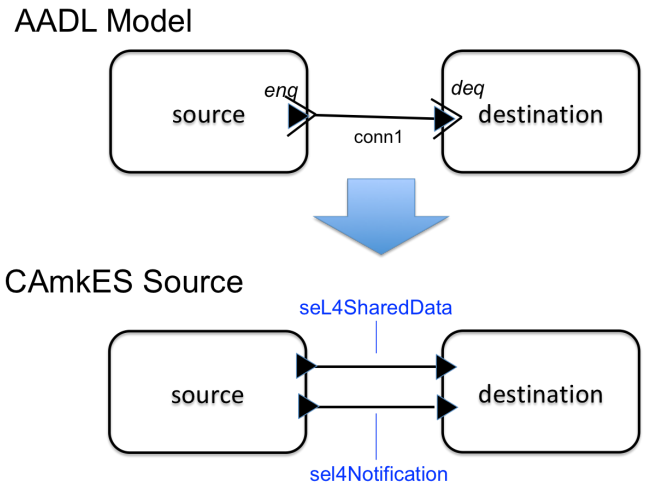

Figure 2: AADL event data ports are mapped to a notification/shared data connection pair when translated into CAmkES source. Following AADL run-time semantics, the data transferred on the shared data connection is queued at the destination end.

A simple in/out event data port connection in AADL is converted into a shared data connection and notification connection pair within CAmkES. The source component may only write to the shared data port, and the destination component may only read from the shared data port. The shared data is queued at the destination end of the shared data connection via glue-code that is auto-generated by HAMR. The AADL source, *test_event_data_port.aadl*, is shown here.

```
package test_event_data_port
public
    with HAMR;
    with Base_Types;

    thread emitter_t
       features
             enq: out event data port Base_Types::Integer_8;
       properties
             Dispatch_Protocol => Sporadic;
             Source_Text =>
                  ("behavior_code/components/emitter/src/emitter.c");
             Initialize_Entrypoint_Source_Text =>
                   "test_event_data_port_emitter_component_init";
             Compute_Entrypoint_Source_Text => "run_emitter";
    end emitter_t;

    thread implementation emitter_t.impl
    end emitter_t.impl;

    thread consumer_t
       features
             deq: in event data port Base_Types::Integer_8 {
                   Compute_Entrypoint_Source_Text =>
                        "test_event_data_port_consumer_s_event_handler";
             };
       properties
             Dispatch_Protocol => Sporadic;
             Source_Text =>
                  ("behavior_code/components/consumer/src/consumer.c");
             Initialize_Entrypoint_Source_Text =>
                  "test_event_data_port_consumer_component_init";
    end consumer_t;
```

```
    thread implementation consumer_t.impl
    end consumer_t.impl;

    processor proc
    end proc;

    processor implementation proc.impl
    end proc.impl;

    process top_process
    end top_process;

    process implementation top_process.impl
       subcomponents
             src: thread emitter_t.impl;
             dest: thread consumer_t.impl;
       connections
             conn1: port src.enq -> dest.deq;
    end top_process.impl;

    system top
    end top;

    system implementation top.impl
       subcomponents
             proc: processor proc.impl;
             test_event_data_port: process top_process.impl;
       properties
             Actual_Processor_Binding => (reference (proc))
                  applies to test_event_data_port;
             HAMR::Platform => (seL4_TB, seL4_Only);
    end top.impl;
end test_event_data_port;
```

The resulting CAmkES top-level assembly is shown here.

```
import <std_connector.camkes>;

import "components/emitter_t_impl/emitter_t_impl.camkes";
import "components/consumer_t_impl/consumer_t_impl.camkes";

assembly {
  composition {
    component emitter_t_impl src;
    component consumer_t_impl dest;
```

```
    connection seL4Notification
            conn1(from src.sb_enq_1_notification,
                  to dest.sb_deq_notification);
    connection seL4SharedData
            conn2(from src.sb_enq_queue_1, to dest.sb_deq_queue);
  }

  configuration {
      src.sb_enq_queue_1_access = "W";
      dest.sb_deq_queue_access = "R";
  }
}
```

Data Port

The data port communication pattern is similar to the event data port, except that the data is not queued by the monitor (or alternatively, the shared data represents a queue of size one). If the sending thread sends subsequent data before the receiving component has read the data from the shared memory, then the prior data is overwritten. For completeness, the AADL *test_data_port.aadl* is shown here.

```
package test_data_port
public
    with Base_Types;
    with HAMR;

    thread source_t
       features
             write_port: out data port Base_Types::Integer_8;
       properties
             Dispatch_Protocol => Sporadic;
             Source_Text =>
                  ("behavior_code/components/source/src/source.c");
             Initialize_Entrypoint_Source_Text =>
                  "test_data_port_source_component_init";
             Compute_Entrypoint_Source_Text => "run_sender";
    end source_t;

    thread implementation source_t.impl
    end source_t.impl;

    thread destination_t
       features
             read_port: in data port Base_Types::Integer_8;
       properties
```

```
            Dispatch_Protocol => Sporadic;
            Source_Text =>
                 ("behavior_code/components/destination/src/destination.c");
            Initialize_Entrypoint_Source_Text =>
                 "test_data_port_destination_component_init";
            Compute_Entrypoint_Source_Text => "run_receiver";
      end destination_t;

      thread implementation destination_t.impl
      end destination_t.impl;

      processor proc
      end proc;

      processor implementation proc.impl
      end proc.impl;

      process top_process
      end top_process;

      process implementation top_process.impl
         subcomponents
            src: thread source_t.impl;
            dest: thread destination_t.impl;
         connections
            conn1: port src.write_port -> dest.read_port;
      end top_process.impl;

      system top
      end top;

      system implementation top.impl
         subcomponents
            proc: processor proc.impl;
            test_data_port: process top_process.impl;
         properties
            Actual_Processor_Binding =>
                 (reference (proc)) applies to test_data_port;
            HAMR::PLatform => (seL4_TB, seL4_Only);
      end top.impl;
end test_data_port;
```

The resulting CAmkES top-level assembly generated by HAMR is shown here.

```
import <std_connector.camkes>;
```

```
import "components/source_t_impl/source_t_impl.camkes";
import "components/destination_t_impl/destination_t_impl.camkes";

assembly {
  composition {
    component source_t_impl src;
    component destination_t_impl dest;

    connection seL4SharedData
            conn1(from src.sb_write_port, to dest.sb_read_port);
  }

  configuration {
    src.write_port_access = "W";
    dest.read_port_access = "R";
  }
}
```

Event Port

Event port communications represent a simple one-way notification message between a sender and a receiver thread without associated data. Event port communication and their translation to CAmkES are illustrated in Figure 3.

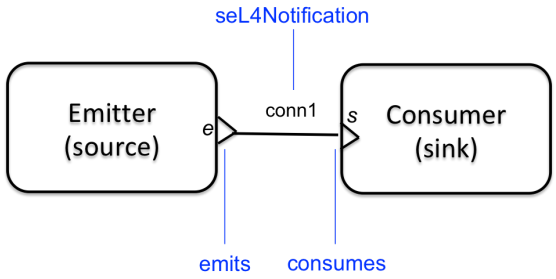

Figure 3: AADL event ports are mapped to the emits/consumes pairs when translated into CAmkES source.

A sample AADL model *test_event_port.aadl* that includes an event port is shown here.

```
package test_event_port
public
    with HAMR;

    thread emitter
       features
             e: out event port;
       properties
             Dispatch_Protocol => Sporadic;
```

```
          Initialize_Entrypoint_Source_Text =>
               "test_event_port_emitter_component_init";
          Source_Text =>
               ("behavior_code/components/Emitter/src/emitter.c");
          Compute_Entrypoint_Source_Text => "run_emitter";
end emitter;

thread implementation emitter.impl
end emitter.impl;

thread consumer
   features
          s: in event port {
                Compute_Entrypoint_Source_Text =>
                     "test_event_port_consumer_s_event_handler";
          };
   properties
          Dispatch_Protocol => Sporadic;
          Initialize_Entrypoint_Source_Text =>
               "test_event_port_consumer_component_init";
          Source_Text =>
               ("behavior_code/components/Consumer/src/consumer.c");
end consumer;

thread implementation consumer.impl
end consumer.impl;

processor proc
end proc;

processor implementation proc.impl
end proc.impl;

process top_process
end top_process;

process implementation top_process.impl
   subcomponents
          src: thread emitter.impl;
          snk: thread consumer.impl;
   connections
          conn1: port src.e -> snk.s;
end top_process.impl;

system top
end top;

system implementation top.impl
```

```
        subcomponents
              proc: processor proc.impl;
              test_event_port: process top_process.impl;
        properties
              Actual_Processor_Binding =>
                    (reference (proc)) applies to test_event_port;
              HAMR::Platform => (seL4_TB, seL4_Only);
    end top.impl;
end test_event_port;
```

The resulting CAmkES top-level assembly generated by HAMR is shown here.

```
import <std_connector.camkes>;

import "components/emitter_impl/emitter_impl.camkes";
import "components/consumer_impl/consumer_impl.camkes";

assembly {
  composition {
    component emitter_impl src;
    component consumer_impl snk;

    connection seL4Notification conn1(from src.e, to snk.s);
  }

  configuration {
  }
}
```

Virtual Machine Communications

This section describes the communication infrastructure utilized by the CASE system build environment specifically for communications to and from a (Linux) virtual machine hosted within a CAmkES component.

The diagram below shows a simple example of a Linux-based virtual machine in a CAmkES component interacting with a “ping client” hosted on a regular, non-virtual CAmkES component. The Linux virtual machine (name `vm`) sends ping messages and the receiving ping client (named `ping`) responds accordingly following the standard ping protocol. The process `vm` is bound to the virtual processor `vproc`, while `vproc` in turn is bound to the physical processor `proc`. The process `ping` is bound to `proc`.

Currently the CASE system build only supports event data port communication for virtual machines.

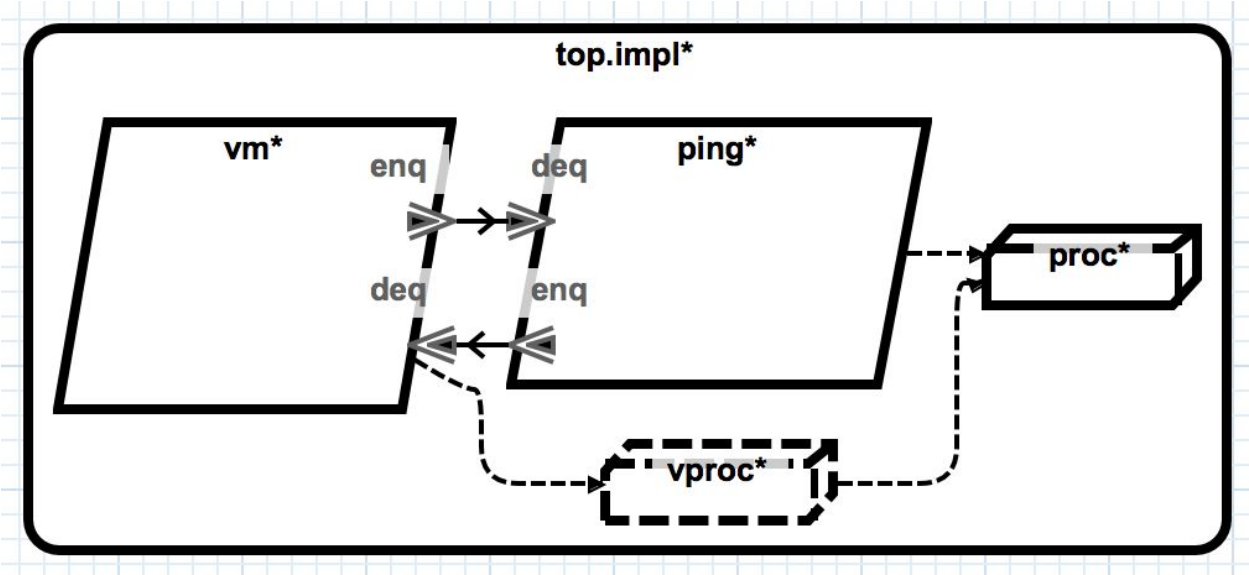

Figure 4: AADL representation of a Linux-based virtual machine within a CAmkES component interacting with a separate non-virtual CAmkES component.

Here is example AADL source code representing the model.

```
package test_event_data_port_periodic_domains
public
    with HAMR;
    with Base_Types;
    with CASE_Scheduling;

    thread emitter_t
       features
             write_port: out event data port Base_Types::Integer_8;
       properties
             Dispatch_Protocol => Periodic;
             Period => 1000ms;
             Compute_Execution_Time => 10ms .. 10ms;
             Source_Text =>
                  ("behavior_code/components/emitter/src/emitter.c");
             Initialize_Entrypoint_Source_Text =>
                  "test_event_data_port_emitter_component_init";
             Compute_Entrypoint_Source_Text =>
                  "test_event_data_port_emitter_time_triggered_handler";
    end emitter_t;
```

```
thread implementation emitter_t.impl
end emitter_t.impl;

process emitter_p
   features
         write_port: out event data port Base_Types::Integer_8;
   properties
         CASE_Scheduling::Domain => 2; -- pacer 1, source 2, destination 3
         HAMR::Component_Type => VIRTUAL_MACHINE;
end emitter_p;

-- process specifies boundary of spatial isolation
process implementation emitter_p.impl
   subcomponents
         src_thread: thread emitter_t.impl;
   connections
         write_connection: port src_thread.write_port -> write_port;
end emitter_p.impl;

thread consumer_t
   features
         read_port: in event data port Base_Types::Integer_8 {
               Compute_Entrypoint_Source_Text =>
                     "Periodic_thread_so_should_be_ignored";
         };
   properties
         Dispatch_Protocol => Periodic;
         Period => 1000ms;
         Compute_Execution_Time => 10ms .. 10ms;
         Source_Text =>
               ("behavior_code/components/consumer/src/consumer.c");
         Initialize_Entrypoint_Source_Text =>
               "test_event_data_port_consumer_component_init";
         Compute_Entrypoint_Source_Text =>
               "test_event_data_port_consumer_time_triggered_handler";
end consumer_t;

thread implementation consumer_t.impl
end consumer_t.impl;

process consumer_p
   features
         read_port: in event data port Base_Types::Integer_8;
   properties
         CASE_Scheduling::Domain => 3; -- pacer 1, source 2, destination 3
end consumer_p;

-- process specifies boundary of spatial isolation
```

```
    process implementation consumer_p.impl
       subcomponents
             dst_thread: thread consumer_t.impl;
       connections
             read_connection: port read_port -> dst_thread.read_port;
    end consumer_p.impl;

    processor proc
    end proc;

    processor implementation proc.impl
       properties
             Frame_Period => 1000ms;
             Clock_Period => 2ms;
             CASE_Scheduling::Max_Domain => 3;
             CASE_Scheduling::Schedule_Source_Text =>
                  "behavior_code/kernel/domain_schedule.c";
    end proc.impl;

    system top
    end top;

    system implementation top.impl
       subcomponents
             proc: processor proc.impl;
             src_process: process emitter_p.impl;
             dst_process: process consumer_p.impl;
       connections
             data_interconnect:
                  port src_process.write_port -> dst_process.read_port;
       properties
             Actual_Processor_Binding =>
                  (reference (proc)) applies to src_process;
             Actual_Processor_Binding =>
                  (reference (proc)) applies to dst_process;
             HAMR::Platform => (seL4_TB, seL4_Only);
    end top.impl;
end test_event_data_port_periodic_domains;
```