# AADL Modeling Guidelines for CASE

December 5, 2019

## 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)
- **Legacy Component Verification (TA 3)**
  - Ivaldi (BBN)
- **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*<br>**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.

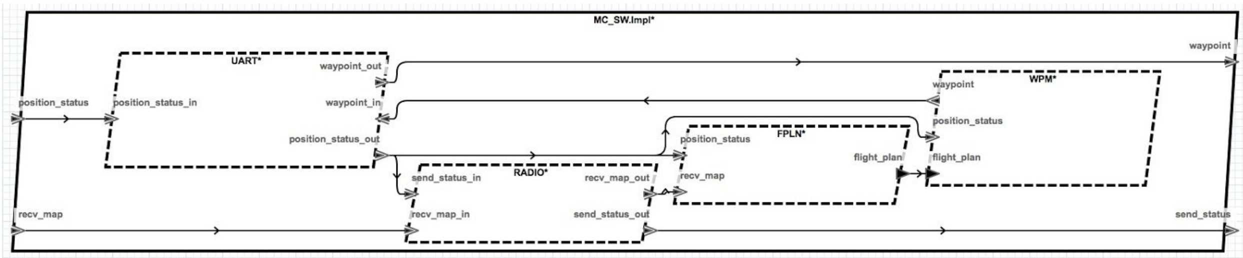


**Figure 1: The primary software architecture model of the mission computer application in the "Simple UAV" example developed under the CASE program.**

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* **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 collection 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). While both AADL and CAmkES support nested thread subcomponents, the system build toolchain currently does not support this feature. 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*<br>**dispatch_protocol_specified** | Threads should have the dispatch_protocol property specified | ⚠ |

| | | |
|---|---|---|
| *rule*<br>**valid_dispatch_protocol** | Threads can only specify a dispatch_protocol property of *periodic* or *sporadic* | ⊗ |

## 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*<br>**bounded_integers** | Integer types must be bounded (cannot use Base_Types::Integer) | ⊗ |

| | | |
|---|---|---|
| *rule*<br>**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

[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.

*MISSING_TYPE* and issue a warning. For example, HAMR will attach the *MISSING_TYPE* classifier to [SW::WifiDriver.gimbal_command]

| *rule*<br>**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 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*<br>**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*<br>**array_dimension** | Array dimensions must be specified | ⊗ |

| | | |
|---|---|---|
| *rule*<br>**one_dimensional_arrays** | Arrays can only have one dimension | ⊗ |

| | | |
|---|---|---|
| *rule*<br>**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, AADL subprogram group access into sel4 RPC, and AADL data access ports translate into seL4 shared data channels. More complicated communication mechanisms use the monitor construct, developed under prior DARPA funding.

Refer to the following table:

| AADL Port Type | seL4 Communications | CAmkES Description |
|---|---|---|
| Event Data Port | RPC + Notification | For communications between (non-virtualized) CAmkES components, implemented as a custom monitor filter. For communications between CAmkES components that host a virtual machine (i.e., a Linux instance in a virtual machine), the connection is implemented as a custom *virtqueue* filter. See details in the appendix section of this document entitled “Virtual Machine Communications”. |
| Data Port | RPC + Notification | Implemented as a custom monitor filter. |
| Event Port | Notification | Implemented as an emit/consumes pair. |
| Data Access | Shared Data | Implemented as a provides/requires pair. |
| Subprogram Group Access | RPC | Implemented as a provides/requires pair. |

**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. Note that HAMR currently only supports event data ports to and from CAmkES components hosting virtual machines.

| | | |
|---|---|---|
| *rule*<br>**event_data_ports_on_vm** | Components bound to a virtual processor may only communicate with components bound to other processors via event data ports | |

## 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).

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

```
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 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 a 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*<br>**no_processor_subcomponents** | Processor subcomponents may be ignored | ⚠ |

# Appendix

The following is a set of simple AADL model examples that exercise specific communication types when converted to CAmkES application source code using the HAMR model translator.

## Event Data Port Monitor

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

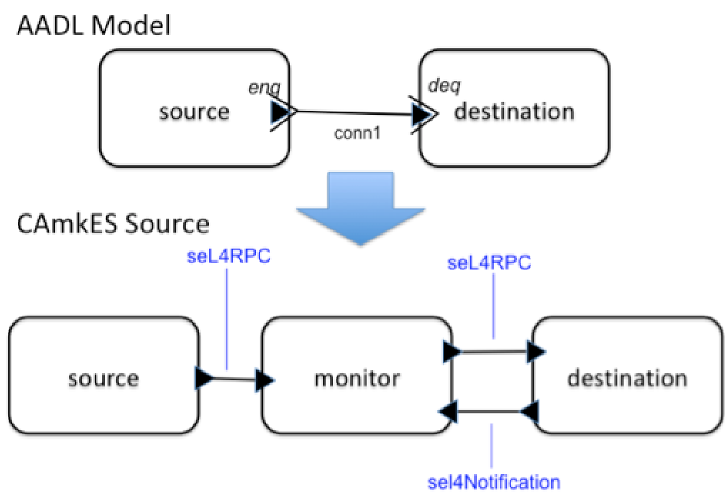


**Figure 2: AADL event data ports are mapped to the monitor filter when translated into CAmkES source.** The monitor filter is implemented as its own CAmkES partitioned component, and queues the data for the destination component. AADL data ports also use the monitor filter.

A simple in/out event data port pair in AADL is converted into a monitor components that manages communications between the sending and receiving threads. The monitor enqueues data incoming from the sending thread, and dequeues data once it receives a notification from the receiving thread. The AADL source, *testepmon.aadl*, is shown here.

---

```
package testepmon
public
        with SB_SYS;
        with Base_Types;

        thread emitter_t
                features
                        enq: out event data port Base_Types::Integer_8;
        properties
                Source_Text =>
                        ("behavior_code/components/emitter/src/run_emitter.c");
                Initialize_Entrypoint_Source_Text =>
                        "testepmon_emitter_component_init";
                SB_SYS::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 {
```

```
                        SB_SYS::Compute_Entrypoint_Source_Text =>
                              ("testepmon_consumer_s_event_handler");
                  };
            properties
            Source_Text =>
                        ("behavior_code/components/consumer/src/run_consumer.c");
            Initialize_Entrypoint_Source_Text =>
                        "testepmon_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;
                  testepmon: process top_process.impl;
            properties
                  Actual_Processor_Binding =>
                        (reference (proc)) applies to testepmon;
      end top.impl;
end testepmon;
```

---

The resulting CAmkES top-level assembly, *testepmon.camkes*, 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";
import "components/tb_Monitors/tb_dest_deq_Monitor/tb_dest_deq_Monitor.camkes";

assembly {
  composition {
    component emitter_t_impl src;
    component consumer_t_impl dest;
    component tb_dest_deq_Monitor tb_dest_deq_monitor;

    connection seL4RPCCall
            conn1(from src.tb_enq0, to tb_dest_deq_monitor.mon);
    connection seL4RPCCall
            conn2(from dest.tb_deq, to tb_dest_deq_monitor.mon);
    connection seL4Notification
            conn3(from tb_dest_deq_monitor.monsig,
                  to dest.tb_deq_notification);
  }

  configuration {
  }
}
```

## Data Port Monitor

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

---

```
package testdpmon
public
      with SB_SYS;
      with Base_Types;

      thread source_t
            features
                  enq: out data port Base_Types::Integer_8;
            properties
            Source_Text =>
                        ("behavior_code/components/source/src/source.c");
            Initialize_Entrypoint_Source_Text =>
                        "testdpmon_source_component_init";
```

```
            SB_SYS::Compute_Entrypoint_Source_Text =>
                  ("run_sender");
end source_t;

thread implementation source_t.impl
end source_t.impl;

thread destination_t
      features
            deq: in data port Base_Types::Integer_8;
      properties
            Initialize_Entrypoint_Source_Text =>
                  "testdpmon_destination_component_init";
      Source_Text =>
                  ("behavior_code/components/destination/src/destination.c"
            );
      SB_SYS::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.enq -> dest.deq;
end top_process.impl;

system top
end top;

system implementation top.impl
      subcomponents
            proc: processor proc.impl;
            testdpmon: process top_process.impl;
      properties
```

```
                        Actual_Processor_Binding =>
                              (reference (proc)) applies to testdpmon;
        end top.impl;
  end testdpmon;
```

The resulting CAmkES top-level assembly, *testdpmon.camkes*, 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";
import "components/tb_Monitors/tb_dest_deq_Monitor/tb_dest_deq_Monitor.camkes";

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

    connection seL4RPCCall
            conn1(from src.tb_enq0, to tb_dest_deq_monitor.mon);
    connection seL4RPCCall
            conn2(from dest.tb_deq, to tb_dest_deq_monitor.mon);
    connection seL4Notification
            conn3(from tb_dest_deq_monitor.monsig,
                  to dest.tb_deq_notification);
  }

  configuration {
  }
}
```

## 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.

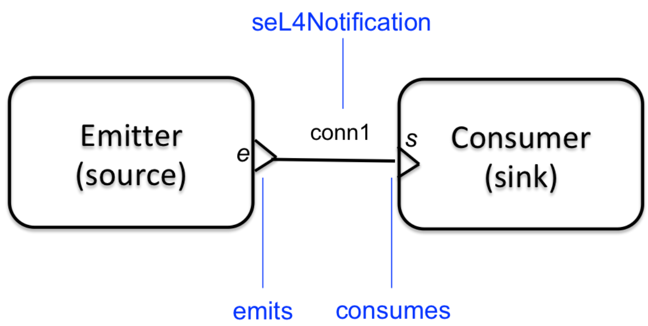


**Figure 3: AADL event ports are mapped to the emits/consumes pairs when translated into CAmkES source.**

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

---

```
package testevent
public
      with SB_SYS;

      thread emitter
            features
                  e: out event port;
            properties
            Initialize_Entrypoint_Source_Text =>
                        "testevent_emitter_component_init";
            Source_Text =>
                        ("behavior_code/components/Emitter/src/emitter.c");
            SB_SYS::Compute_Entrypoint_Source_Text =>
                        ("run_emitter");
      end emitter;

      thread implementation emitter.impl
      end emitter.impl;

      thread consumer
            features
                  s: in event port {
                        SB_SYS::Compute_Entrypoint_Source_Text =>
                              ("testevent_consumer_s_event_handler");
                  };
            properties
            Initialize_Entrypoint_Source_Text =>
                        "testevent_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;
			testevent: process top_process.impl;
		properties
			Actual_Processor_Binding => (reference (proc))
				applies to testevent;
	end top.impl;
end testevent;
```

The resulting CAmkES top-level assembly, *testevent.camkes*, 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 {
    }
}
```

## Data Access

Data access communications in AADL are translated by the HAMR tool into shared data communications, in which both threads have read-write access to a block of memory that is protected by a semaphore. See Figure 4 below.

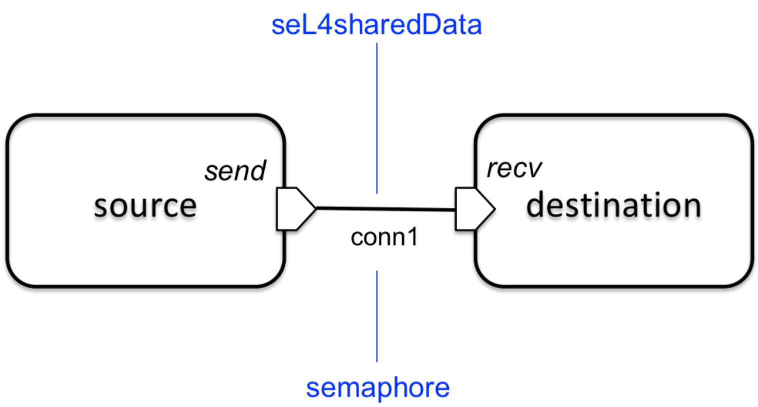


**Figure 4: AADL data access connections are mapped to the seL4sharedData communications when translated into CAmkES source.** The generated CAmkES also includes semaphore support that protects the shared data, inserted into the behavior code of the source and destination thread components.

A sample AADL model *testshare.aadl* that includes a data access communication is shown here. In the example, the data representing the block of shared memory is a custom data type that is a record with four short integers.

---

```
package testshare
public
    with SB_SYS;
    with Base_Types;

    data Thing_t
    end Thing_t;

    data implementation Thing_t.impl
        subcomponents
            lepht: data Base_Types::Integer_8;
            right: data Base_Types::Integer_8;
            top: data Base_Types::Integer_8;
```

```
            bottom: data Base_Types::Integer_8;
end Thing_t.impl;

thread publisher
        features
            b1: requires data access Thing_t.impl;
        properties
            Initialize_Entrypoint_Source_Text =>
                "testshare_publisher_component_init";
        Source_Text =>
                ("behavior_code/components/publisher/src/publisher.c");
        SB_SYS::Compute_Entrypoint_Source_Text =>
                ("run_publisher");
end publisher;

thread implementation publisher.impl
end publisher.impl;

thread subscriber
        features
            b2: requires data access Thing_t.impl;
        properties
        Initialize_Entrypoint_Source_Text =>
                "testshare_subscriber_component_init";
        Source_Text =>
                ("behavior_code/components/subscriber/src/subscriber.c");
        SB_SYS::Compute_Entrypoint_Source_Text =>
                ("run_subscriber");
end subscriber;

thread implementation subscriber.impl
end subscriber.impl;

processor proc
end proc;

processor implementation proc.impl
end proc.impl;

process top_process
end top_process;

process implementation top_process.impl
        subcomponents
            publisher_inst: thread publisher.impl;
            subscriber_inst: thread subscriber.impl;
            shared: data Thing_t.impl {
                SB_SYS::CAmkES_Owner_Thread =>
```

```
                                "testshare::publisher.impl";
                    };
            connections
                    conn2: data access shared -> subscriber_inst.b2;
                    conn1: data access shared -> publisher_inst.b1;

        end top_process.impl;

        system top
        end top;

        system implementation top.impl
                subcomponents
                        proc: processor proc.impl;
                        testshare: process top_process.impl;
                properties
                        Actual_Processor_Binding => (reference (proc))
                                applies to testshare;
        end top.impl;
end testshare;
```

---

The resulting CAmkES top-level assembly, *testshare.camkes*, generated by HAMR is shown here.

```
import <std_connector.camkes>;

import "interfaces/sb_testshare__Thing_t_impl_shared_var.idl4";
import "components/publisher_impl/publisher_impl.camkes";
import "components/subscriber_impl/subscriber_impl.camkes";

assembly {
  composition {
    component publisher_impl publisher_inst;
    component subscriber_impl subscriber_inst;
    connection seL4SharedData
      conn1(from subscriber_inst.b2, to publisher_inst.b1);
  }

  configuration {
  }
}
```

## Subprogram Access

Subprogram access communications between threads in AADL are translated by HAMR into remote procedure calls (RPC) communications. See Figure 5.

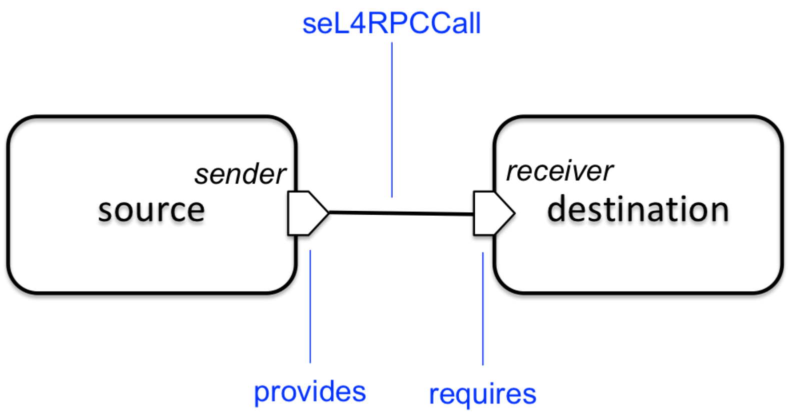


**Figure 5: AADL subprogram access connections are mapped to the seL4RPCCall communications when translated into CAmkES source.**

A sample AADL model *testsubprogram.aadl* that includes a data access communication is shown here.

---

```
package testsubprogram
public
      with Base_Types;
      with SB_SYS;

      subprogram add_uint32
            features
                  A: in parameter Base_Types::Unsigned_32;
                  B: in parameter Base_Types::Unsigned_32;
                  result: out parameter Base_Types::Unsigned_32;
      end add_uint32;

      subprogram subtract_uint32
            features
                  A: in parameter Base_Types::Unsigned_32;
                  B: in parameter Base_Types::Unsigned_32;
                  result: out parameter Base_Types::Unsigned_32;
      end subtract_uint32;

      subprogram group operations_interface
            features
                  add: provides subprogram access add_uint32;
                  subtract: provides subprogram access subtract_uint32;
      end operations_interface;

      thread sender
            features
```

```
        operations: requires
            subprogram group access operations_interface;
end sender;

thread implementation sender.impl
    properties
        Source_Text =>
            ("behavior_code/components/sender/src/sender.c");
        Initialize_Entrypoint_Source_Text =>
            "sender_init";
        SB_SYS::Compute_Entrypoint_Source_Text =>
            ("run_sender");
end sender.impl;

thread receiver
    features
        operations: provides
            subprogram group access operations_interface;
end receiver;

thread implementation receiver.impl
    properties
        Source_Text =>
            ("behavior_code/components/receiver/src/receiver.c");
end receiver.impl;

processor proc
end proc;

processor implementation proc.impl
end proc.impl;

process top_process
end top_process;

process implementation top_process.impl
    subcomponents
        source_inst: thread sender.impl;
        destination_inst: thread receiver.impl;
        subgroup: subprogram group operations_interface;
        sub1: subprogram add_uint32;
        sub2: subprogram subtract_uint32;
    connections
        source_to_destination :
            subprogram group access
                source_inst.operations ->
                    destination_inst.operations;
end top_process.impl;
```

```
    system top
    end top;

    system implementation top.impl
        subcomponents
            proc : processor proc.impl;
            testsubprogram: process top_process.impl;
        properties
            Actual_Processor_Binding =>
                (reference (proc)) applies to testsubprogram;
    end top.impl;

end testsubprogram;
```

The resulting CAmkES top-level assembly, *testsubprogram.camkes*, generated by HAMR is shown here.

```
import <std_connector.camkes>;
import "interfaces/add_uint32.idl4";
import "interfaces/subtract_uint32.idl4";
import "interfaces/operations_interface.idl4";
import "components/sender_impl/sender_impl.camkes";
import "components/receiver_impl/receiver_impl.camkes";

assembly {
  composition {
    component sender_impl source_inst;
    component receiver_impl destination_inst;

    connection seL4RPCCall
        conn1(from source_inst.operations,
              to destination_inst.operations);
  }

  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.

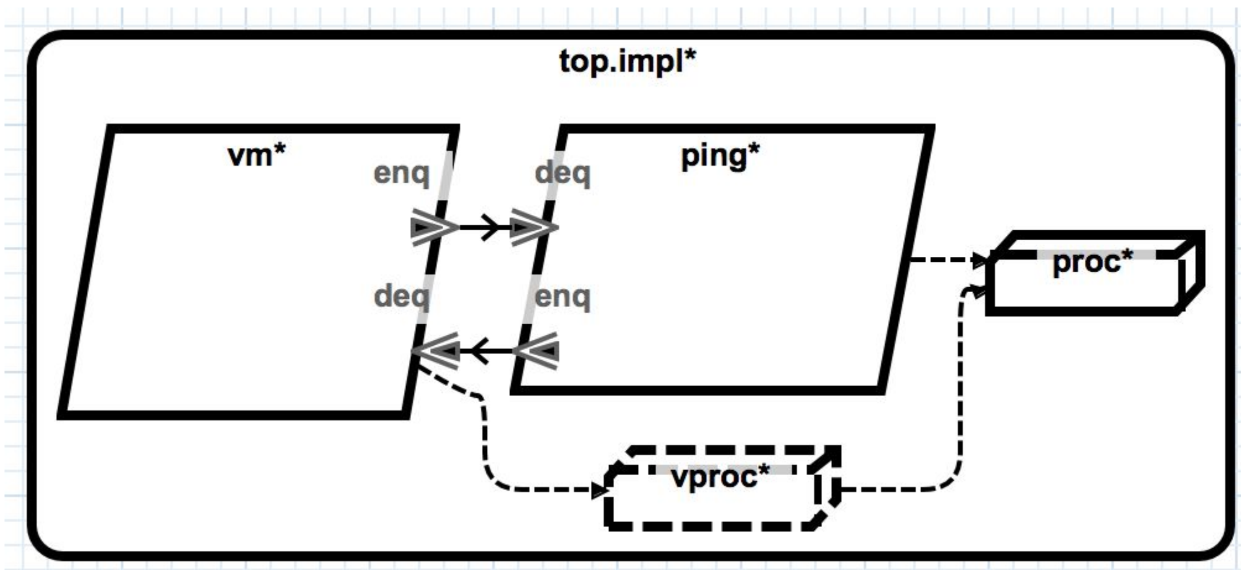


**Figure 6: AADL representation of a Linux-based virtual machine within a CAmkES component interacting with a separate non-virtual CAmkES component.**

Here is the full AADL source code representing the model.

---

```
package testvm
public
    data TIPC_Message
    end TIPC_Message;

    thread emitter_t
       features
             enq: out event data port TIPC_Message;
             deq: in event data port TIPC_Message;
```

```
end emitter_t;

thread implementation emitter_t.impl
end emitter_t.impl;

thread consumer_t
   features
         enq: out event data port TIPC_Message;
         deq: in event data port TIPC_Message;
   properties
         Source_Text =>
               ("behavior_code/components/consumer/src/run_ping.c");
         Initialize_Entrypoint_Source_Text => "testepvm_ping_init";
end consumer_t;

thread implementation consumer_t.impl
end consumer_t.impl;

processor proc
end proc;

processor implementation proc.impl
end proc.impl;

virtual processor vproc
end vproc;

virtual processor implementation vproc.impl
end vproc.impl;

process vm_p
   features
         enq: out event data port TIPC_Message;
         deq: in event data port TIPC_Message;
end vm_p;

process implementation vm_p.impl
   subcomponents
         vm : thread emitter_t.impl;
   connections
         outgoing : port vm.enq -> enq;
         incoming : port deq -> vm.deq;
end vm_p.impl;

process ping_p
   features
         enq: out event data port TIPC_Message;
```

```
            deq: in event data port TIPC_Message;
    end ping_p;

    process implementation ping_p.impl
       subcomponents
             vm : thread emitter_t.impl;
       connections
             outgoing : port vm.enq -> enq;
             incoming : port deq -> vm.deq;
    end ping_p.impl;

    system top
    end top;

    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;
end testvm;
```

The following figure shows the components that are generated by HAMR from the model.

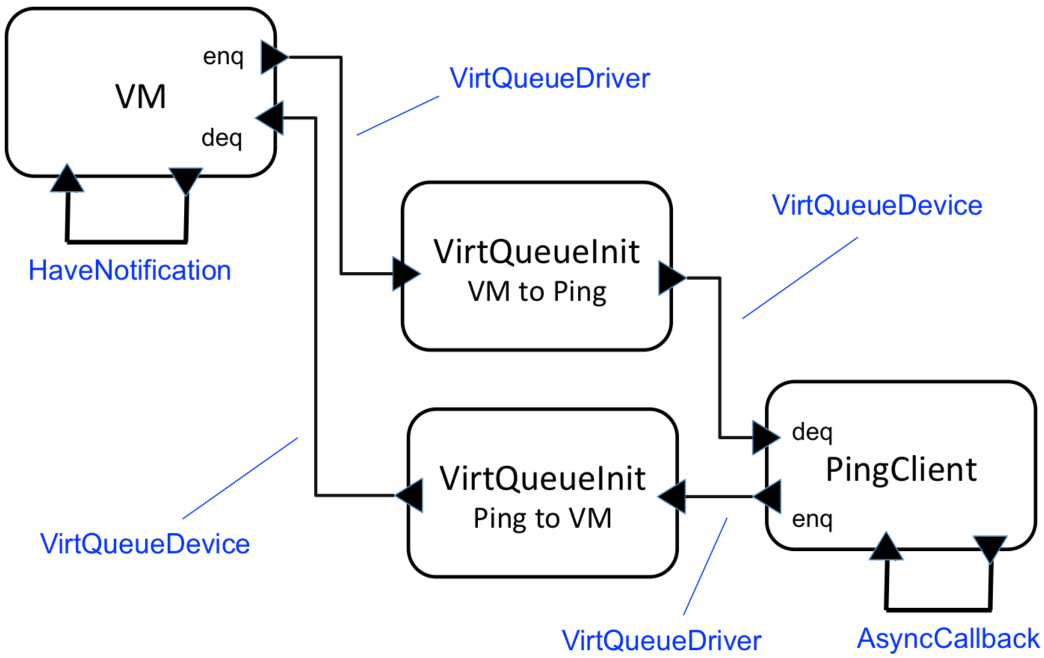


**Figure 7: The VirtQueue components generated by HAMR perform the mediation and queueing of messages to and from a virtual machine CAmkES component.**

The generated system build include the internal communication components *virtqueue*, CAmkES themselves, which mediate and queue the communications between the Linux virtual machine and the ping client. All of the internal communication paths are unidirectional, including the return acknowledgements. Thus, there are separate *virtqueue* component for each to and from the virtual machine.

Communications going from the sending CAmkES component and to the *VirtQueueInit* component utilize the connection type *VirtQueueDriver*, while communications going from the VirtQueueInit component to the receiving CAmkES component utilize the connection type *VirtQueueDevice*. The data queue maintained within each VirtQueueInit component is not typed, but instead a fixed block of memory, and the individual messages identify specify the size of each queue element.

Operations used within the behavior code of the CAmkES component to manage the virtqueue communications are provided in the table below. Note that all operations are non-blocking.

| Name | Description |
|---|---|
| `virtqueue_{driver|device}_t` | Handles to *VirtQueueDev* or *VirtQueueDrv* connections. |
| `camkes_virtqueue_{driver|device}_init` | Function that initializes virtqueue connections. |
| `camkes_virtqueue_{driver|device}_alloc` | Function to allocate memory for a new message. |
| `camkes_virtqueue_{driver|device}_free` | Function to free memory allocated to a message on the queue, typically made inside a send acknowledgement callback (see below). |
| `virtqueue_{driver|device}_poll` | Function to poll for an event on a virtqueue connection. Non-blocking. |
| `virtqueue_{driver|device}_enqueue` | Function that sends a message (or acknowledgement message) to a virtqueue. |
| `virtqueue_{driver|device}_signal` | Function that sends a signal event that indicates a new message (or acknowledgement message) has been sent to a virtqueue. |
| `virtqueue_{driver|device}dequeue` | Function that removes a message (or return acknowledgement message) from a virtqueue. |

**Table 4: CAmkES Virtqueue Operations.**

From the sending CAmkES component, the `virtqueue_*_alloc` operation reserves memory on the virtqueue for the outgoing message, the `virtqueue_*_enqueue` operation puts the message on the virtqueue queue, and the `virtqueue_*_signal` operation notifies the virtqueue and receiving component that a new message is available on the queue. The sending component is also responsible for calling the `virtqueue_*_free` operation, to deallocate the memory dedicated to a message, once the message has been safely acquired by the virtqueue component. This is implemented by using a callback, which is explained below.

From the receiving component, the `virtqueue_*_dequeue` operation removes a message from the virtqueue. In the “ping” example, the ping-client example responds to incoming messages by formatting and sending an appropriate ping response message, which is similarly sent via a separate set of virtqueue driver and device channels, which transmit the message in the opposite direction of the original ping message.

Both sending and receiving CAmkES components with virtqueue connections are responsible for initializing their incoming/outgoing connections using the `virtqueue_*_init` operation.

Callback functions are used to indicate that a new message is available on the virtqueue to receive via a `virtqueue_device_t` connection, and that a sent message has been received by the virtqueue component via a `virtqueue_driver_t` connection (and subsequently the memory allocated for the sent message may be safely freed). The handle name of the callback is inferred from the name of the process hosted in the CAmkES component.

Refer to the following behavior code designated for the "ping-client" component from the example above, which has been condensed for clarity.

---

```
virtqueue_device_t *recv_virtqueue;
virtqueue_driver_t *send_virtqueue;

void handle_recv_callback(virtqueue_device_t *vq);
void handle_send_callback(virtqueue_driver_t *vq);

...

int send_outgoing_packet(char *outgoing_data, size_t outgoing_data_size)
{
	volatile void *alloc_buffer = NULL;
	int err = camkes_virtqueue_buffer_alloc(
		send_virtqueue,
		&alloc_buffer,
		outgoing_data_size);
	if (err) {
		return -1;
	}

	char *buffer_data = (char *)alloc_buffer;
	memcpy(buffer_data, outgoing_data, outgoing_data_size);
	err = virtqueue_driver_enqueue(send_virtqueue,
				alloc_buffer, outgoing_data_size);
	if (err != 0) {
		ZF_LOGE("Client send enqueue failed");
		camkes_virtqueue_buffer_free(send_virtqueue, alloc_buffer);
		return -1;
	}

	err = virtqueue_driver_signal(send_virtqueue);
	if (err != 0) {
		ZF_LOGE("Client send signal failed");
		return -1;
	}
	return 0;
}

void handle_recv_data(char *recv_data, size_t recv_data_size)
{
```

```
	int err;

	/* Check if there is data still waiting in the send virtqueue */
	int send_poll_res = virtqueue_driver_poll(send_virtqueue);
	if (send_poll_res) {
		/* makes call to send_outgoing_packet() */
		handle_send_callback(send_virtqueue);
	}

	struct ethhdr *rcv_req = (struct ethhdr *) recv_data;
	if (ntohs(rcv_req->h_proto) == ETH_P_ARP) {
		create_arp_req_reply(recv_data, recv_data_size);
	} else if (ntohs(rcv_req->h_proto) == ETH_P_IP) {
		char ip_packet[ETHERMTU];
		memcpy(ip_packet,
			recv_data + sizeof(struct ethhdr),
			recv_data_size - sizeof(struct ethhdr));
		print_ip_packet(ip_packet, recv_data_size - sizeof(struct ethhdr));
		/* makes call to send_outgoing_packet() */
		create_icmp_req_reply(recv_data, recv_data_size);;
	}
}

void handle_recv_callback(virtqueue_device_t *vq)
{
	volatile void *buf = NULL;
	size_t buf_size = 0;
	int err = virtqueue_device_dequeue(vq,
	                                   &buf,
	                                   &buf_size);
	if (err) {
		ZF_LOGE("Client virtqueue dequeue failed");
		return;
	}

	char *recv_buffer = (char *)buf;
	handle_recv_data(recv_buffer, buf_size);

	err = virtqueue_device_enqueue(recv_virtqueue, recv_buffer, buf_size);
	if (err) {
		ZF_LOGE("Unable to enqueue used recv buffer");
		return;
	}

	err = virtqueue_device_signal(recv_virtqueue);
	if (err) {
		ZF_LOGW("Failed to signal on recieve virtqueue");
	}
}
```

```
void handle_send_callback(virtqueue_driver_t *vq)
{
	volatile void *buf = NULL;
	size_t buf_size = 0;
	int err = virtqueue_driver_dequeue(vq,
	                                   &buf,
	                                   &buf_size);
	if (err) {
		ZF_LOGE("Client virtqueue dequeue failed");
		return;
	}
	/* Clean up and free the buffer we allocated */
	camkes_virtqueue_buffer_free(vq, buf);
}

void ping_wait_callback(void)
{
	int err;
	int recv_poll_res = virtqueue_device_poll(recv_virtqueue);
	if (recv_poll_res) {
		handle_recv_callback(recv_virtqueue);
	}
	if (recv_poll_res == -1) {
		ZF_LOGF("Client recv poll failed");
	}

	int send_poll_res = virtqueue_driver_poll(send_virtqueue);
	if (send_poll_res) {
		handle_send_callback(send_virtqueue);
	}
	if (send_poll_res == -1) {
		ZF_LOGF("Client send poll failed");
	}
}

int run(void)
{
	ZF_LOGE("Starting ping echo component");

	/* Initialise recv virtqueue */
	int err = camkes_virtqueue_device_init(&recv_virtqueue, 0);
	if (err) {
		ZF_LOGE("Unable to initialise recv virtqueue");
		return 1;
	}

	/* Initialise send virtqueue */
	err = camkes_virtqueue_driver_init(&send_virtqueue, 1);
	if (err) {
		ZF_LOGE("Unable to initialise send virtqueue");
```

```
		return 1;
	}
	return 0;
}
```

---

In this example, the `ping_wait_callback` function handles both the callback for receiving connection from the incoming virtqueue, as well as the callback for the sending connection to the outgoing virtqueue.