Issues and Semantics for AADL Runtime Services

Brian R Larson John Hatcliff
{brl,hatcliff}@ksu.edu
The Laboratory for Specification, Analysis, and Transformation of Software (SAnToS)

June 7, 2017

Motivation

Kansas Statue University and Adventium Labs are working together on the ISOSCELES[1] under Department of Homeland Security to create an intrusion-resistant, open platform for medical devices.

Small medical device companies could use the platform by attaching sensors and actuators to perform medical functions relying on the platform to provide appropriate security.

[1] can't remember acronym

ISOSCELES uses AADL

Because ISOSCELES is to be a *platform*, we think it should provide AADL Runtime Services and standard dispatch protocols and thread scheduling (§5.4) together with some form of mode switching.

We expect devices to be able to “break glass” in an emergency to operate as a stand-alone device sans networking and authorization. This is a perfect application for (simple) AADL modes.

So, we endeavored to study AADL Runtime Services such that users of the platform (of the tools they use) can use AADL Runtime Services, hoping to discern or create formal semantics such that every platform implementing them would behave the same way.

Issues with modes and thread scheduling will be subject of later discussions.

We Failed

We failed in our attempt to formalize AADL runtime services, and find that even thread scheduling defined by timed automata (for all dispatch protocols) to be difficult.

This presentation summarizes our findings (or not).

We are interested in what RAMSES and Ocarina generate and what PolyOrb-HI does with them.

AADL Runtime Services Have Issues

AADL Runtime Services are summarized in section A.9 of the core AADL standard.[2]

Each service is defined as an AADL subprogram, although with many `<implementation-dependent ...>` data types.

There must be some library (API) defined for each language (and each target RTOS) which uses that language's grammar, that performs each (suitably named) subprogram.

From some of the issues found, I doubt anyone has actually implemented a platform that provides *all* of them.

[2] these from as5506-2.2-03102016.pdf

A.9 Predeclared Runtime Services

> *Two sets of runtime services are predeclared. The first set declares service subprograms that can be called by the application source text directly. The second set declares service subprograms that are intended to be called by an AADL runtime executive that can be generated from an AADL model, thus is not expected to be used directly by application component developers.*

Actually, neither set will (should) appear in user-written code, but runtime executive services need to be invoked somehow.

Port Input and Output

All of the “Application Runtime Services" deal with port input and output.

Invocation of subprograms are supported by AADL runtime services. There are blocking, and non-blocking properties that the target RTOS must accommodate.

Send_Output

A Send_Output runtime service allows the source text of a thread to explicitly cause events, event data, or data to be transmitted through outgoing ports to receiver ports.

The Send_Output service takes a port list parameter that specifies for which ports the transmission is initiated. The send on all ports is considered to occur logically simultaneously.

Send_Output is a non-blocking service.

An exception is raised if the send fails with exception codes indicating the failing port and type of failure.

```
subprogram Send_Output
features
  OutputPorts: in parameter <implementation-dependent port list>;
  -- List of ports whose output is transferred
  SendException: out event data; -- exception if send fails to complete
end Send_Output;
```

Send_Output is Not Really Necessary

The Await_Dispatch service (discussed later) has a list of ports "whose output is sent at completion/deadline", so Send_Output is not really necessary.

The AADL default is at completion, but deadline is often used for periodic tasks where the deadline is half of the period.

No BA/BLESS behavior that uses Await_Dispatch to suspend execution upon entering a complete state need ever invoke Send_Output .

Put_Value

A `Put_Value` runtime service allows the source text of a thread to supply a data value to a port variable.

This data value will be transmitted at the next `Send_Output` call in the source text or by the runtime system at completion time or deadline.

```
subprogram Put_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: in parameter; -- value to be stored
  DataSize: in parameter; -- size in bytes (optional)
end Put_Value;
```

Note that the target is `requires data access` which in AADL is a constant pointer–not a port identifier–and does not specify its type.

Also the `DataValue` type is not specified.

Implementation?

So how would `Put_Value` be implemented in Ada or C with an untyped reference, and an untyped parameter?

This would be used to perform port output in BA/BLESS: `p!(e)`

What would be the invocation generated?

```
temp := e;
Put_Value(p_reference,temp,sizeof(temp));
```

This becomes more confusing when ports can be (1-dimensional) arrays. Can an array of values be assigned to an array of ports in one go, or does a loop need to cycle through them?

Receive_Input

A `Receive_Input` runtime service allows the source text of a thread to explicitly request port input on its incoming ports to be frozen and made accessible through the port variables.

Any previous content of the port variable is overwritten, i.e., any previous queue content not processed by `Next_Value` calls is discarded.

The `Receive_Input` service takes a parameter that specifies for which ports the input is frozen. Newly arriving data may be queued, but does not affect the input that thread has access to (see Section 9.1).

`Receive_Input` is a non-blocking service.

```
subprogram Receive_Input
features
  InputPorts: in parameter <implementation-dependent port list>;
  -- List of ports whose input is frozen
end Receive_Input;
```

Not Really Necessary Either

The `Await_Dispatch` service has a list of ports "whose whose input is received at dispatch", so `Receive_Input` is not really necessary.

No BLESS behavior will ever need to use `Receive_Input` because its input port values are always frozen at dispatch.

BA has some facility to freeze ports during execution, but nobody has provided an explanation why that would be useful.

Discarding queue content by `Receive_Input` also seems strange.

Get_Value

A Get_Value runtime service shall be provided that allows the source text of a thread to access the current value of a port variable. The service call returns the data value.

Repeated calls to Get_Value result in the same value to be returned, unless the current value is updated through a Receive_Input call or a Next_Value call.

```
subprogram Get_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: out parameter; -- value being retrieved
  DataSize: in parameter; -- size in bytes (optional)
end Get_Value;
```

Fine for Data Ports

`Get_Value` works fine for data ports, `p?(v)`, which can't have a queue of values.

Despite the poor definition of its parameters, `Get_Value` for data ports makes sense. It returns the last data value received, regardless.

It gets tricky for event data ports, which can have queues. What does `Get_Value` do when there's nothing in the queue?

Next_Value

A `Next_Value` runtime service shall be provided that allows the source text of a thread to get access to the next queued element of a port variable as the current value.

A `NoValue` exception is raised if no more values are available.

In the case of event data ports each data value is retrieved from the queue through the `Next_Value` call and made available as port variable value.

Subsequent calls to `Get_Value` or direct access of the port variable will return this value until the next `Next_Value` call.

```
subprogram Next_Value
features
  Portvariable: requires data access; -- reference to port variable
  DataValue: out parameter; -- value being retrieved
  DataSize: in parameter; -- size in bytes (optional)
  NoValue: out event port; -- exception if no value is available
end Next_Value;
```

Many Questions

Must an event data port use `Get_Value` for the first value in the queue because `Next_Value` throws it away?

If the first `Next_Value` retrieved the first element of the queue, what does `Get_Value` return if executed before the first `Next_Value`.

What does the `NoValue` out event port do?

How should `NoValue` be handled, because this is being invoked by source code–not by an AADL component that knows about ports and events?

Get_Count

A Get_Count runtime service shall be provided that allows the source text of a thread to determine whether a new data value is available on a port variable, and in case of queued event and event data ports, who[3] many elements are available to the thread in the queue.

A count of zero indicates that no new data value is available.

```
subprogram Get_Count
features
  Portvariable: requires data access; -- reference to port variable
  CountValue: out parameter BaseTypes::Integer; -- content count of port variable
end Get_Count;
```

[3] how

Counting Queued Elements for Event Data Ports

`Get_Count` can be used to tell if there are any, or how many elements are in the queue.

Can there be multiple queued *events*?

This seems strange that multiple events would arrive before dispatch. Perhaps a periodic thread wants to count events since last completion.

Get_Count Indicates Freshness for Data Ports

> *The* `Get_Count` *will return the value 1 if the value has been updated, i.e., is fresh. If the data is not fresh, the value zero is returned.*

What if the new value received at a data port is the same as the previous value?

Is it that the sender has sent a value since previous completion, or that the value is different?

Updated Indicates New Values Arrived, but not put in queue

A Updated runtime service shall be provided that allows the source text of a thread to determine whether input has been transmitted to a port since the last Receive_Input service call.

```
subprogram Updated
features
  Portvariable: in parameter <implementation-dependent port reference>;
    -- reference to port variable
  FreshFlag: out parameter BaseTypes::Boolean; -- true if new arrivals
end Updated;
```

BaseTypes should be Base_Types.

Note here that Portvariable is “implementation-dependent port reference" where others its requires data access.

Runtime Executive Services

> *The following are subprograms may be explicitly called by application source code, or they may be called by an AADL runtime system that is generated from an AADL model.*

This directly contradicts the first paragraph which says

> *The second set declares service subprograms that are intended to be called by an AADL runtime executive that can be generated from an AADL model, thus is not expected to be used directly by application component developers.*

Locks in CPS are hazards

(Brian's Opinion)

Don't use `Get_Resource` or `Release_Resource`.

BA also has operators for entering and leaving critical sections, but there are no runtime services that perform them.

Besides there are usually more than just one critical section, if there are any, with no way in BA to indicate which critical section is entered of left.

Any functionality provided by shared resources can be accomplished by encapsulation in threads which manage access fairly and without hazards.

Resource locking is legacy junk which should be deprecated, and used less than `goto`.

Await_Dispatch

```
subprogram Await_Dispatch
features
-- List of ports whose output is sent at completion/deadline
OutputPorts: in parameter <implementation-defined port list>;
-- List of ports that can trigger a dispatch
DispatchPorts: in parameter <implementation-defined port list>;
-- list of ports that did trigger a dispatch
DispatchedPort: out parameter < implementation-defined port list>;
-- optional function as dispatch guard, takes port list as parameter
DispatchConditionFunction: requires subprogram access;
-- List of ports whose input is received at dispatch
InputPorts: in parameter <implementation-defined port list>;
end Await_Dispatch;
```

The `Await_Dispatch` runtime service is called to suspend the thread execution at completion of its dispatch execution.

It is the point at which the next dispatch resumes.

The service call takes several parameters.

It takes a `DispatchPort` *list and an optional trigger condition function to identify the ports and the condition under which the dispatch is triggered. If the condition function is not present any of the ports in the list can trigger the dispatch.*

It takes a `DispatchedPort` *as out parameter to return the port(s) that triggered the dispatch.*

It takes `OutputPorts` *and* `InputPorts` *as port lists.*

`OutputPorts`*, if present, identifies the set of ports whose sending is initiated at completion of execution, equivalent to an implicit* `Send_Output` *service call.*

`InputPorts`*, if present, identifies the set of ports whose content is received at the next dispatch, equivalent to an implicit* `Receive_Input` *service call.*

The Big Wu

`Await_Dispatch` is the most important runtime service. It is invoked when entering a `complete` state, and provides the point at which execution resumes (also indicated by the `Compute_Entrypoint` property.

Its lists of `OutputPorts` and `InputPorts` obviate the need for `Send_Output` and `Receive_Input`.

The list of `DispatchPorts` tells the scheduler what ports to consider when deciding to dispatch by applying the `DispatchConditionFunction`.

Dispatch Condition Function

A single `complete` state may have several, outgoing, transitions with different dispatch conditions.

When the `complete` state was entered, and `Await_Dispatch` called, there is only one `DispatchConditionFunction`.

This could be a problem, except both BA and BLESS restrict dispatch conditions to disjunction of conjunctions. The disjunction of multiple dispatch conditions will also be a disjunction of conjunctions. So the scheduler need only evaluate disjunction of conjunctions of ports in the `DispatchPorts` list. This restriction on `DispatchConditionFunction` should be explicit in the definition of `Await_Dispatch`.

AADL has no Functions

AADL defines no functions (that return a value), only subprograms.

For BLESS, if a subprogram has only `in` parameters, except for the last which must be `out` then the subprogram can be invoked like a function in expressions.

Raise_Error

A Raise_Error runtime service shall be provided that allows a thread to explicitly raise a thread recoverable or thread unrecoverable error.

Raise_Error takes an error type identifier as parameter.

```
subprogram Raise_Error
features
  errorID: in parameter <implementation-defined error type>;
end Raise_Error;
```

However, it doesn't define what happens when the error is raised!

This recalls the controversy over exception raising and handling in BA/BLESS. (BA has nothing on exceptions.)

AADL needs a coherent mechanism for raising and handling exceptions.

Get_Error_Code

A Get_Error_Code runtime service shall be provided that allows a recover entrypoint to determine the type of error that caused the entrypoint to be invoked.

```
subprogram Get_Error_Code
features
  errorID: out parameter <implementation-defined error type>;
end Get_Error_Code;
```

Which recover entrypoint should be used when an error occurs to use Get_Error_Code?

Await_Result

A Await_Result runtime service shall be provided that allows an application to wait for the result of a semi-synchronous subprogram call.

```
subprogram Await_Result
features
  CallID: in parameter <implementation-defined call ID>;
end Await_Result;
```

Await_Result will never be used by a BLESS behavior (subprogram invocations must take negligible time to perform). I don't recall BA behavior for semi-synchronous subprogram calls.

Timeout

Although Jerome disagreed, my advocacy of an AADL runtime service for timeout dispatch triggers has been informally accepted.

The `Timeout` service acts like a hardware timer, with a `Reset` event input, an `Expired` event input, and a `Duration` data port to indicate how long the timer should run before expiration.

```
device Timeout
features
  Reset : in event port;  --restart the timer, sample Duration
  Duration : in data port BLESS_Types::Time;  --time of expiration after reset
  Expired: out event port;  --a reset occurred the duration time previously, and not since
end Timeout;
```

The AADL property `Time` cannot be used for types of ports. `BLESS_Types::Time` is used here because it has already been suitably defined. AADL v3 could predeclare data types which includes `Time`.

Timestamp

The BLESS concept of `now` has been (informally) accepted as necessary. Its implementation requires a timestamp to provide the current time.

AADL has already defined such a value: “ST"".[4]

```
subprogram Timestamp
features
  CurrentTime : out parameter BLESS_Types::Time;   --the current time
end Timestamp;
```

[4] §5.4.6 System Synchronization Requirements

AADL Runtime Services are Too Loosely Defined to be Implemented Consistently

AADL runtime services need formal definitions.

This will be hard, and need acceptance by Peter Feiler, whose understanding of the services may be different than the standard text.