AADL BA errors producing operators issues

Denis Buzdalov
ISPRAS

February 2, 2015

1 Background

1.1 Ports runtime support

1.1.1 Ports in AADL core

In AADL actions with ports are performed in two ways: though array-in-memory representation and though special *runtime services* provided by the standard. Those services are `Send_Output`, `Put_Value`, `Receive_Input`, `Get_Value`, `Get_Count`, `Next_Value` and `Updated`.

Some of the services are *receiving* and work only with `in` and `in out` ports. Some of them are *sending* and work only with `out` and `in out` ports.

Two of them — `Send_Output` and `Next_Value` — may finish not properly and definitions of these calls contain events for exceptions.

Also, overflow handling may produce an exception and require recovery when too much data has arrived to a port while this port has `Overflow_Handling_Protocol` property set to the `Error` value.

Putting a value to a port At the current version of the AADL core `Put_Value` call cannot produce errors on event data ports since (as it was discussed in Valencia at 2014) `out event data` ports behave the same as `out data` ports, i.e. they have only a single value and `Put_Value` rewrites it without producing an error. It also have been said that this semantics does not correspond to some use cases (like output ports in AFDX) and have to be changed.

In case of putting values to the output queue, overflow handling procedure have to be introduced here. This means that an error can be produced during the `Put_Value` execution for `out event data` ports.

Overwriting during freezing Also, AADL core says not really clearly about existing data overwriting and discarding during freezing of a port.

Consider a case when a port is frozen with no new input data. One variant of interpretation says that such freezing is not changing the current value (which is returned by the `Get_Value` call) considering this data to be *not fresh* (i.e. `Get_Count` returning zero). Another variant of interpretation is that getting value is not legal in this case.

Nevertheless, it is completely not clear why the `Put_Value` call does not contain any exception output for case of never frozen port or frozen port which has never received data.

1.1.2 Ports in AADL BA

One can work with ports in Behavior Annex only using special *port operators and operations* and making BA runtime system automatically freeze ports when it is needed.

Meaning	BA operator	Corresponding service call
put	`p := v`	`Put_Value`
send	`p!`	`Send_Output`
put and send	`p!(v)`	`Put_Value` and then `Send_Output`
freeze	`p>>`	`Receive_Input`
test of updated	`p'updated`	`Updated`
get unread count	`p'count`	`Get_Count`
test of freshness	`p'fresh`	`Get_Count` > 0
read	`p`	`Get_Value`
read and dequeue	`p?`	`Get_Value` and then `Next_Value`
read and dequeue	`p?(var)`	`Get_Value` and then `Next_Value`

Table 1: Approximate correspondence of BA operations and AADL core service calls

In general, BA port operators and operations correspond semantically to AADL core service calls. This correspondence can be found in the table 1. It is not really strict and will be discussed below.

Freezing operator, `updated`, `count` and `fresh` operations are defined as a simple usage of appropriate service calls.

Sending operator is defined as if it simply calls `Send_Output` but nothing is said about `SendException` handling (erratum `D.5-06`).

Putting operator is defined as "`p := v` assigns the data value `v` to the port variable of a data or event data port `p`". This practically does same as the `Put_Value` call does, but this definition does not support future extension of semantics of this call for event data ports.

Obviously, operator of consequent putting and sending inherit all properties of both operators.

Reading operation may differ from the `Get_Value` call only in the case of freezing an input port with no new input data. BA says that reading operator still returns the same value which is considered to be *not fresh*. This is the first interpretation in the description at the "Overwriting" paragraph of the section 1.1.1.

Still, reading operator does not support any potential exception management of reading from not ready port (not ever frozen or empty or whatever).

The biggest difference between BA operator and semantically according service call is reached for the dequeuing operators. Dequeuing service call has an explicit exception event port while dequeuing operations in BA are defined in a way that they do not dequeue if they cannot do it and no error appears.

1.2 Threads and exceptions

1.2.1 Exceptions in AADL threads

AADL threads, in general case, have *nominal* and *recovery* entrypoints. Execution of a thread goes to the nominal one on the beginning of dispatch. If some exception occurs during the nominal execution, thread goes to the recovery entrypoint.

If recovery section successfully manages the exception, dispatch of the thread ends as if the nominal execution completes. If recovery execution is unable to complete successfully, thread is aborted and put onto the *halted* state. After halting thread may be loaded and initialized again.

1.2.2 Threads from BA point of view

States of BA automaton are listed explicitly.

BA obliges to mark one state as *initial*. This state will be the current state after the component initialization.

All states in BA are divided into two groups: *complete* and *not complete*. Only complete states can be current at the end of a dispatch.

BA does not declare any implicit states. At the same time, BA does not have any facilities for explicit declaration of error or recover states.

2 Problem statement

There are some statements that seems to be problems at some point of view.

One of them is actually the core language problem that closely touches behaviour definitions. The problem is that it is not clear whether exceptions of service calls are somehow connected automatically with exceptions of a thread.

Let us consider that if exceptional output of a system call is not connected explicitly somewhere, then exception of the system call puts the running thread into the recovery state. This consideration looks rational and sound with definition of system calls and exceptions in threads.

Now we consider problems that touch errata `D.5-04`, `D.5-06` and `D.5-05`. The main problems of the discussed area in the Behavior Annex are the following.

(P1) BA operators and operations are very similar but not exactly correspond to AADL core service calls. This makes a little mess in understanding when both approaches are used for expressing behaviour in a model.

(P2) There is a try to avoid errornous behaviour of BA operators. Unfortunately, it makes unable to represent management of errors in situations when they cannot be cut off beforehand (like exception during sending from `out` ports). It seems to be very useful in behavior modelling.

(P3) No ability to represent special error state or recovery entrypoint in BA code which is intended to be the stange situations handler. This can be useful when one uses non-standard subprograms which can raise exceptions using their special `out` ports. Not having any special recovery procedure it is really inconvenient to model behaviour of interruption-like recovery states (which would return to different states depending on the state before an error).

3 General principles

One of BA design ideas is to try to eliminate implicit things. Thus it defines operators in a way that problem P1 exists.

Generally implicit things in languages cannot be considered strongly a bad thing. Basing on the experience of programming languages (which also are used to express behaviour), usage of implicit things can raise cleanness and understandability of code. Of course, bad usage of implicits would make code not understandable and would lead to errors.

4 Possible solutions

We will condider both partial and full solutions of the discussed problems.

Obviously, partial solutions are simpler and easier. They require less changes and effort. But still, some problems would leave and probably even would become heavier.

4.1 The minimal change

The simplest thing that can be done is to redefine all BA operators in the way that they cannot produce errors.

Particularly, sending operator in this solution have to be explicitly redefined that it calls `Send_Output` service call and manages the possible exception in some way. Similarly, some default value have to be introduced for reading operation for ports that have not ever received any data.

In this solution, dequeueing operator should not be redefined.

This solution partually solves the problem P1 because it should be explicitly written that BA operators should not be thought as a BA syntax for AADL service calls. But problems P2 and P3 are leaving unsolved, moreover the problem P2 becomes deeper since it is not really clear how exception-free operation of sending can be implemented though the core call `Send_Output` which may do nothing in case of exception.

4.2 Closer to the core

We can get closer to the core. The way to do this is to redefine BA operators in a way that they can be used as the core service calls. This would completely solve the problem P1.

The difficulty of this approach is that some (at least primitive) exception handling have to be invented to BA. This is a legacy of the core where exception handling exists.

There are several ways to organize it.

4.2.1 Explicit in-place handling

Each operator that can produce an exception can be guarded with some exception handler.

There are at least some variants how can this be implemented.

Such handler may be set only for some particular kind of exception (like `NoValue` or `SendingException` from the core) or to any exception that can be raised by the operator it guards.

What is allowed to put into the exception handler is another question to discuss considering this approach. This question is independent with the previous. At least it should be decided whether handler block can contain

- exception-free BA operators;
- operators which can produce exceptions;
- nested exception-handling blocks;
- forced transitions to other states.

Another independent question is whether there should be some default handler or each operator that can produce an exception have to be excplicitly guarded.

This approach basicly solves the problem P2.

4.2.2 Special recovery state

A notion of the special *recovery state* can be introduced.

Similarly to the provious approach, it is not clear in general case whether only one recovery state should be invented or some recovery states should be used for different kinds of errors.

If the single recovery state approach is chosen, it is not clear how can a single recovery state know which type of error has appeared.

But still, the semantics of such recovery state should be set similar to the recovery entrypoint of the core AADL. This means that recovery state can initiate correct dispatch ending as well as it can consider that it cannot manage error and thus ruine the thread.

Obligaton issue Also, there is a question whether the recovery state have to be defined for each thread or not.

Actually, there is no need in the recovery state if no dangerous operator is used in the thread.

Not having an obligation of explicit recovery states is good for simple behaviours and when all dangerous operators are called only in good conditions (e.g. they are in conditional operator block which guards them from raising an exception).

In such case recovery state can be defined implicitly. Entering the default recovery state means the thread halting. Such implicit state means that behaviour author does not expect not handled exceptions, i.e. no operator should produce it. This makes simple code cleaner and more understandable.

In the case of complicated exception handling, recovery state can be defined explicitly.

4.2.3 Combination

The recovery state approach seems to provide a good solution for the problem P3.

Also, a combination of two previous approaches can be used.

It is suggested to have an implicit recovery state (with the fault behaviour) with availability to define it explicitly and to define appropriate recovery behaviour.

It also suggested in the combination approach to have in-place exception handlers that can both change the state and perform transitions. The default exception handler perform transition to the recovery state.

This approach gives the most flexibility and solves all mentioned problems and makes BA basis closer to AADL. But it is much harder to implement. Also, it may scare those BA users which did not get used to exceptions.

5 Conclusion

Some problems related to discussed errata were mentioned.

Some solutions with lots of undecided variants were suggested.

It is suggested to think of problems and to consider costs of problems and solutions and to decide which of them should be reflected in the standard.