Reconciliation of the Behavior Annex (BA) with Behavior Language for Embedded Systems with Software (BLESS)

Brian R Larson

May 3, 2016

Intent

In 2015, reconciliation of the Behavior Annex (BA) and Behavior Language for Embedded Systems with Software was begun.

Although the reconciliation efforts narrowed the differences between the languages, full synthesis was not achieved.

This presentation explores the remaining differences.

BLESS Genesis

While reviewing the proposed BA standard document, Brian Larson saw similarities with earlier work transforming concurrent programs with proof outlines into complete, formal proofs.

He thought that by augmenting BA with assertions (which would be ignored like comments by a compiler) his proof engine could be adapted to provide formal proofs that state machine behavior met its specification.

He started with the proposed BA standard document and began augmenting it together with a proof engine that transforms programs annotated with assertions to be a proof outline into a complete formal proof.

The result was the Behavior Language for Embedded Systems with Software (BLESS).

What is Proof?

For many, *proof* is a verb meaning convincing evidence, such as presented at judicial proceedings.

Here, *proof* is a noun meaning sequence of theorems, each of which is given or axiomatic, or derived from earlier theorems by sound inference rules. Thus, by induction, every theorem in the sequence will be true. Call this *inductive proof* if needed to distinguish it from other forms of “proof”.

Such *proof* (noun) is a stand-alone entity, independently true from its creator–human, machine, or combination of both.

In contrast, popular theorem prover like Coq produce scripts tactics they call “proof". However, looking at a Coq “proof” cannot determine what is being proved! Only in the context of the prover, together with huge libraries of tactics, does a Coq “proof” have any meaning at all!

BA and BLESS have Different Goals

The intent for BA is to *model* thread behavior–an abstraction for early analysis–to later be replaced by software written in conventional languages as the actual design artifacts.

The intent for BLESS is to replace traditional programming, treating programs, their specifications, and their executions as mathematical objects.

Therefore, every BLESS language construct was given formal semantics from its inception, which were corrected, refined, and extended as the BLESS proof tool was used to transform programs with proof outlines into deductive proofs.

BA Errata and JP's Formal Semantics

Since first adoption of BA as an AADL annex standard, errata discovered have been managed by Etienne Borde, although many have contributed. The results of these efforts will be balloted following this meeting.

During this time Jean-Pierre Talpin defined formal, state-machine semantics for BA for timing analysis with Polychrony.

JP's formal semantics for BA were in many ways, a different means for describing the same formal semantics as BLESS.

BA ⊂ BLESS

In the reconciled BLESS language, every BA behavior (corresponding to the BA annex document to be balloted imminently) is also a BLESS program.

Because, JP's formal semantics have been incorporated in BLESS, the behavior will be the equivalent.[1] We have been working together such that BLESS programs can be transformed into Signal for analysis with Polychrony. JP's team is further developing state machine semantics for Signal, to which translations from BLESS have contributed.

[1] JP's semantics are yet incomplete, but those that have been defined, have been merged with BLESS semantics

BA migration to BLESS

So, while the errata-corrected BA annex document goes to ballot, the BLESS Annex Document is being prepared reflecting the merged formal semantics. This will not be “BA v2” but a formalized superset.

Any behaviors developed in BA can seamlessly migrate to BLESS, having both correctness proofs, *and* timing analysis with Polychrony.

Additional BLESS Constructs

1) Assertions: attached to states, and interspersed throughout actions

2) Port event timeout: must have as dispatch trigger–not just dispatch condition

3) Null

4) Conditional expression

5) Function invocation

Port Event Timeout

BA added it, but as a dispatch condition–not a dispatch trigger

```
dispatch_trigger ::= in_event_port_name | in_event_data_port_name
  | port_event_timeout_catch

port_event_timeout_catch ::=
  timeout ( port_identifier { [ or ] port_identifier }* )
    behavior_time
```

Null

Denis Buzdalov has made some strong arguments for the elimination of `null`.

It's semantics can be considered 'uninitialized' or 'unknown'.

Initially took it out of the BLESS grammar, but then found that the PCA pump drug library used it in its proof outline to distinguish the uninitialized library, which then must be loaded before used.

`null` is also used to define outputs during ServiceOmission errors.

Conditional Expression

```
conditional_expression ::=
  ( boolean_expression_or_relation ?? expression : expression )
```

Conditional expression was added to the grammar and the proof engine in 2012 because an example thread behavior could not be proved correct otherwise.

Tried Ada-style conditional expression, but expressions with multiple conditions became verbose and difficult to understand.

```
conditional_expression ::=
  if boolean_expression_or_relation then
  expression else expression
```

Function Invocation

```
function_call ::= { package_identifier :: }*
    function_identifier ( $ [ function_parameters ] )
```

Wanted to include function calls in expressions like BLESS predecessor language.

Simultaneous Assignment

```
simultaneous_assignment ::=
  ( variable_name{ , variable_name }+
  := ( expression | record_term )
  { , ( expression | record_term ) }+ )
```

Reason: As in Dijkstra's guarded commands:

```
(iSeg',nSeg',i',v',s',a',b',iMA' :=
  iSeg, nSeg,i,v,s,a,b,iMA)
```

to set 'next' values together

Alternative

```
alternative ::=
  if guarded_action { [] guarded_action }+ fi
  |
  if ( boolean_expression_or_relation ) behavior_actions
  { elsif ( boolean_expression_or_relation ) behavior_actions }*
  [ else behavior_actions ]
  end if
```

Match guarded commands symmetric if-fi, and eliminate problematic ‘else’.

Unified Types

BLESS has defined type system with mappings to both AADL property types and Data Model annex.

Will incorporate results of Unified Type System work.

when throw catch

```
catch_clause ::= catch { ( exception_label : basic_action ) }+
```

```
exception_label ::= { exception_identifier }+ | all
```

Catch does not need try. Implicit domain is the block containing the catch.

Can put out event data on error port: `error!("message")`

Separate Assertion Annex?

Have prepared a separate Assertion language annex document (with Xtext parser).

Separate Assertion document for behavior interface specification language (BISL) and allow other tools to use assertions.

Assertion language is more stable, and could be balloted soon–perhaps with BA.

Or wait for BLESS annex document.

BLESS will be updated with more of JP's formal semantics, exception handling, and unified type system.

BLESS document(s) on Wiki

`https://wiki.sei.cmu.edu/aadl/index.php/Standardization#SAE_AADL_AS5506B_Errata`

Will upload BLESS/Assertion annex documents & syntax cards to wiki for consideration.

Update site for both Xtext editor and BLESS proof tool to be hoisted this weekend:

`http:www.multitude.net/update`