BLESS Behavior Correctness Proof as Convincing Verification Artifact

Brian R Larson
brl@multitude.net
Ehsan Ahmad
e.ahmad@seu.edu.sa

June 16, 2023

The BLESS Methodology applies to an architectural *model* of a cyber-physical system using the Architecture Analysis and Design Language (AADL).

The BLESS Methodology creates programs together with deductive proofs that every possible program execution will conform to its specification.

Colloquially, “proof” usually means “evidence”, with perhaps some reasoning about it.

Proof is (or should be) an *argument* to *convince* people of its conclusion.

A *deductive proof* is a sequence of theorems, each of which is given, or an axiom, or derived from theorems in the sequence by some reason.

The last theorem is the conclusion: what is being proved.

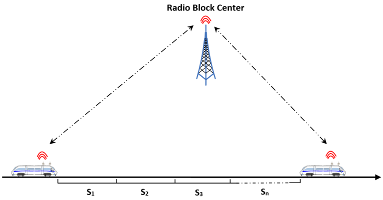
When

- the **axioms** have been proved (by some other means) to be tautology (always true),
- the **reasons** are inference rules proved to be sound (derive true facts from true facts), and
- the **givens** appropriately describe the subject of the argument,

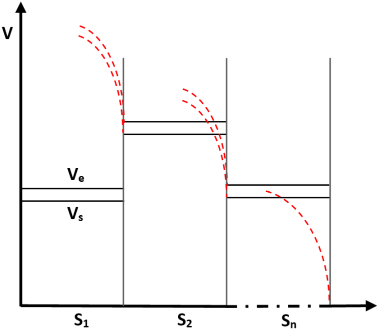
people can decide if they believe the conclusion and with what confidence.

BLESS proofs are deductive proofs that every possible program execution conforms to its specification.

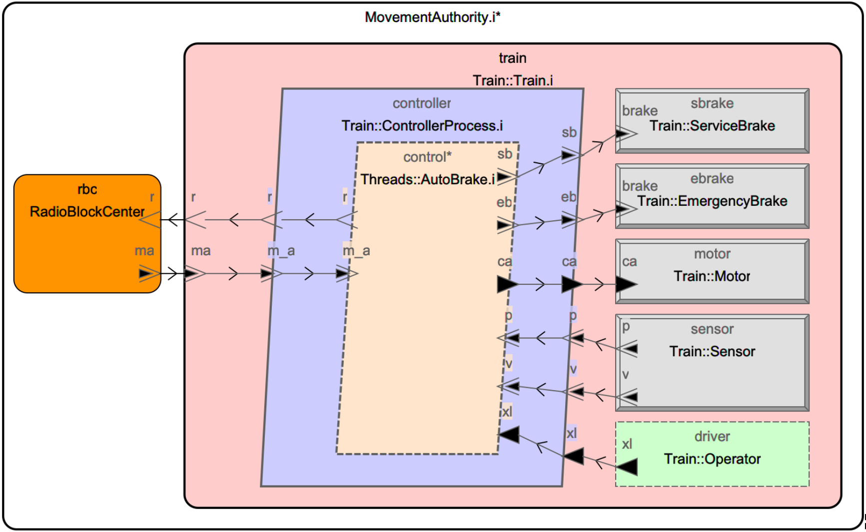
For an AADL architecture, atomic component behaviors are proved to meet their specifications, and composite components are proved from the specifications and interconnection of their subcomponents.



Chinese Train Control System Level 3 (CTCS-3) Movement Authority scenario allows trains to move only when they have been given a movement authorization (MA) for a specific length of track which is divided into segments.



A train operator normally controls the acceleration (and thus speed) of the train.
For safety, the train's service brake is automatically applied if the train velocity exceeds a safe limit, V_s for the train's current segment of its MA.
If the service brake fails so that the train velocity exceeds V_e, the emergency brake is automatically applied.
If the next segment has lower velocity limits, automatic braking may applied (dotted curves).


MovementAuthority.i*
rbc
RadioBlockCenter
r
ma
train
Train::Train.i
controller
Train::ControllerProcess.i
control*
Threads::AutoBrake.i
r
m_a
sb
eb
ca
p
v
xl
sbrake
brake
Train::ServiceBrake
ebrake
brake
Train::EmergencyBrake
motor
ca
Train::Motor
sensor
p
v
Train::Sensor
driver
xl
Train::Operator

AADL properties of features (ports) declaratively specify behavior.

```
thread AutoBrake
 features
  sb: out event data port BLESS_Types::Boolean -- apply service brake
  {BLESS::Assertion => "<<SB() and not EB()>>";};
  eb: out event data port BLESS_Types::Boolean -- apply emergency brake
  {BLESS::Assertion => "<<EB()>>";};
  r: out event port; -- request new movement authorization (MA)
  m_a: in event data port CTCS_Types::movementAuthorization -- received MA
  {BLESS::Value => "<<returns movementAuthorization := RMA>>";};
  p: in event data port CTCS_Types::Position -- current measured position
  {BLESS::Value => "<<returns quantity m := POSITION>>";};
  v: in event data port CTCS_Types::Velocity -- current measured velocity
  {BLESS::Value => "<<returns quantity mps := VELOCITY>>";};
  xl : in data port CTCS_Types::Acceleration  --operator chosen acceleration
  {BLESS::Value => "<<returns quantity mpss := OPERATOR_XL>>";};
  ca : out  data port CTCS_Types::Acceleration  --acceleration to motor
  {BLESS::Value => "<<returns quantity mpss := TRAIN_XL()>>";};
 properties
  Dispatch_Protocol => Sporadic;
end AutoBrake;
```

Labelled assertions may be used to shorten predicates.

```
<<SB: :  --apply service brake
 v >= iSeg.v_n or v*v >= nSeg.v_n*nSeg.v_n + 2*b*(iSeg.e-p)>>
<<EB: :  --apply emergency brake
 v >= iSeg.v_e or v*v >= nSeg.v_e*nSeg.v_e + 2*e*(iSeg.e-p)>>
```

For threads, BLESS state-transition machines are deliberately similar to the Behavior Annex (BA) sublanguage of AADL.

Both have

- persistent local variables
- states: initial, final, complete, and execution
- transitions: dispatch conditions leaving complete state, boolean expressions otherwise

Each state may have an assertion of what is true about the system when in that state.

```
states
 Start: initial state --train stopped
 WaitFirstMA: complete state  --Wait for first MA
 CheckFirstMA: state     -- Check first MA
 MoveForward: complete state  --Move Forward
  << i<CMA.num_segments and iSeg=CMA.seg[i] and nSeg=CMA.seg[i + 1] and ma=CMA>>
 CheckMoveForward: state    --Check Move Forward
  << i<CMA.num_segments and iSeg=CMA.seg[i] and nSeg=CMA.seg[i + 1] and ma=CMA>>
 CheckForLastSegment: state  --check for last segment
  << iSeg = CMA.seg[i] and ma=CMA >>
 MoveForwardLastSegment: complete state  --Move Forward Last Segment, no new MA
  << i=CMA.num_segments and iSeg=CMA.seg[i] and nSeg=NULL_SEGMENT() and ma=CMA>>
 CheckMoveForwardLastSegment: state --check move forward last segment, no new MA
  << i=CMA.num_segments and iSeg=CMA.seg[i] and nSeg=NULL_SEGMENT() and ma=CMA>>
 GotNewMA: complete state  --on last segment, got new MA
  << i=CMA.num_segments and iSeg=CMA.seg[i] and nSeg=NEXT_MA.seg[1] and ma=CMA
   and next_ma=NEXT_MA >>
 CheckMATransition: state  --change to new MA?
  << i=CMA.num_segments and iSeg=CMA.seg[i] and nSeg=NEXT_MA.seg[1] and ma=CMA
   and next_ma=NEXT_MA >>
 FAIL: final state  --failure occurred
```

Requesting a movement authorization, and starting to move when received:

```
transitions
 Go: --request movement authorization
 Start -[]-> WaitFirstMA { r! }

 FirstMA: --dispatch before first MA
 WaitFirstMA -[on dispatch p]-> CheckFirstMA

 NotYet: --did not get requested movement authorization
 CheckFirstMA -[not m_a'fresh]-> WaitFirstMA

 GotFirstMA:  --received movement authorization
 CheckFirstMA -[m_a'fresh]-> MoveForward
  {  << AXIOM_CMA_IS_RMA() >>
  m_a?(ma)  --save received movement authorization
  ;  << ma=CMA >>
  i := 1  --first segment of new movement authorization
  ; << i=1 and ma=CMA >>
  iSeg := ma.seg[1] --set current segment to first segment
  ; << i=1 and ma=CMA and iSeg=CMA.seg[i]
      and AXIOM_NUM_SEG(ma:ma) >>
  nSeg := ma.seg[2]  --set next segment to second segment
  << i=1 and ma=CMA and iSeg=CMA.seg[i]
     and nSeg=CMA.seg[i+1] and AXIOM_NUM_SEG(ma:CMA) >>
  }
```

Automatic braking:

```
CheckSpeed:
MoveForward -[on dispatch p]-> CheckMoveForward
{ << i<CMA.num_segments and iSeg=CMA.seg[i]
  and nSeg=CMA.seg[i + 1] and ma=CMA and AXIOM_B()
  and AXIOM_E() and AXIOM_V(seg:iSeg)
  and AXIOM_V(seg:nSeg) >>
if  --exceed emergency brake velocity?
   (v >= iSeg.v_e  )~>
   { eb!(true) & sb!(false) & ca!(0 mpss) }
[]  --emergency brake for next segment?
   (v*v >= nSeg.v_e*nSeg.v_e + 2*e*(iSeg.e-p) )~>
   { eb!(true) & sb!(false) & ca!(0 mpss) }
[]  --exceed service brake velocity?
   (v >= iSeg.v_n and v < iSeg.v_e and
  v*v < nSeg.v_e*nSeg.v_e + 2*e*(iSeg.e-p) )~>
   { sb!(true) & eb!(false) & ca!(0 mpss) }
[]  --service brake for next segment?
   (v*v < nSeg.v_e*nSeg.v_e + 2*e*(iSeg.e-p)
     and v < iSeg.v_e
     and v*v >= nSeg.v_n*nSeg.v_n + 2*b*(iSeg.e-p) )~>
   { sb!(true) & eb!(false) & ca!(0 mpss) }
[]  --no auto brake needed
   ( v < iSeg.v_n
     and v*v < nSeg.v_n*nSeg.v_n + 2*b*(iSeg.e-p)
     and v*v < nSeg.v_e*nSeg.v_e + 2*e*(iSeg.e-p) )~>
   { sb!(false) & eb!(false) & ca!(xl) }
fi
}
```

For a sequential program S, beginning with predicate P being true applied to program variables, will terminate with predicate Q being true applied to program variables has been traditionally represented as a Hoare triple:

$$\{P\}\ S\ \{Q\}$$

Because in BLESS state-transition machines, curly brackets are used for action grouping, the verification condition for S is expressed as:

$$\ll P \gg\ \ S\ \ \ll Q \gg$$

Each non-final state has a verification condition.

Assertions of complete states must imply the thread's invariant.

Assertions of initial and execution states must imply the disjunction of conditions of outgoing transitions.

Each transition in a BLESS state machine has a verification condition:

$$\ll P \wedge b \gg \;\; S \;\; \ll Q \gg$$

where P is the assertion of the source state, Q is the assertion of the destination state, b is the transition condition, and S is the action of the transition.

The proof of AutoBrake.i has 658 theorems.

Each theorem says which axiom or inference rule is the reason it's true.

The last theorem says all the verification conditions have been proved.

```
Theorem (658)                                [serial 1002]
P [33] <<   >>
S [39] ->
Q [33] << AutoBrake.i proof obligations >>
Why created:  Initial proof obligations for AutoBrake.i
Solved by:  Component verification conditions
and theorems 1 2 3 5 9 11 12 15 17 21 25 26 27 106 313 315 322 332 357 358 549
  550 593 595 596 657:
Theorem (1) [serial 1003] used for:
  <<M(MoveForwardLastSegment)>> -> <<I>> from invariant I when complete state
  MoveForwardLastSegment has Assertion <<M(MoveForwardLastSegment)>>
  in its definition.
Theorem (2) [serial 1004] used for:
  <<M(WaitFirstMA)>> -> <<I>> from invariant I when complete state WaitFirstMA
  has Assertion <<M(WaitFirstMA)>> in its definition.
Theorem (3) [serial 1005] used for:
  <<M(MoveForward)>> -> <<I>> from invariant I when complete state MoveForward
  has Assertion <<M(MoveForward)>> in its definition.
Theorem (5) [serial 1006] used for:
  <<M(GotNewMA)>> -> <<I>> from invariant I when complete state GotNewMA
  has Assertion <<M(GotNewMA)>> in its definition.
Theorem (9) [serial 1007] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  CheckMoveForward, <<M(CheckMoveForward)>> -> <<e1 or e2 or . . . en>>
Theorem (11) [serial 1008] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  CheckMoveForwardLastSegment,
  <<M(CheckMoveForwardLastSegment)>> -> <<e1 or e2 or . . . en>>
Theorem (12) [serial 1009] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  Start,  <<M(Start)>> -> <<e1 or e2 or . . . en>>
Theorem (15) [serial 1010] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  CheckForLastSegment, <<M(CheckForLastSegment)>> -> <<e1 or e2 or . . . en>>
```

```
Theorem (17) [serial 1011] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  CheckFirstMA, <<M(CheckFirstMA)>> -> <<e1 or e2 or . . . en>>
Theorem (21) [serial 1012] used for:
  Serban's Theorem:  disjunction of execute conditions leaving execution state
  CheckMATransition, <<M(CheckMATransition)>> -> <<e1 or e2 or . . . en>>
Theorem (25) [serial 1013] used for:
   <<M(Start)>> A <<M(WaitFirstMA)>> for GoStart-[ ]->WaitFirstMA{A};
Theorem (26) [serial 1014] used for:
   <<M(WaitFirstMA) and x>> -> <<M(CheckFirstMA)>> for
   FirstMAWaitFirstMA-[x]->CheckFirstMA{};
Theorem (27) [serial 1015] used for:
   <<M(CheckFirstMA) and x>> -> <<M(WaitFirstMA)>> for
   NotYetCheckFirstMA-[x]->WaitFirstMA{};
Theorem (106) [serial 1016] used for:
   <<M(CheckFirstMA) and x>> A <<M(MoveForward)>> for
   GotFirstMACheckFirstMA-[x]->MoveForward{A};
Theorem (313) [serial 1017] used for:
   <<M(MoveForward) and x>> A <<M(CheckMoveForward)>> for
   CheckSpeedMoveForward-[x]->CheckMoveForward{A};
Theorem (315) [serial 1018] used for:
   <<M(CheckMoveForward) and x>> -> <<M(MoveForward)>> for
   SameSegmentCheckMoveForward-[x]->MoveForward{};
Theorem (322) [serial 1019] used for:
   <<M(CheckMoveForward) and x>> A <<M(CheckForLastSegment)>> for
   NextSegmentCheckMoveForward-[x]->CheckForLastSegment{A};
Theorem (332) [serial 1020] used for:
   <<M(CheckForLastSegment) and x>> A <<M(MoveForward)>> for
   NotLastSegmentCheckForLastSegment-[x]->MoveForward{A};
Theorem (357) [serial 1021] used for:
   <<M(CheckForLastSegment) and x>> A <<M(MoveForwardLastSegment)>> for
   IsLastSegmentCheckForLastSegment-[x]->MoveForwardLastSegment{A};
Theorem (358) [serial 1022] used for:
   <<M(CheckForLastSegment) and x>> -> <<M(FAIL)>> for
   PastLastSegmentCheckForLastSegment-[x]->FAIL{};
```

```
Theorem (549) [serial 1023] used for:
   <<M(MoveForwardLastSegment) and x>> A <<M(CheckMoveForwardLastSegment)>> for
   LastSegmentMoveForwardLastSegment-[x]->CheckMoveForwardLastSegment{A};
Theorem (550) [serial 1024] used for:
   <<M(CheckMoveForwardLastSegment) and x>> -> <<M(MoveForwardLastSegment)>> for
   NoMAYetCheckMoveForwardLastSegment-[x]->MoveForwardLastSegment{};
Theorem (593) [serial 1025] used for:
   <<M(CheckMoveForwardLastSegment) and x>> A <<M(GotNewMA)>> for
   GetNewMACheckMoveForwardLastSegment-[x]->GotNewMA{A};
Theorem (595) [serial 1026] used for:
   <<M(GotNewMA) and x>> -> <<M(CheckMATransition)>> for
   LastBitOfMaGotNewMA-[x]->CheckMATransition{};
Theorem (596) [serial 1027] used for:
   <<M(CheckMATransition) and x>> -> <<M(GotNewMA)>> for
   NotEndOfMACheckMATransition-[x]->GotNewMA{};
Theorem (657) [serial 1028] used for:
   <<M(CheckMATransition) and x>> A <<M(MoveForward)>> for
   StartNextMaCheckMATransition-[x]->MoveForward{A};
```

```
Theorem (106)                                [serial 1016]
P [85] << m_a'fresh >>
S [86]   << AXIOM_CMA_IS_RMA() >>
  m_a?(ma)
  ;
  << ma = CMA >>
  i := 1
  ;
  << i = 1
    and ma = CMA >>
  iSeg := ma.seg[1]
  ;
  << i = 1
    and ma = CMA
    and iSeg = CMA.seg[i]
    and AXIOM_NUM_SEG(ma : ma) >>
  nSeg := ma.seg[2]
  << i = 1
    and ma = CMA
    and iSeg = CMA.seg[i]
    and nSeg = CMA.seg[i + 1]
    and AXIOM_NUM_SEG(ma : CMA) >>
Q [57] << i < CMA.num_segments
  and iSeg = CMA.seg[i]
  and nSeg = CMA.seg[i + 1]
  and ma = CMA >>
```

```
Why created:   <<M(CheckFirstMA) and x>> A <<M(MoveForward)>> for
   GotFirstMA: CheckFirstMA-[x]->MoveForward{A};
Solved by:  Sequential Composition Rule:
   <<P1>> S1 <<Q1 and P2>>
   <<Q1 and P2>> S2 <<Q2 and P3>>
   . . .
   <<Qk-1 and Pk>> Sk <<Qk>>
   P=>P1, Qk=>Q
   _____________________________________________
   <<P>> S <<Q>>
   where S is <<P1>> S1 <<Q1>> ; . . . ; <<Pk>> Sk <<Qk>>
and theorems 30 42 48 55 78 105:
Theorem (30) [serial 1069] used for:
  <<P>> -> <<P1>> in sequential composition for [serial 1016]
Theorem (42) [serial 1070] used for:
  <<Q4>> -> <<Q>> in sequential composition for [serial 1016]
Theorem (48) [serial 1071] used for:
  <<P1>> S1 <<Q1 and P2>> in sequential composition for [serial 1016]
Theorem (55) [serial 1072] used for:
  <<Q1 and P2>> S2 <<Q2 and P3>> in sequential composition for [serial 1016]
Theorem (78) [serial 1073] used for:
  <<Q2 and P3>> S3 <<Q3 and P4>> in sequential composition for [serial 1016]
Theorem (105) [serial 1074] used for:
  <<Q3 and P4>> S4 <<Q4>> in sequential composition for [serial 1016]
```

Contention that a proof is a convincing argument for its conclusion should honestly state reasons for doubt.

We claim that a given deductive proof means that BLESS behavior meets its specification for *every* possible execution.

1. The executable code generated from the state machine may be incorrect.
2. The set of verification conditions generated for a state machine may be incorrect or incomplete.
3. The formal semantics of the BLESS language may be incorrect (or implemented incorrectly).
4. The built-in axioms may not be tautologies (or implemented incorrectly).
5. User-defined axioms (really givens) may be incorrect or inappropriate.
6. The inference rules may not be sound (or implemented incorrectly).
7. The specification of state machine behavior may be incorrect or incomplete.

Verification artifacts should be persuasive arguments understood by people.

Don't need to trust the tool; proofs should be self-evident regardless of how they were constructed.

Must be honest about what is verified, and reasons for doubt.

BLESS correctness proofs (try to) meet these criteria.