BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software[1]

Brian R. Larson, Patrice Chalin, John Hatcliff

Kansas State University

May 16, 2013

[1] Work supported in part by the US National Science Foundation (NSF) (#0932289, #1239543), the NSF US Food and Drug Administration Scholar-in-Residence Program (#1065887, #1238431) the National Institutes of Health / NIBIB Quantum Program, and the US Air Force Office of Scientific Research (AFOSR) (#FA9550-09-1-0138).

Current Systems Engineering Challenges

- involve both hardware and software (design process needing to move functionality between the two)
- bigger systems (more μP; more software)
- many teams (geographically dispersed, different organizations)
- challenges of systems integration (getting teams to agree so that the system pieces will eventually "glue together")
- benefits from multiple forms of analysis (earlier is better)

Architecture Analysis and Design Language

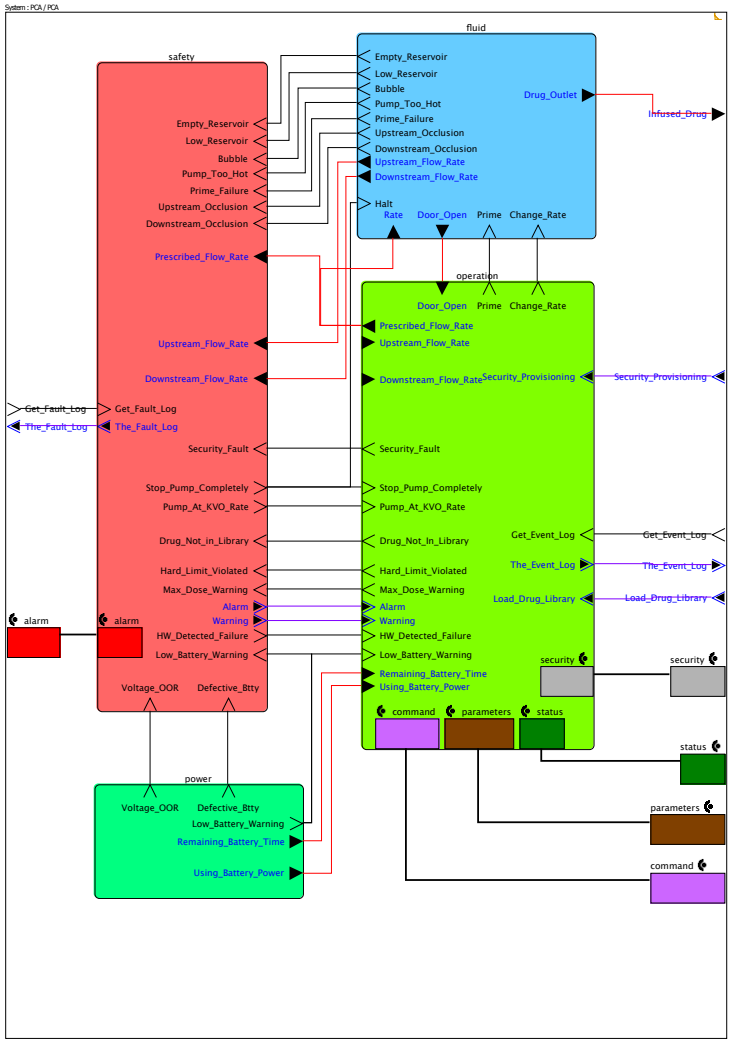
AADL is a component-oriented modeling language for embedded systems.

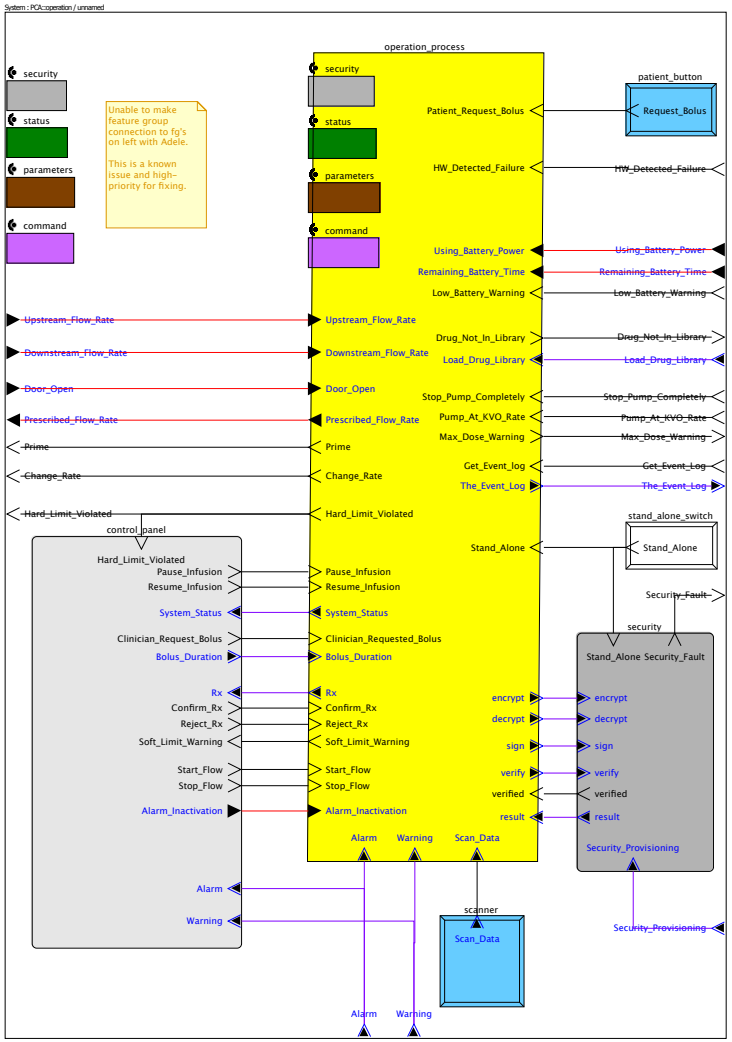
SAE International standard AS5506B (v2.1 2012) defines core language semantics rigorously, but natural language.

AADL includes constructs for both hardware (physical) and software (logical) components.

Extensible through annex sublanguages and user-defined properties.

AADL Graphical Notation





AADL Textual Notation

```
system PositionControlSystem
features
PositionSetpoint: in event data port Position;
properties
   Timing_Properties::Clock_Period_Range=>PSC::StepDuration;
end PositionControlSystem;

system implementation PositionControlSystem.common
subcomponents
  c: system Controller;  --processor, memory, process, threads
  a: system Actuator;  --motor, valve, hard-wired circuits
connections
  ps: port PositionSetpoint->c.PositionSetpoint;
  ac: subprogram access c.ActuatorCommand -> a.ActuatorCommand;
end PositionControlSystem.common;
```

AADL Tools

Open-Source AADL Tool Environment (OSATE): provides reference implementation as Eclipse plugin.[2]

AADL Inspector: stand-alone commercial tool[3]

many analysis tools available:
scheduling (Cheddar), code generation (Ocarina-RAMSES), requirements (RDALTE), mass, power, port connection consistency, bus power draw, ARINC-653 configuration, unhandled faults, fault-tree analysis, failure modes and effects analysis, functional hazard analysis, statistical model checking (PRISM), Lute

[2] Software Engineering Institute at Carnegie Mellon University
[3] Ellidiss Technologies

“Integrate Then Build"

System Architecture Virtual Integration (SAVI):

Boeing, Airbus, Lockheed Martin, BAE Systems, Rockwell Collins, GE Aviation, FAA, DoD, SEI, Honeywell, Goodrich, United Technologies, and NASA

- precise system architecture – machine-analyzable, single architectural model annotated with precise notation
- important interactions are specified and interfaces are designed, and integration verified before the internals of components are built
- produce implementations that are compliant with the architecture

Annex Sublanguages

The AADL standard defines a core language to express system partitioning and connectivity.

The core language allows extension by annex sublanguages.

```
annex MyAnnex {** . . . **}
```

Several annex sublanguages have been standardized by SAE International as annexes to the core standard.

AADL Has No Behavioral Interface Specifications

AADL emphasizes "integration" (as in the SAVI program), but current only provides structural / type-based declaration of interfaces, but no behavior properties

What is true about the component when it issues an event on a port?

What is assumed by a component when it reacts to an event?

What do emitted/received values mean?

Weak Specifications for Internal Component Behavior

AADL provides a Behavioral Annex sublanguage grammar, but no semantics for BA, much less formal semantics.

```
annex behavior_specification {**
variables
  last_beat: BLESS_Types::Time;
states
  power_on : initial state;
  pace : complete state;
  sense : complete state;
  check_pace_vrp : state;
  check_sense_vrp : state;
  off : final state;
```

```
transitions
  T1: power_on-[]->sense
    {n! & last_beat := now};
  T2: pace,sense-[on dispatch stop]->off;
  T3: pace-[on dispatch timeout (p n) l ms]->pace
    {p! & last_beat := now};
  T4: pace-[on dispatch s]->check_pace_vrp;
  T5: check_pace_vrp-[(now-last_beat) < r]->pace;
  T6: check_pace_vrp-[(now-last_beat) >= r]->sense
    {n! & last_beat := now};
  T7: sense-[on dispatch timeout (p n) l ms]->pace
    {p! & last_beat := now};
  T8: sense-[on dispatch s]->check_sense_vrp;
  T9: check_sense_vrp-[(now-last_beat) < r]->sense;
  T10: check_sense_vrp-[(now-last_beat) >= r]->sense
    {n! & last_beat := now};
**};  --end of BA for VVI
```

No Reasoning Framework

AADL emphasizes analysis, but doesn't provide a semantics nor foundational verification algorithms for reasoning about component composition nor BA to interface compliance.

AADL Needs

- formal behavior interface specification language
- formal component behavior language
- verification method that implementation meets specification
- verification tools that produce independently auditable evidence of compliance of behaviors to specs

BLESS is Annex Sublanguage of AADL

BLESS programs are attached to system architecture to define component behavior.

SAE International standard AS5506B defines the Architecture Analysis and Design Language (AADL). Discovered in 2007, AADL replaced crude structural constructs of DAREN.

BLESS is pure behavior; AADL is pure structure.

BLESS is Programming Language to Control Machines

Behavior Language for Embedded Systems with Software (BLESS) mathematically defines embedded programs, their specifications, and their executions from first principles

- BLESS assertions provide formal behavior interface specification language
- BLESS annex subclauses provide formal component behavior language
- BLESS proof tool enables verification method that implementation meets specification that produces independently auditable evidence of compliance of behaviors to specs

BLESS Proves Component Behavior Correctness

Formally prove that every execution upholds its specification by:

- Write BLESS contracts for AADL component interfaces
- Write BLESS internal component behaviors
- Annotate programs with BLESS assertions forming proof outlines.
- Use proof tool to transform outlines into proofs.

BLESS akin BA

Behavior specification annex sublanguage standardized as annex document of AS5506 ; known as “BA"

BA inspired BLESS; coordinated grammars during standardization process. Like BA, BLESS behaviors are state-transition systems augmented with simple temporal logic formulas.

assert
  assertion declarations
invariant
  invariant assertion
variables
  variable declarations
states
  state declarations
transitions
  source-[condition]->destination {action};

BLESS Assertions

Proof outlines are Assertions[4] attached to states, and inserted before and after actions.

Assertions are bounded, first-order predicates augmented with simple temporal operators: `@ ^ '`

Assertions delimited by double angle brackets: `<< >>`

```
<<VS: : s@now and notVRP()>>
```

[4] Capital 'A' for temporal logic formulas used for BLESS

Verification Conditions

Verification conditions are Hoare triples:

{P} S {Q} ≡ `<<P>>S<<Q>>`

where P and Q are Assertions and S is an action.

BLESS Proof Tool Makes Proofs from Outlines

The BLESS proof tool transforms programs having proof outlines into a complete, formal proof[5] semi-automatically.

[5] Proofs are sequences of theorems, each of which is given, axiomatic, or derived from earlier theorems by sound inference rules. No sequence of theorems–no proof.

BLESS Proof Tool is Proof Checker

The BLESS proof tool applies human-selected tactics.

All information needed for proof must appear in BLESS program source text.

The BLESS proof tool is a verification condition generator + proof checker–not a theorem prover.

Resulting correctness proof created as witness during program proof checking.

Generate VCs, Pound Into Normal Form

The BLESS proof tool

- generates verification conditions from BLESS program text
- reduces compound actions to atomic actions
- transforms atomic actions into implications
- pounds implications into axiomatic normal form

Human directed tactics selected from GUI, or read from script, applied to each unsolved proof obligation in current pool.

BLESS Assertions

BLESS Assertions[6] are first-order predicates enclosed in `<<>>` with a simple temporal operator.

`p@t` means predicate `p` evaluated at real-valued time `t`.

Assertions may be attached as `BLESS::Assertion` properties of ports, or appear within BLESS annex subclauses.

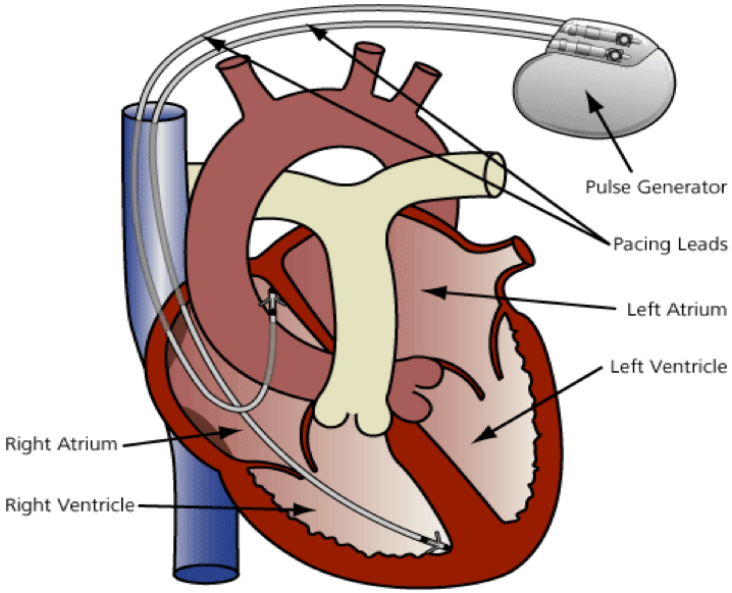
`p^k` means predicate `p` evaluated at `k` periods from `now`.

`p'` is shorthand for the value of `p` one period hence: `p'≡p^1`

[6] capital 'A' is used as a proper noun for BLESS Asserions

VVI is ‘Hello World!’

VVI-mode cardiac pacing is ‘Hello World!’ example of single-component behavior.



Composition of proved-correct AADL components into proved-correct systems will be the subject of future papers and presentations.

VVI-Mode Pacemaker

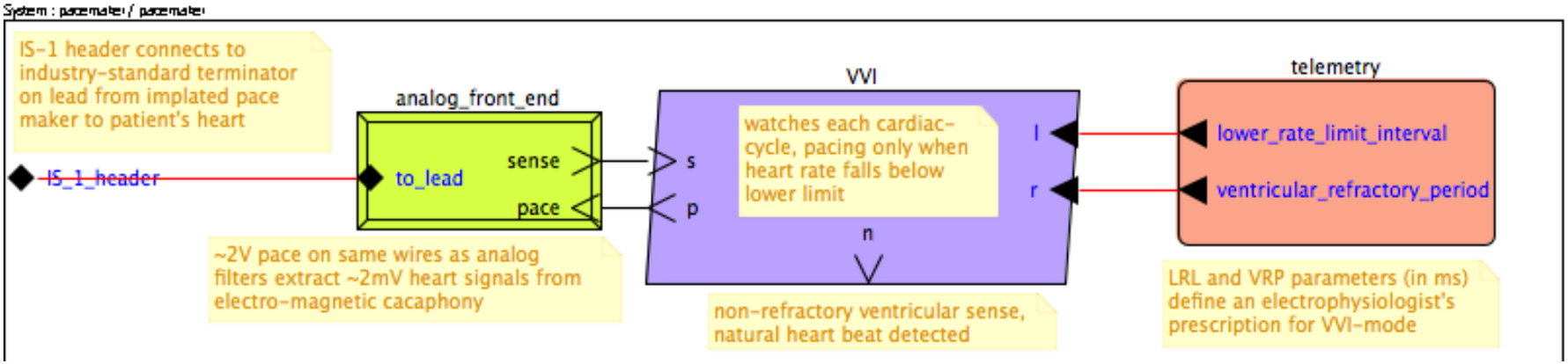
“VVI" is a cardiac pacing mode that lets a patient’s heart beat on its own above a prescribed rate, but take over to emit a short current to cause contraction when the patient’s intrinsic rate fell below the prescribed rate.[7]

The first “V" of “VVI" says pace ventricle (right-ventricle unless otherwise indicated), the second “V" says sense ventricle, and the “I" says to inhibit pacing when sensed beats are sufficiently fast.

The lower rate limit (LRL) is the heart rate, prescribed by the physician in beats per minute at which the pacemaker will not let the heart beat more slowly. In practice, the lower rate limit is less thought of by its rate in beats-per-minute, but by its duration in milliseconds.

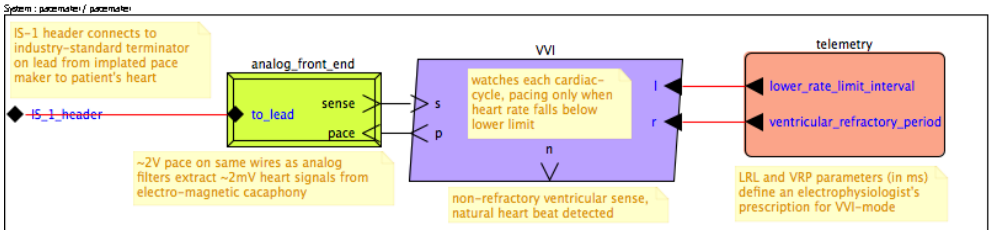
[7] *PACEMAKER System Specification*, Boston Scientific, 2007.

VVI.aadl Component



VVI.aadl Component

```
thread VVI
  features
    s: in event port; --ventricular contraction has been sensed
    p: out event port --pace ventricle
      {BLESS::Assertion=>"<<VP()>>";};
    n: out event port --non-refractory ventricular sense
      {BLESS::Assertion=>"<<VS()>>";};
    l: in data port T;  --lower rate limit interval
    r: in data port T;  --ventricular refractory period
  properties
    Dispatch_Protocol => Aperiodic;
annex BLESS {** . . . **};
end VVI;
```



Effectiveness Property

The invariant that keeps the patient lively is:

> *"There will always be a pace or a (non-refractory) sense in the previous lower-rate limit interval."*

Long pauses between heartbeats must not occur. Cardiologists choose a lower-rate limit (LRL) maintained by the pacemaker, on demand, when the patient's intrinsic rate would be too slow.

A typical LRL of 60 beats-per-minute (bpm) has an LRL interval of 1000 ms.

Real hearts are electrically-noisy after contraction. Therefore, during ventricular refractory period (VRP) following a sense or pace, electrical signals should be ignored.

Thread Invariant

Thread behavior is *specified* by its thread invariant, much like a loop invariant, and its **BLESS::Assertion** properties of ports.

The current instant is **now**.

```
invariant
 <<LRL(now)>>  --LRL is true, whenever "now" is
```

Assertion LRL

Assertion `LRL` takes a parameter `x`.

The invariant says `LRL(now)` will be true, whenever `now` happens to be.

```
<<LRL:x: --Lower Rate Limit
  exists t:T  --there was a moment
    in x-l..x   --within the previous LRL interval
    that (n@t or p@t) >>  --with a pace or non-refractory sense
```

(there is a time, t in the lower-rate limit interval before time x in which either a ventricular-pace, or non-refractory ventricular-sense event occurred.)

Ventricular Refractory Period (VRP)

After contraction, hearts have electrical noise that should be ignored. The ventricular refractory period (VRP) determines the period of unresponsiveness. `notVRP` becomes true after VRP has expired.

```
<<notVRP: : --Ventricular Refractory Period
  (n or p)@last_beat --last beat before now,
  and (now-last_beat)>=r>>  --older than VRP
```

Port Assertions

Assertion properties of out event ports specify what must be true when an event is sent by the port.

```
<<VS: : --ventricular sense detected, not in VRP
  s@now and notVRP() >>
<<VP: : --cause ventricular pace
  (n or p)@(now-l)  --last beat occurred LRL interval ago,
  and --not since then
    not (exists t:T  --there is no time
      in now-l,,now  --since then, ",," means open interval
      that (n or p)@t) >>  --with a pace or sense
```

States

Thread states may be

- **initial** starting state, must have exactly one
- **final** ending state, no outgoing transitions
- **complete** suspend until next dispatch upon entering
- **execute** transitory states

States may have Assertions that specify what is true when the thread is in a state.

States

```
states
  power_on : initial state  --powered-up,
    <<VS()>>; --start with "sense"
  pace : complete state
      --a ventricular pace has occured in the
      --previous LRL-interval milliseconds
    <<PACE(now)>>;
  check_pace_vrp : state
      --execute state to check if s sooner than VRP after pace
    <<s@now and PACE(now)>>;
  sense : complete state
      --a ventricular sense has occured in the
      --previous LRL-interval milliseconds
    <<SENSE(now)>>;
  check_sense_vrp : state
      --execute state to check if s sooner than VRP after sense
    <<s@now and SENSE(now)>>;
  off : final state;  --upon "stop"
```

State Assertions

```
<<PACE:x: --pace occurred in the previous LRL interval
  p@last_beat and  --previous beat was a pace
  (exists t:T  --there is a time
    in x-l..x  --in the previous LRL interval
    that p@t) >>  --with a ventricular pace
<<SENSE:x:  --sense occurred in the previous LRL interval
  n@last_beat and  --previous beat was a sense
  (exists t:T  --there is a time
    in x-l..x  --in the previous LRL interval
    that n@t) >>  --with a non-refractory sense
```

Initial and Stop Transitions

Transitions have one or more source states, transition condition, destination state, and possibly an action.

```
transitions
  T1_POWER_ON: --initialization
  power_on -[ ]-> sense
    {<<VS()>>
    n!<<n@now>>   --first "sense" at initialization
    & last_beat:=now<<last_beat=now>>};

  T2_STOP: --turn off pacing
  pace,sense -[on dispatch stop]-> off{};
```

Transitions After Pace

```
T3_PACE_LRL_AFTER_VP: --pace when LRL times out
pace -[on dispatch timeout (p n) l ms]-> pace
  { <<VP()>>
  p!<<p@now>>   --cause pace when LRL times out
  & last_beat:=now <<last_beat=now>>};

T4_VS_AFTER_VP: --sense after pace=>check if in VRP
pace -[on dispatch s]-> check_pace_vrp{};
```

Check if in VRP

```
T5_VS_AFTER_VP_IN_VRP:  -- s in VRP,  go back to "pace" state
check_pace_vrp -[(now-last_beat)<r]-> pace{};

T6_VS_AFTER_VP_IS_NR: --s after VRP,
  --go to "sense" state, send n!, reset timeouts
check_pace_vrp -[(now-last_beat)>=r]-> sense
  {  <<VS()>>
  n!<<n@now>>  --send n! to reset timeouts
  &last_beat:=now <<last_beat=now>>};
```

Verification Conditions

Subprogram behaviors have one verification condition.

Thread behaviors have a verification condition for each state and transition.

VVI.aadl requires 15 verification conditions.

Complete State Proof Obligations

The Assertions of complete states must imply the thread invariant.

```
P [64] <<PACE(now)>>
S [51] ->
Q [51] <<LRL(now)>>
  What for: <<M(pace)>> -> <<I>> from invariant I
    when complete state pace has Assertion
    <<M(pace)>> in its definition.
```

Execute State Proof Obligations

The execute states, `check_pace_vrp` and `check_sense_vrp`, must have an enabled, outgoing transition:

```
P [71] <<s@now and PACE(now)>>
S [71]->
Q [71] <<((now-last_beat) < r) or ((now-last_beat) >= r)>>
  What for: Serban's Theorem:  disjunction of execute conditions
  leaving execution state check_pace_vrp,
  <<M(check_pace_vrp)>> -> <<e1 or e2 or . . . en>>
```

Initial Transition Proof Obligation

For transition `T1_POWER_ON` from the `power_on` initial state:

```
P [60] <<VS()>>
S [82]<<VS()>>
n!
<<n@now>>
&
last_beat := now
<<last_beat = now>>
Q [68] <<SENSE(now)>>
  What for:  <<M(power_on)>> A <<M(sense)>> for
    T1_POWER_ON:power_on-[ ]->sense{A};
```

Proof of VVI.aadl

Though rather long, inspecting the generated proof is the means to convince oneself that all of the obligations have indeed been proved.

The proof of VVI.aadl requires 123 theorems, that last of which says all verification conditions have proofs.

pace upholds invariant

The first three theorems prove that the Assertion of complete state pace upholds the thread invariant.

```
Theorem (1)                                    [serial 1155]
 76 {P} <<(exists t:Timing_Properties::Time
    in now-PP::lower_rate_limit_interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
 64 S  =>
 64 {Q} <<(exists t:Timing_Properties::Time
  in now-PP::lower_rate_limit_interval..now
  that nr_vs@t )
or (exists t:Timing_Properties::Time
  in now-PP::lower_rate_limit_interval..now
  that vp@t )>>
by And-Elimination/Or-Introduction Schema: (P and Q)=>(P or R)
```

```
Theorem (2)                                    [serial 1129]
 76 {P} <<(exists t:Timing_Properties::Time
    in now-PP::lower_rate_limit_interval..now
    that vp@t )
  and
  vp@last_vp_or_vs>>
 64 S  =>
 64 {Q} <<exists t:Timing_Properties::Time
  in now-PP::lower_rate_limit_interval..now
  that (nr_vs@t or vp@t) >>
by Distribution of preconditions, and over or, and distribution of
and theorem 1.
```

```
Theorem (3)                                        [serial 1002]
 76 {P} <<PACE(now)>>
 64 S   =>
 64 {Q} <<LRL(now)>>
by Substitution of Assertion Labels
and theorem 2:
```

BLESS is

- an AADL annex sublanguage defining behavior of components
- a simple temporal logic to specify behavior (port Assertions and component invariant)
- a plug-in to OSATE which is a plug-in to Eclipse, which
- generates verification conditions
- transforms proof outlines into complete proofs (with human guidance)

Future Work

- coupling of BLESS behavior with EMV2 error models (SEI)
- prove component invariants from proofs of subcomponents
- simulation
- code generation
- proof obligation export to other tools
- use “smarter" inference engine to generate intermediate assertions in proof outline
- submit BLESS to SAE International to become annex standard of AADL