BLESS: Formal Specification and Verification of Behaviors for Embedded Systems with Software*

Brian R. Larson, Patrice Chalin, and John Hatcliff

Kansas State University, Kansas, USA
({brl,chalin,hatcliff}@ksu.edu)

Abstract. Recent experience in the avionics sector has demonstrated the benefits of using rigorous system architectural models, such as those supported by the standard Architectural and Analysis Definition Language (AADL), to ensure that multi-organization composition and integration tasks are successful. Despite its ability to capture interface signatures and system properties, such as scheduling periods and communication latencies as model attributes, AADL lacks a formal interface specification language, a formal semantics for component behavioral descriptions, and tools for reasoning about the compliance of behaviors to interface contracts. In this paper we introduce the Behavioral Language for Embedded Systems with Software (BLESS)—a behavioral interface specification language and proof environment for AADL. BLESS enables engineers to specify contracts on AADL components that capture both functional and timing properties. BLESS provides a formal semantics for AADL behavioral descriptions and automatic generation of verification conditions that, when proven by the BLESS proof tool, establish that behavioral descriptions conform to AADL contracts. We report on the application of BLESS to a collection of embedded system examples, including definition of multiple modes of a pacemaker.

1 Introduction

Recent experiences in the avionics sector have demonstrated the benefits of using rigorous system architectural models, such as those supported by the SAE standard Architectural and Analysis Definition Language (AADL) [19], to ensure that multi-organization integration tasks are successful. For example, on the System Architecture Virtual Integration (SAVI) effort, members of the Avionics Vehicle Systems Institute, including Boeing, AirBus, Honeywell, and Rockwell Collins, conducted pilot studies in the use of AADL to define precise system architectures [23]. Using an "integrate then build" design approach, important interactions are specified, interfaces are designed, and compatibility of modules and crucial system properties are verified before the internals of components are built. Subsequently, stakeholders provide implementations that are compliant with the architecture [9]. This development approach focuses on defining

* 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). The authors with to thank engineers from the US Food and Drug Administration for feedback on this work.

precise interface descriptions and exposing in architectural models important properties needed to perform component-wise and system-wide analysis of real-time scheduling and error propagation properties.

Despite its ability to capture interface signatures and system properties—such as scheduling periods and communication latencies—as model attributes, AADL lacks (i) a formal behavioral interface specification language, (ii) a formal semantics for component behavioral descriptions, and (iii) tools for reasoning about the compliance of behaviors to interface specifications. Obviously, such capabilities are needed to fully support the "integrate then build" vision of SAVI, as well as the full potential of AADL in other contexts.

Previous work on Behavioral Interface Specification Languages (BISLs) [11] has produced a number of specification and verification technologies for programming languages used in system development, and has demonstrated that these techniques can support the type of formal compositional reasoning that would greatly benefit the safety-critical architecture-centric development supported by AADL and illustrated by, *e.g.*, the SAVI project. While several technical concepts and lessons learned from previous work on BISLs can be carried over and applied to the design of a BISL for AADL, there are a number of interesting differences that give rise to significant challenges not considered in BISLs such as JML [8] and Spec# [5]. For instance, while conventional programming languages focus on interactions via method calls and shared variable concurrency, AADL emphasizes component-based designs with synchronous and asynchronous communication via ports. Thus, a different approach is needed for positioning contracts in source artifacts and for generating verification conditions to support compositional reasoning in the presence of buffered and unbuffered port behavior. A behavioral specification framework for AADL must be carefully designed to align with the real-time operating system (RTOS) concepts defined in the standardized AADL runtime environment, which supports systems targeted for deployment on real-time platforms like the ARINC 653, a platform for modular avionics [12]. A behavioral specification framework for AADL must also provide a means of specifying and reasoning about crucial timing properties phrased in terms of the RTOS concepts of scheduling periods exposed as architecture attributes . Finally, since AADL is used in domains that require certification, it is desirable for tools that reason about envisioned AADL specifications produce auditable artifacts that can be assessed as part of the certification process.

In this paper, we introduce the *Behavioral Language for Embedded Systems with Software* (BLESS)—a BISL and an associated proof environment for AADL [13]. The AADL standard provides the notion of an "AADL Annex" to support extensions to the modeling language, and BLESS uses this mechanism to introduce notations for (a) specifying behaviors on component interfaces, (b) defining AADL-runtime aware transition systems that capture the internal behavior of AADL components, and (c) writing assertions to capture important state and event properties within the transition system notation. BLESS annex subclauses can be inserted into AADL components transparently to other uses of the system architecture. Successful definition and tool engineering for formal reasoning frameworks like BLESS require a precisely defined formal semantics, and we have invested considerable effort in defining a such a semantics for

BLESS[1]. Finally, BLESS includes a verification-condition (VC) generation framework and an accompanying proof tool that enables engineers to prove VCs via proof scripts build from system axioms and rules from a user-customizable rule library.

BLESS and AADL are rich, expressive languages. Due to space constraints, this paper gives a cursory introduction to AADL (Section 2), and then focuses on the *core features* of the BLESS language (Sections 3 & 4) and its associated specification and verification methodology which is assisted by the BLESS tool (Section 5). Material is presented using a running example from the medical device domain: a pacemaker [21].

The BLESS tool framework [24] is implemented as a publicly available open source plug-in to the Eclipse-based OSATE environment for AADL [18], and includes and editor for BLESS specifications and an environment operating the BLESS proof engine.

2 Background and Motivation

Avionics & origins of AADL: To manage the ever increasing complexity of electronic control systems, engineers devised ways to recursively partition complex systems into collections of simpler sub-systems. As a result, several *architectural* domain specific languages (DSLs) were invented[2] to capture system structure and, more importantly, the specification of interfaces—which are crucial to successful system composition.

In a movement to standardize an architectural description language for avionics and aerospace systems, SAE International sponsored standard subcommittee AS-2C; it is from this effort that SAE AS5506 *Architecture Analysis & Design Language* (AADL) was created, and revised upon use, now AS5506C. Both commercial and open source tools for creating and analyzing AADL models are available [18].

AADL Core and Annex Sublanguages: AADL was designed with extensibility in mind. It has a core language defined in its own standards document. The core language allows one to express architectural structure—using components, interfaces, connections, containment—but not behavior. The core language is extensible with *annex sublanguages*, some of which have been standardized by an annex standard document approved by AS-2C. The AADL (core) grammar allows insertion, into the text representing components, an annex subclause of the form: `annex AnnexName {** ... **}`. Key standardized annexes include [20]:

- Behavior Annex (BA) extends AADL with the ability of defining component behavior via state machines having: states, state variables and (guarded) state transitions with associated actions written in a simple imperative language.
- Data Modeling Annex (DM) supports the definition of data components.
- ARINC653 Annex defines properties relevant to the elaboration of ARINC653 compliant embedded systems.

BLESS, inspired from BA, improves and extends the state-transition formalism and, more importantly, introduces the notion of *Assertion* as a basic building block for con-

[1] Due to space constraints, the semantics cannot be presented here. We refer interested readers to [13] for the formal semantics and details of the VC generation process.

[2] The section on related work describes some of these DSL.

tractual specifications. BLESS also adds contracts to AADL subprograms, though this feature is not covered here.

Upon reviewing an early draft of the proposed standard annex document, a committee member pondered whether a tool made to transform proof outlines of highly-concurrent programs could be adapted for a suitably-annotated state-transition system. Motivated by this possibility, the first author created BLESS: an AADL annex sublanguage that can be used to annotate BA behavior with Assertions[3].

3 BLESS by Example: A Simple Pacemaker

3.1 Cardiac Pacing

Cardiac pacing is used for a compact illustration of BLESS, because it is a simple safety-critical cyber-physical systems having crucial timing properties. Figure 1 depicts a pacemaker (Pulse Generator) connected by pacing leads to the inside of a heart's right atrium and ventricle.

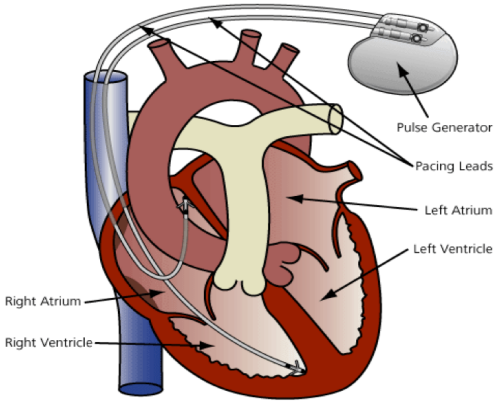

Fig. 1. Pacemaker Environment

The first pacemakers emitted a short (<1 ms), low voltage (1V to 10V) "pace" at a rate fixed (60-80 bpm) in the factory through a single lead to the right ventricle. A pace causes a cascade of cell contractions in both right and left ventricles, expelling blood to lungs and body, respectively. The earliest electrophysiologists selected the pacing rate and voltage when ordering a device. When everything worked, people whose bradycardia[4] made simple walking activities tiresome, felt "normal" after the implant, instead of incessantly-increasing fatigue until succumbing to their heart disease.

However, constant-rate electrical-pacing interfered intrinsic pacing of a patient's heart. Fortunately, the same leads implanted to deliver electrical paces, can also sense intrinsic electrical activity of the heart (approx. 1mV). These signals, sensed within the heart, are called *electrograms*.

Analog design wizards devised ways to amplify and filter millivolt electrograms, and then to compare the signal with a programmable threshold to determine whether an intrinsic contraction occurred (Figure 2). This allowed the pacemaker to monitor the patient and inhibit electrical-pacing when intrinsic pacing voltage exceeds a lower limit. This inhibitory behavior became known as "VVI", short for: Ventricle (chamber(s) paced), Ventricle (chamber(s) sensed), Inhibit [5].

Complicating matters, real hearts are electrically noisy during and after contraction, so the pacemaker must ignore any sensed voltage for a period of time after either pacing or sensing heart contraction. By adjusting this *ventricular refractory period* (VRP) and

[3] Capital 'A' as the proper-noun for assertions defined by BLESS.
[4] Bradycardia is the class of cardiac diseases in which the heart beats too slowly.
[5] Constant-rate pacing is referred to as VOO, for "pace ventricle, no sensing, constant rate".

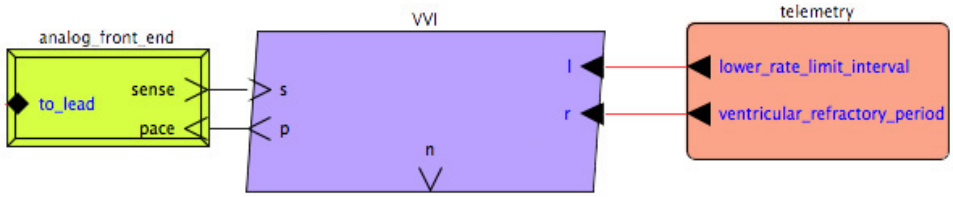

Fig. 2. Pacemaker System Diagram in AADL Graphical Notation

```
thread VVI
  features
    s: in event port; -- sense of a ventricular contraction
    p: out event port;  -- pace ventricle
    n: out event port;  -- non-refractory ventricular sense
    lrl: in data port BLESS_Types::Time;  -- lower rate limit interval
    vrp: in data port BLESS_Types::Time;  -- ventricular refractory period
  properties
    Dispatch_Protocol => Aperiodic;
end VVI;
```

Fig. 3. AADL VVI Thread Component Type

sensing threshold, an electrophysiologist can tune a pacemaker for a specific patient, and by choosing a *lower rate limit* (LRL), ensure minimum heart rate.

The first author worked for five years at a leading pacemaker company, and was responsible for the release of a requirements document for a previous generation pacemaker that since been used in over 25 research papers within the formal methods community [21]. The pacing examples that we discuss in this paper adhere to those requirements.

3.2 AADL VVI Component

Presented using the standardized graphical view of AADL models, Figure 2 shows the top-level architecture of a pacing system where the structure has been simplified to focus tightly on the elements needed to present a description of the VVI pacing mode. Figure 3 shows the AADL component type that describes the interface of a thread that performs the VVI control function. The interface has three asynchronous *event* ports `s`, `p` and `n`. Events arrive at port `s` (ventricular-sense) whenever the analog front end detects electrograms exceeding the prescribed threshold. Events sent out on port `p` (ventricular-pace) cause the front end to administer a pace of prescribed voltage. An event sent out on port `n` (non-refractory ventricular-sense) indicates that a ventricular-sense was detected after expiration of the ventricular refractory period (VRP). Data ports support the automatic propagation of state values from one component to another, without triggering a thread dispatch within the receiving component. Data ports `lrl` and `vrp` communicate the lower rate limit interval and ventricular refractory period programmed by the physician through the telemetry subsystem.

3.3 Behavior for VVI

Figure 4 shows a BLESS state machine (without BLESS Assertions) that captures the behavior of the VVI component. Without Assertions, BLESS annex language sub-

clauses deliberately appear similar to BA subclauses . BLESS annex subclauses have sections for **variables**, **states**, and **transitions**.

Variable values persist from thread suspension to next dispatch . States must be one of **initial**, **complete**, **final**, or *execute* if none of these. Behavior begins in the **initial** state, terminates in a **final**, suspends upon entry to **complete** states until next dispatch. State transitions (in BA and BLESS) are of the form:

transition_name: *source_state*(*s*) -[*condition*]-> *destination_state* {*action*};
Transitions are named (*e.g.*, T2[6]), and each transition includes one or more source states (*e.g.*, the pace and sense states of T2), a single target state (*e.g.*, the off state of T2), guard expression (*e.g.*, the **on dispatch** s of T4 which holds when an event arrives on the s event port), and a (possibly empty) set of actions to execute (*e.g.*, p! & last_beat := **now** of T3). Transitions must be written to guarantee that only a finite number of execute states are passed through before entering a complete or final state. Transitions leaving **complete** states have dispatch conditions evaluated by the run time system . The special dispatch condition **on dispatch stop** holds when an event is received on a **stop** (which is predeclared for every AADL component to cause normal termination). Transitions leaving execute states may have a boolean expression as transition condition, so long as there is at least one enabled transition and a complete state will eventually be reached. Sending an event out on a port p is written p!. The current time is written as **now**. Action sets separated by "**&**" may be executed in any order, or concurrently; action sequences separated by ";" must be executed in order; assignment uses ":=". A **timeout** with a port list and a duration is reset by event arrival or departure at a listed port, and expires when the most recent event on a listed port occurred the duration previously.

The transition system captures the behavior of VVI as follows. The time of the most recent cardiac event is retained in the persistent variable last_beat. After causing a ventricular pace to begin operation [T1], VVI waits for either a ventricular sense [T4] (which it then checks for VRP [T5,T6]) or a timeout of the patient's lower-rate-limit interval since the most-recent cardiac event [T3]. If the time since the most recent cardiac event to the current ventricular sense is within the VRP, then return to the pace state and wait some more [T5]. Otherwise, reset last_beat to "now", and send an event out on n to reset the timeout [T6]. When no cardiac events have occurred in the previous lower-rate-limit interval (timeout), send an event out on p to cause an electrical-pace, and reset the timeout [T7]. The behavior from the sense state is similar [T8,T9,T10].

3.4 Adding BLESS Assertions to VVI

BLESS introduces Assertions to AADL as a basic block to forming rich behavioral specifications. Syntactically, Assertions are always enclosed in double angle brackets, << >>. Semantically, an Assertion is a temporal logic formula extending a first-order predicate calculus with simple temporal operators for continuous p@t and discrete time p^k, where t is the time at which p occurs, and k is the number of dispatch periods from now. As indicated by operators such as p^k, BLESS Assertions are able to reason

[6] We will sometimes abbreviate transition names, using only the "T#" prefix.

```
annex BLESS {**
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:
    power_on -[ ]-> pace  {p! & last_beat := now};
  T2_STOP:
    pace,sense -[on dispatch stop]-> off {};
  T3_PACE_LRL_AFTER_VP:
    pace -[on dispatch timeout (p n) lrl ms]-> pace
      {p! & last_beat := now};
  T4_VS_AFTER_VP:
    pace -[on dispatch s]-> check_pace_vrp {};
  T5_VS_AFTER_VP_IN_VRP:
    check_pace_vrp -[(now-last_beat) < vrp]-> pace {};
  T6_VS_AFTER_VP_IS_NR:
    check_pace_vrp -[(now-last_beat) >= vrp]-> sense
      {n! & last_beat := now};
  T7_PACE_LRL_AFTER_VS:
    sense -[on dispatch timeout (p n) lrl ms]-> pace
      {p! & last_beat := now};
  T8_VS_AFTER_VS:
    sense -[on dispatch s]-> check_sense_vrp {};
  T9_VS_AFTER_VS_IN_VRP:
    check_sense_vrp -[(now-last_beat) < vrp]-> sense {};
  T10_VS_AFTER_VS_IS_NR:
    check_sense_vrp -[(now-last_beat) >= vrp]-> sense
      {n! & last_beat := now};
**};
```

Fig. 4. VVI Thread Behavior (without BLESS assertions)

directly in terms of the logical scheduling periods captured in the AADL run-time environment – which forms an abstraction of common threading and scheduling services, *etc.*, found in widely-used real-time operating systems. Thus, in contrast to other modeling languages and model checking frameworks that assume arbitrary interleaving of concurrent transitions, BLESS assumes an interleaving semantics corresponding to the scheduling abstractions in the standardized AADL run-time environment.

BLESS supports the following kinds of Assertion:

- *Port Assertion* expresses what is true when an event is sent or received on a port.
- *State Assertion* expresses what is true while the machine is in that state.
- *Action Step Assertion* expresses what is true during the execution of a transition's action steps, at that point where the Assertion is inserted.
- *Thread Invariant* expresses a predicate that must be true of every state.

As will be illustrated next, VVI relevant examples of each kind of Assertion are given in Figure 5.

Before doing so we will review BLESS's support for "predicate abstraction" allowing (parameterized) predicates to be named and subsequently used in Assertions. As an example, refer to the `LRL` predicate defined on line 14 in Figure 5. It captures the property that the patient had an intrinsic or electrically-paced heartbeat in the previous lower-rate-limit interval (cf. `lrl` defined in Figure 3). The identifier `x` is a formal parameter representing a particular time. The predicate can be read as follows: there exists a time `t` in the closed interval $[x - lrl, x]$ such that an event was issued on port `n` or port `p` at time `t`.

```
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()>>";};
    lrl: in data port T;  -- lower rate limit interval
    vrp: in data port T;  -- ventricular refractory period
  properties
    Dispatch_Protocol => Aperiodic;
annex BLESS {**
assert
  <<LRL:x: exists t:T in x-lrl..x that (n@t or p@t)>> -- Lower Rate Limit
  <<VS: : s@now and notVRP()>> -- ventricular sense detected, not in VRP
  <<VP: : -- ventricular pace
    (n or p)@(now-lrl)  --last beat occurred LRL interval ago,
    and -- not since then (",," means open interval)
      not (exists t:T in now-lrl,,now that (n or p)@t) >>
  ... -- Not shown are notVRP(), PACE(t), SENSE(t), etc.
invariant
  <<LRL(now)>>  -- LRL is "always" true
variables
  last_beat : T
    <<LAST: : (n or p)@last_beat>>; -- time of last pace or NR sense
states
  power_on : initial state <<VS()>>; --start with "sense"
  pace : complete state
    -- ventricular pace occurred in previous LRL interval
    <<PACE(now)>>;
  sense : complete state
    -- ventricular sense occurred in previous LRL interval
    <<SENSE(now)>>;
  check_pace_vrp : state
    -- execute state to check if s is in vrp after pace
    <<s@now and PACE(now)>>;
  check_sense_vrp : state
    -- execute state to check if s is in vrp after sense
    <<s@now and SENSE(now)>>;
  off : final state;
transitions
  T1_POWER_ON:  power_on -[ ]-> sense
    {<<VS()>> n! <<n@now>> & last_beat := now <<last_beat=now>>};
  ...
  T6_VS_AFTER_VP_IS_NR: check_pace_vrp -[(now-last_beat)>=r]-> sense
  -- s after VRP, go to "sense" state, send n!, reset timeouts
    {<<VS()>> n! <<n@now>> & last_beat := now <<last_beat=now>>};
... **}; ...
```

Fig. 5. BLESS-annotated VVI Thread Component Type

Figure 5 also illustrates how a `BLESS::Assertion` can be attached to a port (cf. lines 5 and 7). By attaching such Assertions one is specifying that the given predicate will be true whenever an event is issued over the port. The VVI thread invariant (`LRL(now)`) is given on line 22; it is effectively stating that: from when the thread leaves its initial state until it enters a final state, the patient has a heart beat in the previous lower rate limit interval. Finally, an example of a state Assertion is given on line 30 for the `pace` state; and inlined action step Assertions are given on, *e.g.*, line 43.

The complete BLESS source for the VVI example is available in [14]. (As a convenience of reviewers, we have included the complete BLESS source for VVI.aadl in an appendix.)

4 Thread Verification Obligations

For a subprogram annotated with a contract, its correctness argument (which we will call a proof obligation) takes the form of: under the assumption that the precondition holds, if the body is executed (and terminates), then the postcondition must hold. A thread has a proof obligation for each complete or execute state, and each transition.

In the subsections that follow, we illustrate each of these kinds of proof obligation. As we shall see, all proof obligations have the form `<<`*P*`>>`*S*`<<`*Q*`>>` where *P* and *Q* are Assertions, and *S* is an action (possibly `Skip`, the empty action).

Complete State Assertions Imply Invariant: Entering a complete state suspends the thread until next dispatch. Therefore each complete state's Assertion must imply the thread's invariant. Thus, *e.g.*, the complete state `pace` has Assertion `<<PACE(now)>>`. Its proof obligation, as generated by the BLESS proof tool, is:[7]

```
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.
```

The "What for" part explains why the proof tool generated the proof obligation; `M(...)` represents a meaning function. Thus, the "meaning" of the `pace` state is the pace state Assertion.

Execute States Have Enabled Outgoing Transition: Execute states are transitory and so they must always have an enabled, outgoing transition. If more than one transition is enabled, the choice is nondeterministic. Each execute state's Assertion must imply the disjunction of outgoing transition conditions[8]. Proof obligation for execute states `check_pace_vrp` is:

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

Execute Transitions Without Actions: For transitions without actions, the conjunction of the Assertion of the source state and the transition condition must imply the Assertion of the destination state. Execute transitions have Boolean-valued expressions for transition conditions. Proof obligation for execute transition `T5` is:

```
P [71] <<s@now and PACE(now) and ((now-last_beat) < vrp)>>
S [97]->
Q [64] <<PACE(now)>>
   What for:  <<M(check_pace_vrp) and x>> -> <<M(pace)>> for
   T5_VS_AFTER_VP_IN_VRP:check_pace_vrp-[x]->pace{};
```

The transition condition `T5` occurs when a sense was in VRP, thus ignored. As was mentioned earlier, `M(check_pace_vrp)` stands for the meaning of `check_pace_vrp`, which is its Assertion: `<<s@now and PACE(now)>>`. Since the `T5` transition condition is `(now-last_beat) < vrp`, their conjunction is the predicate given as `P` in the proof obligation. The Assertion for state `pace`, `<<M(pace)>>`, is `<<PACE(now)>>`, shown as `Q`.

Execute Transitions With Actions: For execute transitions with actions, the conjunction of the Assertion of the source state and the transition condition becomes the pre-

[7] Numbers in square brackets correspond to line #s in the full VVI source given in Section **??**.
[8] Named for Serban Georghe.

condition; and the Assertion of the destination state is the postcondition. The proof obligation for execute transition with actions `T6` is:

```
P [71] <<s@now and PACE(now) and ((now-last_beat) >= r)>>
S [104]<<VS()>>n!<<n@now>> & last_beat := now<<last_beat=now>>
Q [68] <<SENSE(now)>>
  What for:  <<M(check_pace_vrp) and x>> A <<M(sense)>> for
  T6_VS_AFTER_VP_IS_NR:check_pace_vrp-[x]->sense{A};
```

Note that it is similar for `T10`. For both `T6` and `T10`, the sense was after expiration of the ventricular refractory period, with the same action `A`: an event sent out port `n`, and `last_beat` set to the current time.

Initial and Stop Transitions: Transitions leaving `initial` states have proof obligations like execute transitions, with or without actions.

Every AADL component has an implicit `stop` port meant to signal orderly termination and transition to a `final` state. Because `final` states don't do anything, stop transitions without actions like `T2` generate trivial proof obligations logically equivalent to *true*.

Dispatch Transitions With Timeout: Transitions leaving `complete` states must have dispatch conditions evaluated by the runtime system . BLESS follows BA in requiring that dispatch conditions be disjunctions of conjunctions of dispatch triggers. Relative to BA, BLESS extends the definition of the timeout dispatch trigger to include port lists. Without a port list, BLESS, like BA, defines the timeout starting at the *time-of-previous-suspension* (**tops**). With a port list, BLESS timeouts are reset by any event, sent or received, on a listed port.

Dispatch trigger from timeout occurs when

- an event was sent or received on a listed port (one timeout period in the past),
- no other event has been sent or received by any listed port since then, and
- none of the other dispatch conditions leaving the same source state has occurred since the time-of-previous-suspension (tops).

For transition `T3` events leaving either port `p` or port `n`, reset the timeout.

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

The proof obligation for `T3` in Figure 6 has a complex precondition:

- Assertion of the pace source state invariant: `PACE(now)`.
- Port event LRL occurred previously: `(p or n)@(now-lrl)`.
- No port events to reset timeout:
  `not (exists t:T in now-lrl,,now that (p or n)@t)`.
- No stop since time-of-previous-suspension (tops) [`T2`]:
  `not (exists u:T in tops,,now that stop)`.
- No sense since tops [`T4`]: `not (exists u:T in tops,,now that s@u)`.
- No timeout since tops: `not (exists u:T in tops,,now that`
  `((p or n)@(u-lrl) and`
  `not (exists t:T in u-lrl,,u that (p or n)@t)))`

The full listing of initial proof obligations from which these examples are excerpted, can be found in [14].

```
P [88] <<PACE(now) and (p or n)@(now-lrl)
   and not (exists t:T in now-lrl,,now that (p or n)@t)
   and not (exists u:T in tops,,now  that stop)
   and not (exists u:T in tops,,now that s@u)
   and not (exists u:T in tops,,now that ((p or n)@(u-lrl) and
            not (exists t:T in u-lrl,,u that (p or n)@t)))>>
S [91]<<VP()>>p!<<p@now>> & last_beat := now<<last_beat = now>>
Q [64] <<PACE(now)>>
   What for:  <<M(pace) and x>> A <<M(pace)>> for
   T3_PACE_LRL_AFTER_VP:pace-[x]->pace{A};
```

Fig. 6. Verification Obligation for Transition With Timeout

5 Using the BLESS Proof Tool

5.1 Discharing Proof Oblications

The BLESS proof tool both: (i) generates proof obligations from AADL models having BLESS annex subclauses defining behavior adorned with Assertions, and (ii) transforms proof obligations into simpler ones, with human guidance, successively by applying inference rules.When all proof obligations have been solved, the BLESS proof tool produces a formal proof as a list of theorems, each of which is axiomatic, or derived from earlier theorems in the sequence by a stated inference rule. The last theorem in the sequence asserts that all proof obligations have been discharged.

The BLESS proof tool works as a plug-in to the Open-Source AADL Tool Environment version 2 (OSATE 2) for editing and analyzing AADL models [18], together with a growing family of architectural analysis plug-ins. Typically, after loading a model in BLESS, all of the proof obligations are generated together, but solved one-at-a-time, by applying (groups of) proof rules.

The first proof obligation for VVI is:

```
P [64] <<PACE(now)>>
S [51]->
Q [51] <<LRL(now)>>
```

It shows that the Assertion of complete state `pace`, `<<PACE(now)>>`, needs to imply the thread invariant, `<<LRL(now)>>`. By expanding the named predicates we obtain:

```
P [64] <<p@last_beat and
         (exists t:BLESS_Types::Time in now-lrl..now that p@t )>>
S [51]->
Q [51] <<exists t:BLESS_Types::Time in now-lrl..now
         that (n@t or p@t)>>
```

By splitting existential quantification we get:

```
P [64] <<p@last_beat and
         (exists t:BLESS_Types::Time in now-lrl..now that p@t)>>
S [51]->
Q [51] <<(exists t:BLESS_Types::Time in now-lrl..now that n@t)
     or (exists t:BLESS_Types::Time in now-lrl..now that p@t)>>
```

After some normalization of the predicates we get:

```
P [64] <<(exists t:BLESS_Types::Time in now-lrl..now  that p@t)
   and p@last_beat>>
S [51]->
Q [51] <<(exists t:BLESS_Types::Time in now-lrl..now that n@t )
   or (exists t:BLESS_Types::Time in now-lrl..now that p@t)>>
```

which is then recognized as an axiom of the form $(P \wedge Q) \rightarrow (P \vee Q)$.

Then the next proof obligation is placed in the set of currently unsolved proof obligations. The BLESS proof tool accepts script files, and writes actions to a script file

```
Theorem (1)                                    [serial 1020]
P [64] <<(exists t:BLESS_Types::Time in now-lrl..now that p@t )
     and p@last_beat>>
S [51] ->
Q [51] <<(exists t:BLESS_Types::Time in now-lrl..now that n@t )
     or (exists t:BLESS_Types::Time in now-lrl..now that p@t )>>
by And-Elimination/Or-Introduction Schema:
   (P and Q)->(P or R)

Theorem (2)                                    [serial 1019]
P [64] <<(p@last_beat and (exists t:BLESS_Types::Time
     in now-lrl..now that p@t ))>>
S [51] ->
Q [51] <<((exists t:BLESS_Types::Time in now-lrl..now that n@t )
     or (exists t:BLESS_Types::Time in now-lrl..now that p@t ))>>
by Normalization:
     Reflexivity of Conjunction: (m and k) = (k and m)
     Add Unnecessary Parentheses For No Good Reason: a = (a)
and Theorem (1) [serial 1020]

...

Theorem (4)                                    [serial 1003]
P [64] <<PACE(now)>>
S [51] ->
Q [51] <<LRL(now)>>
by Substitution of Assertion Labels
and Theorem (3) [serial 1018]
Theorem (119)                                    [serial 1002]
P    <<VVI proof obligations>>
S [51] ->
Q    <<VVI proof obligations>>
by Initial Thread Obligations
and theorems 4 8 11 14 28 29 30 50 52 53 74 94 96 97 118:

...

Theorem (97) [serial 1016] used for:
   <<M(check_sense_vrp) and x>> -> <<M(sense)>> for
   T9_VS_AFTER_VS_IN_VRP:check_sense_vrp-[x]->sense{};
Theorem (118) [serial 1017] used for:
   <<M(check_sense_vrp) and x>> A <<M(sense)>> for
   T10_VS_AFTER_VS_IS_NR:check_sense_vrp-[x]->sense{A};
```

Fig. 7. Selected Theorems from the Complete VVI Proof

that can be edited, and invoked, to make re-play of proof strategies easy. When all of the proof obligations have been solved, theorem numbers are assigned depth first, and the complete proof is emitted.

While the current proof process is manual, we plan to capitalize on the growth in power and usability of theorem provers (*e.g.*, SMT solvers) to help automate as much of the proof process as possible.

5.2 Complete Formal Proof for VVI

In all, the complete proof for VVI requires 119 theorems, some of which are illustrated in Figure 7. Sample theorems include: Theorem (4), which discharges the first proof obligation considered in Section 5.1; the last theorem (119) states that all the proof obligations have been discharged.

6 Evaluation

As can be seen from Table 1, we have been writing BLESS architectural specifications on progressively large case studies and attempting to complete verification proofs. The table gives the AADL/BLESS Model name, the number of AADL components in the model and the number of Source Lines Of Code (SLOC). In addition to the VVI pacing

Table 1. BLESS Models

AADL Model	Number of Component	SLOC	Number of Theorems	Number of Proof script steps
VVI pacemaker	1	100	119	101
DDD pacemaker	1	302	1274	582
PulseOx Smart Alarm App	5	949	1003	460
Isolette	15	628	verif. in progress	288
PCA Pump	43	1389	verif. not started	-

mode described here, the examples include DDD pacing mode (a more complex pacing mode in which both atrium and ventricle are paced and sensed instead of just the ventricle as in VVI), an application that implements "smart alarms" for pulse oximetry monitoring, the control logic for an infant incubator (Isolette), and a detailed architectural specification (43 software and hardware components) for a Patient-Controlled Analgesia Pump developed in collaboration with engineers from the US Food and Drug Administration. The verification proofs for the two pacemaker case studies have been completed, as indicated by the availability of the number of theorems required to complete the proof. The last three examples have arisen as part of our work on a Medical Application Platform (MAP) for coordinating collections of networked devices [10]. While these case studies are larger, their verification proofs are still in progress. This is due, in part, to the fact that the models are still evolving. Each case study has allowed us to identify the need to enhance the BLESS tool's set of rules for use in discharging proofs. It has also made evident, especially in the context of evolving models, of the need for further proof automation.

7 Related Work

Much work has been done on formal methods for reasoning about behavioral descriptions in high-level modeling languages such as UML. While model checking applied to notations such as Statecharts and other state machine notations has been well-studied (*e.g.*, [16]), such approaches primarily focus on verifying temporal properties or simple assertions, instead of the strong functional properties expressible in BLESS, or properties that relate directly to timing and scheduling periods in the run-time environment. Relatively little has been done on proof tools for behavioral descriptions in architecture definition languages. In one example, Thums and Balser [22] provide an interactive verification framework for Statecharts based on Dynamic Logic.

The B Method [1] and successor Event-B [2] are nice examples of mature frameworks with tool support that emphasize proof for high-level designs of realistic systems. The Atelier B tool generates proof obligations from behavioral models and provides a manually oriented proof environment. Recent work, *e.g.*, [15], provides tools that translate Atelier B proof obligations into Why3 so as to leverage SMT solvers. Rodin [3] provides proof support for Event B by generating proof obligations. Both B and Event-B focus much more heavily on refinement of high-level semantic descriptions rather than on behaviors in the context of architectural descriptions.

For previous work on verification of the behavioral aspects of AADL models, a translation of a subset of AADL BA into an extension of Petri nets is given by [7], while Ölveczky *et al.* translate AADL BA into Real-Time Maude [17]. These strategies employ model checking and term rewriting, respectively, to verify simple assertions and temporal properties. Thus, they achieve a much higher degree of automation, but treat less expressive specification languages.

Also mentioned in the introduction was the important class of specification language named Behavioral Interface Specification Languages. Notable members of this class include the Java Modeling Language and Spec# (a BISL for C#). Most BISLs are enriched contract-based languages intended for the specification and verification of single-threaded software systems. The KeY Project supports the addition of specifications to software models expressed in UML using languages such as JML and OCL [4, 6].

8 Conclusion

The primary result of this paper—the BLESS framework for behavioral interface specification and verification—represents a significant advance in AADL capabilities that relates directly to AADL's core objectives of supporting rigorous system integration in critical systems. The key technical contributions include designing specifications that capture both functional and timing properties in a manner that aligns with AADL's inter-component communication primitives, scheduling framework, and run-time environment.

Working from the specification notations, formal semantics, and tool support developed here, multiple automated verification methods, including model checking and symbolic execution, can be developed to support verification of AADL component behaviors and interface descriptions. In this first stage of our work, our goal has been to provide tool support that exercises the entirety of the BLESS language, provides strong verification, and provides a formal framework on which more automated, lighter-weight analyses can be built.

In addition to directly supporting interface specification and verification, BLESS annotations can support other facets of critical system development. For example, BLESS is being used in US Army-funded work at the Software Engineering Institute (SEI) on AADL's Error Model Annex to capture system conditions under which faults/errors may arise and propagate. BLESS is also being used in the AADL Requirements Definition and Analysis Language (RDAL) Annex to formally define requirements.

References

1. J.-R. Abrial. *The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.
2. Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, New York, NY, USA, 2010.
3. Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, November 2010.

4. Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hhnle, Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steffen Schlager, and Peter H. Schmitt. The KeY tool. *Software and Systems Modeling*, 4:32–54, 2005.
5. Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system: An overview. *Construction and Analysis of Safe, Secure and Interoperable Smart devices (CASSIS)*, 3362:49–69, 2004.
6. Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt. *Verification of object-oriented software: The KeY approach*. Springer-Verlag, Berlin, Heidelberg, 2007.
7. Bernard Berthomieu, Jean-Paul Bodeveix, Christelle Chaudet, Silvano Zilio, Mamoun Filali, and François Vernadat. Formal verification of AADL specifications in the Topcased environment. In *Proceedings of the 14th Ada-Europe International Conference on Reliable Software Technologies*, pages 207–221, 2009.
8. Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Journal on Software Tools for Technology Transfer (STTT)*, 7(3):212–232, June 2005.
9. Peter H. Feiler, Jorgen Hansson, Dionisio de Niz, and Lutz Wrage. System architecture virtual integration: An industrial case study. Technical Report CMU/SEI-2009-TR-017, 2009.
10. John Hatcliff, Andrew King, Insup Lee, Anura Fernandez, Julian Goldman, Alasdair McDonald, Mike Robkin, Eugene Vasserman, and Sandy Weininger. Rationale and architecture principles for medical application platforms. In *Proceedings of the 2012 International Conference on Cyberphysical Systems*, 2012.
11. John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.
12. Visar Januzaj, Ralf Mauersberger, and Florian Biechele. Performance modelling for avionics systems. In Roberto Moreno-Daz, Franz Pichler, and Alexis Quesada-Arencibia, editors, *Computer Aided Systems Theory - EUROCAST 2009*, volume 5717 of *Lecture Notes in Computer Science*, pages 833–840. Springer, Heidelberg, 2009.
13. Brian R. Larson. *Behavior Language for Embedded Systems with Software Annex Sublanguage for AADL*, 2012. Available at [24].
14. Brian R. Larson, Patrice Chalin, and John Hatcliff. BLESS: Formal specification and verification of behaviors for embedded systems with software. Technical Report SAnToS 2012-12-01, Kansas State University, Computing and Information Sc. Dept., 2012. Available at [24].
15. David Mentré, Claude Marché, Jean-Christophe Filliâtre, and Masashi Asuka. Discharging proof obligations from Atelier B using multiple automated provers. In *Proceedings of the Conf. on Abstract State Machines, Alloy, B, VDM, and Z*, pages 238–251, 2012.
16. Erich Mikk, Yassine Lakhnech, Michael Siegel, and Gerard J. Holzmann. Implementing Statecharts in PROMELA/SPIN. In *Proceedings of the Workshop on Industrial Strength Formal Specification Techniques (WIFT)*, Washington, DC, USA, 1998. IEEE Computer Society.
17. Peter Csaba Ölveczky, Artur Boronat, and José Meseguer. Formal semantics and analysis of behavioral AADL models in Real-Time Maude. In *FORTE'10*, pages 47–62, 2010.
18. Osate 2 web site. `wiki.sei.cmu.edu/aadl/index.php/Osate_2`, 2012.
19. SAE International. *SAE AS5506A. Architecture Analysis & Design Language (AADL)*, 2009.
20. SAE International. *SAE AS5506/2. Architecture Analysis & Design Language (AADL) Annex Volume 2*, 2011.
21. Boston Scientific. Pacemaker system specification. sqrl.mcmaster.ca/pacemaker.htm, 2007.
22. Andreas Thums and Michael Balser. Interactive verification of statecharts. In *Integration of Software Spec. Tech. (INT)*, 2002.
23. System Architecture Virtual Integration (SAVI) Initiative. `wiki.sei.cmu.edu/aadl-/index.php/Projects_and_Initiatives`, 2012.
24. SAnToS TR 2012-12-01 web site. `info.santoslab.org/research/aadl/bless`.