



SCHLOSS DAGSTUHL  
Leibniz-Zentrum für Informatik



# Brian Larson's Criteria Catalogue

## for the Pacemaker Challenge

Brian R Larson

Kansas State University

February 2-7, 2013

Dagstuhl Seminar 14062: The Pacemaker Challenge

This presentation describes involvement with the PACEMAKER System Specification document (the Spec) which is the subject of this Pacemaker Challenge Dagstuhl seminar.



I apologize, in advance, for reading my slides then extemporaneously elaborating the intended message.<sup>1</sup>

---

<sup>1</sup>My colleague at Kansas State University, John Hatcliff, makes beautiful slides, with many pictures, and restrained animation. John even writes-out intended messages and practices delivery together with animations. However, John skillfully uses PowerPoint (which caused me injury last Dagstuhl seminar) and many hours of preparation (which I cannot myself find). Expect four dozen LaTeX-Beamer slides of unrelenting text.



# I (re-)wrote PACEMAKER System Specification

This Dagstuhl seminar's existence is enormously gratifying, because I (largely) caused the PACEMAKER System Specification to be.

This is the story of how that happened.



# Inception

It all started at Formal Methods 2006<sup>2</sup>, at which several papers addressed that year's "challenge" problem: Mondex electronic purse.

Having the same subject helped understanding different formal methods, as applied to a small, well-understood, "toy" subject.

---

<sup>2</sup>hosted by McMaster University in Hamilton, Ontario, Canada

While catching a ride from my hotel to FM06, some person I'd never met before (Jim Woodcock) begged me for a real-world subject to apply formal methods, when he found out I worked for a medical device manufacturer.

Jim wanted something more challenging, more realistic, yet not so complex to overwhelm a small academic team.

I told him I'd see what I could do.



Upon returning to Minnesota, I looked for a system specification, finding all of them too long and detailed, until I talked with an engineer (Gary Seim) that had been with the company (Guidant) since early days.

He gave me a copy of a system specification used for a pacemaker designed in the early 1990s.

It was remarkably concise, but it was company confidential, and was full of Guidant-specific terminology.



# Transformation

Loathing Word, original text converted to LaTeX.

Then the 'Spec' was pruned of Guidant-specific terminology in successive iterations, with review and suggestions by Guidant's Formal Methods Group.

Some pruning global find and replace: PACEMAKER for the all caps codename of the device being engineered.

Others was more subtle, like reducing the subject of the requirements document to a single model, from a family with different subsets of functionality

Chapter 5, was re-written to be as declarative as possible.



Chapter 5 also was re-written while I was developing the math I will show later.

My ulterior motive was to set up a perfect venue for comparison with other formal methodologies.



Not a word in the dozen-page corporate policy on confidentiality expressed how to get a company-confidential document publicly released. Therefore needed VP-level approval.

Review by Legal.

Agreement by McMaster University (Alan Wassying) to host the Spec, maintain FAQ, and aggregate questions so I wouldn't have to keep answering the same ones repeatedly.

Finally, Guidant, now Boston Scientific, released the Spec on January 3, 2007.



# Mathematize Software

Treat programs, their specifications, and their executions as mathematical objects,

such that proofs can be constructed that every program execution conforms to its specification.



The attendees of this seminar were the target of the Spec all along.

You,

- are top people with depth and breath of accomplishment in formal methods, and can do the math.
- have studied and used the Spec in your own research and teaching.
- know the subject, so will readily understand how the words in the spec become formal specification.
- can visualize *sets* of behaviors, starting with VVI pacing as subject.
- will understand a proof (VVI) that an entire set of possible behaviors conforms to its specification.
- will forevermore understand the words of the Spec in terms of a simple temporal logic.



## Emerging Research

### Available now:

- Eclipse (OSATE<sup>3</sup>) plug-in at update site: [BLESSUpdateSiteURL](#)
- LRM (etc.), example Eclipse projects for VVI.aadl and DDD.aadl with scripts and proofs: [PACEMAKERExamplesURL](#)

### Staying in Europe to work with:

- AdaCore for Project-P for DO-178C certifiable code generation
- Telecom ParisTech for RAMSES code generator for BA

---

<sup>3</sup>Open-Source AADL Tool Environment



# Goal

Complete system development environment integrating architecture behavior and proofs with tool chain generating certifiable (DO-178c) code

Trustworthy code generation from proved-correct programs

Half my life.



only pulse generator for proof  
all for architecture modeling

# Software

none

All

Architecture Analysis and Design Language<sup>4</sup> (AADL);  
plus the math I made-up<sup>5</sup> to define language semantics.

---

<sup>4</sup>SAE International standard AS5506B

<sup>5</sup>Mike Whalen, University of Minnesota, likes to point out that I made up much of the math, so how does he know the “proofs” mean anything at all? Because I’m going to the trouble of constructing a soundness proof for the math I made up, I will demand that Mike scrutinize the proof.



## Why?

AADL was designed for safety-critical systems engineering, has defined semantics, a tool that checks all the legality and naming rules, and many architecture analysis plug-ins. AADL defines annex subclauses and libraries so the core architecture language can be extended in orderly ways. AADL expresses system structure well.

The math I made up was needed to treat programs (behavior), specifications, and executions as mathematical objects.

Absolutely! Mathematizing software is *raison d'être* for the Spec.  
AADL defines some standard, system services in terms of timed automata.



# Math is Hard

Merely specifying behavior formally is challenging.

Writing behavior together with its proof outline is harder.

Even with help from a tool, choosing proof strategies that transform proof outlines into formal proofs is harder still.





## Simulation is Worthy

Putting together a Simulink model, and a description of the environment, and simulating has been, and will be, a reasonable thing to do.<sup>6</sup>

Practicing engineers want to do both:

- slap together a quick-and-dirty executable model to simulate behavior with its environment (early validation),
- later formalizing specifications and behaviors *incrementally* to prove crucial safety and efficacy properties hold.

---

<sup>6</sup>I want to build upon the good work Mats Heimdahl's group is doing, but haven't been able to find textual grammar/semantics for Stateflow. If anyone knows of an open source equivalent to Simulink with state-transition machine behavior, please let me know.

Everything not software

Why? Software is problematic.

Software can be perfect in a way that machines<sup>7</sup> can never be.

Yet software is (mostly) crap.

Software needs math to be good.

---

<sup>7</sup>unavoidable manufacturing variation, wear, damage, and cosmic ray induced neutron bit-flips

Software.

Only when software engineering treats its subject as a mathematical object, like every other engineering discipline, will software be engineered.


unmeasurable at this time

LRL, VRP, ARP, URL, AV-delay, PVARP,

it depends what's meant by 'construction'

Transforming proof outlines<sup>8</sup> into formal proofs.

---

<sup>8</sup>A proof outline annotates a program with logic formulas that define what in (supposed to be) true about values of variables at particular points in execution. Proof outlines for state transition systems attach logic formulas to state declarations too. 



Proving software correctness is novel; software engineers will need training.

The math was intended to be easy, once learned, with libraries of proved correct component behaviors.

There seem to be archetypes, which once proved correct<sup>9</sup> can be easily tweaked. As more archetypical component examples are available, the easier it will be to prove correctness.

---

<sup>9</sup>including proof strategy script



# Validation by Inspection

Check that math expresses message.

Inspect that the formal specification (math) faithfully expresses the meaning of the natural language of the Spec.

Want to generate Simulink/Stateflow from behavior for validation by early simulation with continuous-time model of environment.<sup>10</sup>

---

<sup>10</sup>Such simulation/validation is the only thing that UofM's PCA pump AADL model does that K-State's PCA pump AADL model doesn't.

Is the functional design verified against the requirements specification?

Yes. In my view:

Requirements  $\implies$  Specification  $\implies$  Implementation

**Requirements** Natural language expression of customer need.  
Domain experts *validate* requirements do the right thing.

**Specification** Requirements expressed formally. Inspection to *validate* that math correctly abstracts the intent of the words

**Implementation** The device, approved, manufactured, and shipped.  
Verification that implementation meets specification has many forms of definitive evidence: proofs, tests, static analysis, model checking, etc.

Finally, system feature tests (SFT<sup>11</sup>) *validate* that implementation meets requirements, bypassing specification.

---

<sup>11</sup> Although SFTs are intended for validation, they can reveal defects that verification should have caught, but didn't.

no

To prove VVI.aadl and DDD.aadl, I adapted the Java proof tool I made to transform proof outlines of highly-concurrent programs<sup>12</sup> to use simple temporal formulas instead of first-order predicates.

Java application became plug-in to OSATE which is a plug-in to Eclipse. OSATE has many analysis tools for AADL architectures.

Uses over 50 ANTLR grammars to do logic on ASTs.

20k LoC, human-written becomes 200k Java source.

---

<sup>12</sup>DANCE

Brand new.

Now, much. Early adopters will be seminar-level experts.

As libraries of proved-correct components make putting together proved-correct systems easier, the expertise will be lowered.

Pedagogical material must be created.

Semantics made as simple as possible. Eventually, high school students who understand that  $X$  is a placeholder for a value that is unknown or may change, will be able to prove their programs are correct.



proofs of vital safety and efficacy properties

## DO-17c qualifiable code generation

The Spec provided mental model of timing-critical embedded system actions.

The Spec provided the first two behaviors to be proved correct .

VVI.aadl used as “Hello world!” program for NFM2013 paper

Tutorial(s)

Industry Advisor to UMN Senior ECE Design Project: complete pulse generator PCB design<sup>13</sup>, fab, and programing for all pacing modes, in one semester.

Following semester's design project to make a heart simulator by inverting the Spec failed.

---

<sup>13</sup>adapted by Mark Lawford at McMaster U and built for teaching platforms

To provide an intellectual soap-box to evangelize a powerful formal method, fundamentally-different from model-checking, static analysis, or theorem proving.



After the break, quotations from the Spec will be compared with its formalization in a simple extension of first-order predicate calculus. Request: please make notes comparing *your* formal method with mine:

- How are the formal methods similar? How is your formal method better? What can your formal method do that mine can't? Similar work by others?<sup>14</sup>
- Want to get the proof tool, run it on VVI.aadl and DDD.aadl yourself, to see what it does and how it works?
- ~~Want to try writing and proving your own behavior?~~<sup>15</sup>

<sup>14</sup>references, please. I'm unaware of anyone else (semi-)automatically transforming proof outlines into correctness proofs.

<sup>15</sup>I have many potential behaviors that can use VVI.aadl and DDD.aadl for inspiration, yet will still be an original proof: hysteresis pacing, rate response, magnet mode, AV-squeeze. Others, non-cardiac.