Compositional Safety and Security Analysis of Architecture Models

Mike Whalen

Program Director

University of Minnesota Software Engineering Center

University of Minnesota

Software Engineering Center

Acknowledgements

- Rockwell Collins (**Darren Cofer**, **Andrew Gacek**, Steven Miller, Lucas Wagner)
- UPenn: (Insup Lee, Oleg Sokolsky)
- UMN (Mats P. E. Heimdahl)
- CMU SEI (Peter Feiler)

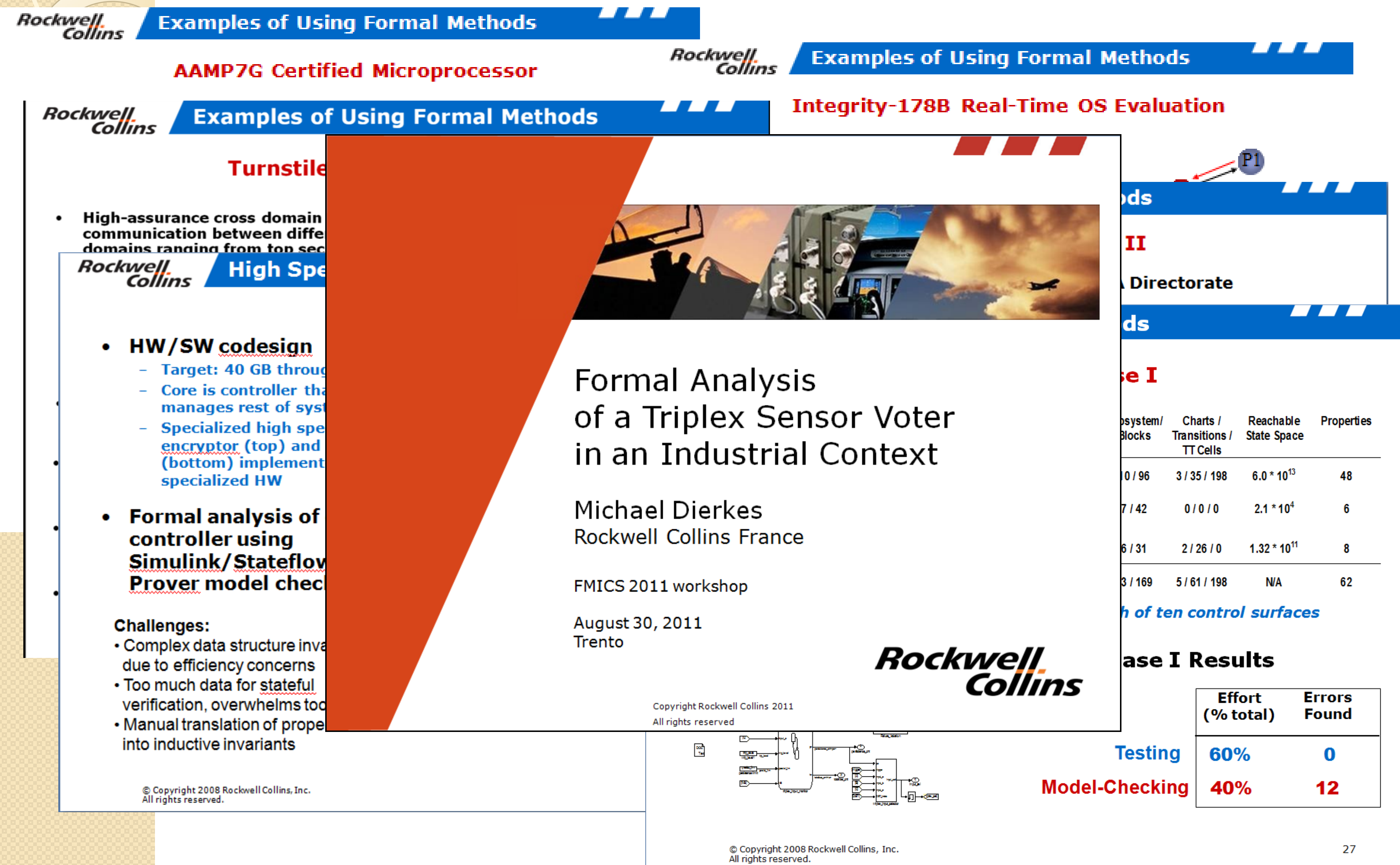
Main Messages

We need a variety of reasoning approaches and partitioning methods for system-level requirements and analysis

Your How is My What: *requirements vs. design is a often matter of perspective*

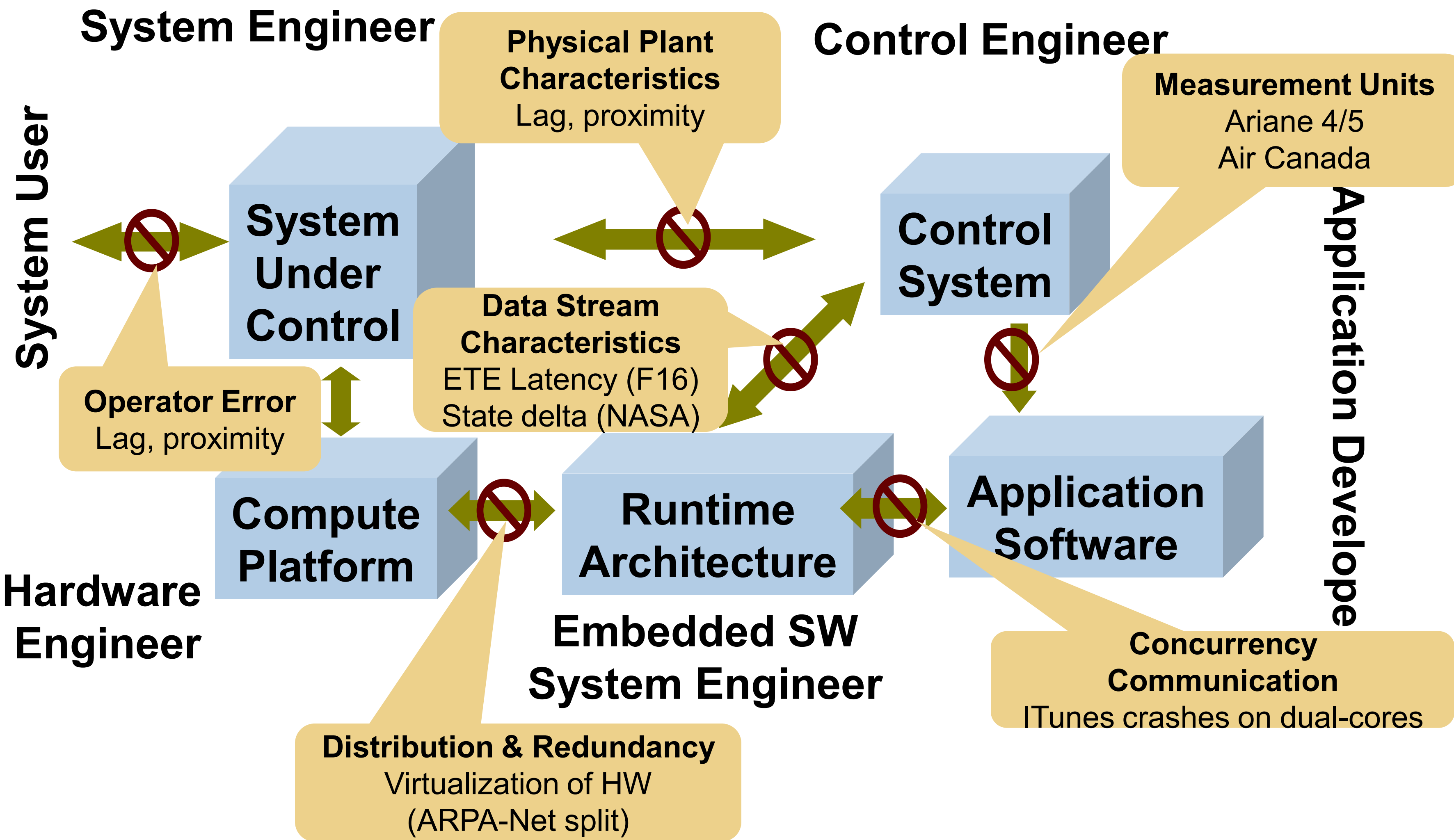
Requirements hierarchies often follow system and software architectures.

Component Level Formal Analysis Efforts




YOU
ARE
FIRED

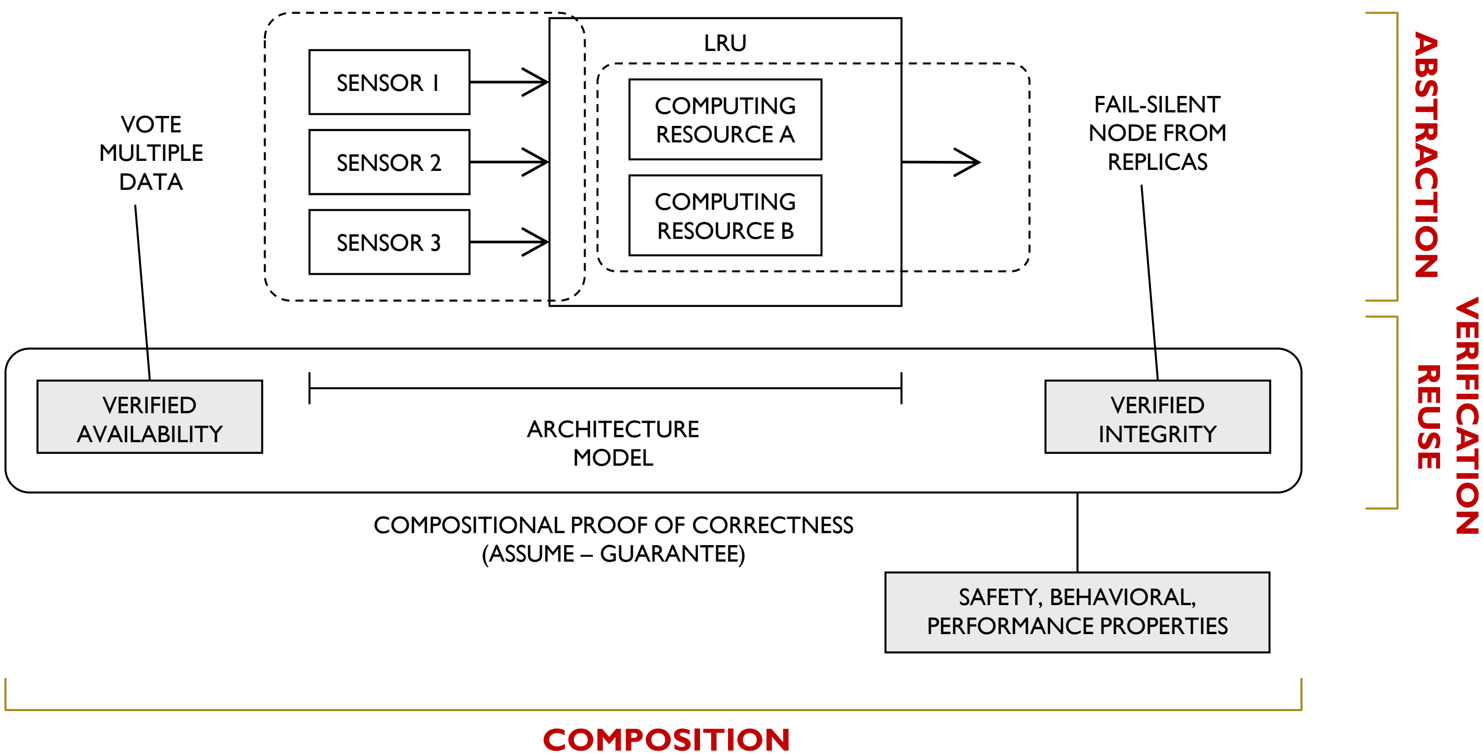
Mismatched Assumptions



Slide from: An Overview of AADL v2 by Peter Feiler, 2010

Vision

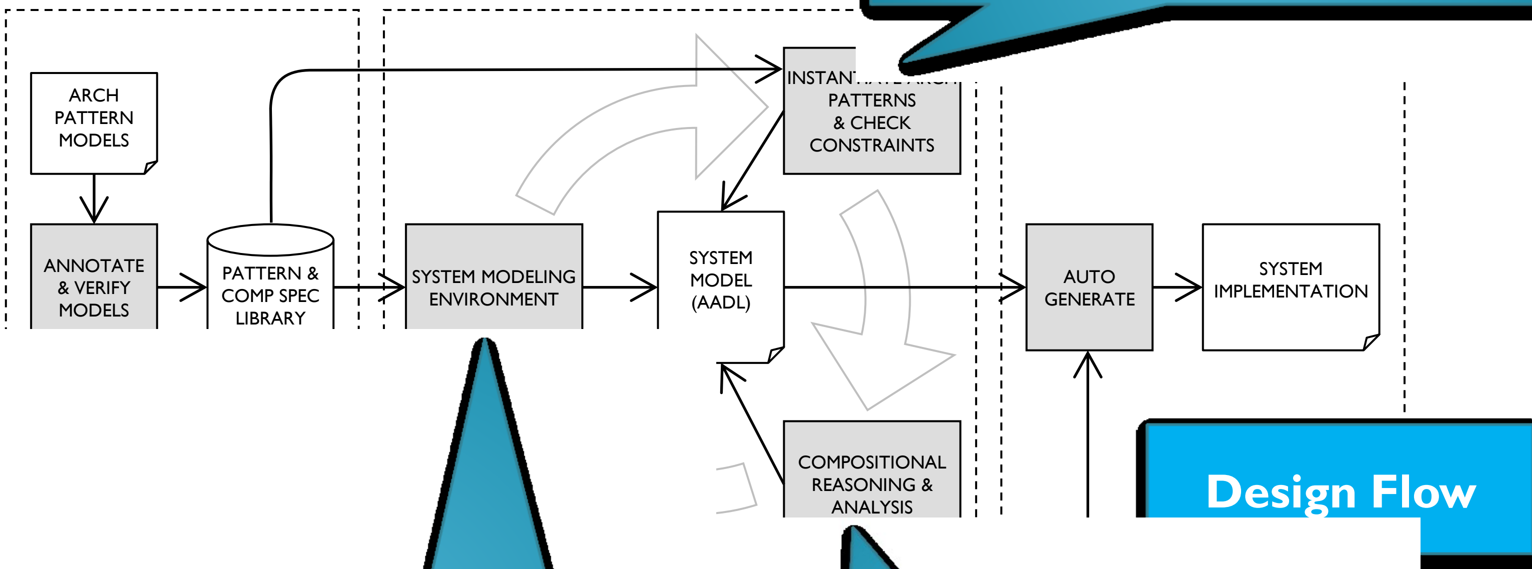
System design & verification through pattern application and compositional reasoning



Approach

Complexity-reducing design patterns
- Capture best solutions to architectural design problems
- Reuse of formally verified solutions
- Increase level of design abstraction

2



System architecture modeling
- Apply formal specification and analysis tools to system-level design
- Separate component specification and implementation
- Automated model translation

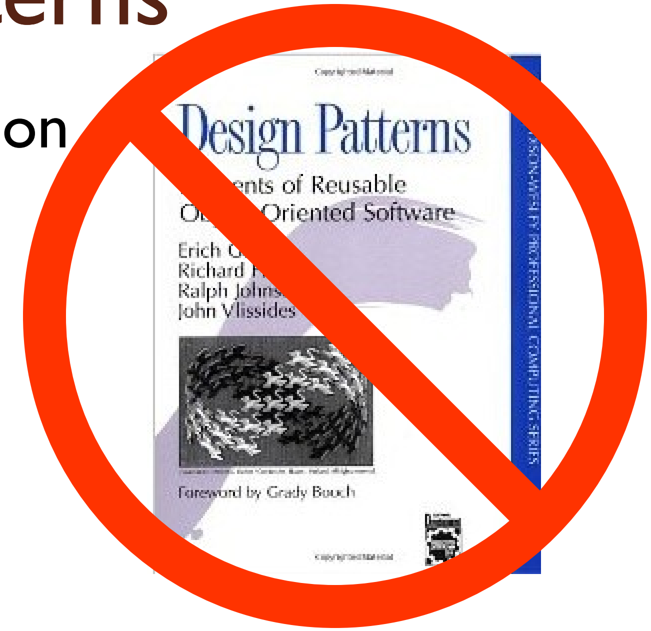
1

Compositional verification

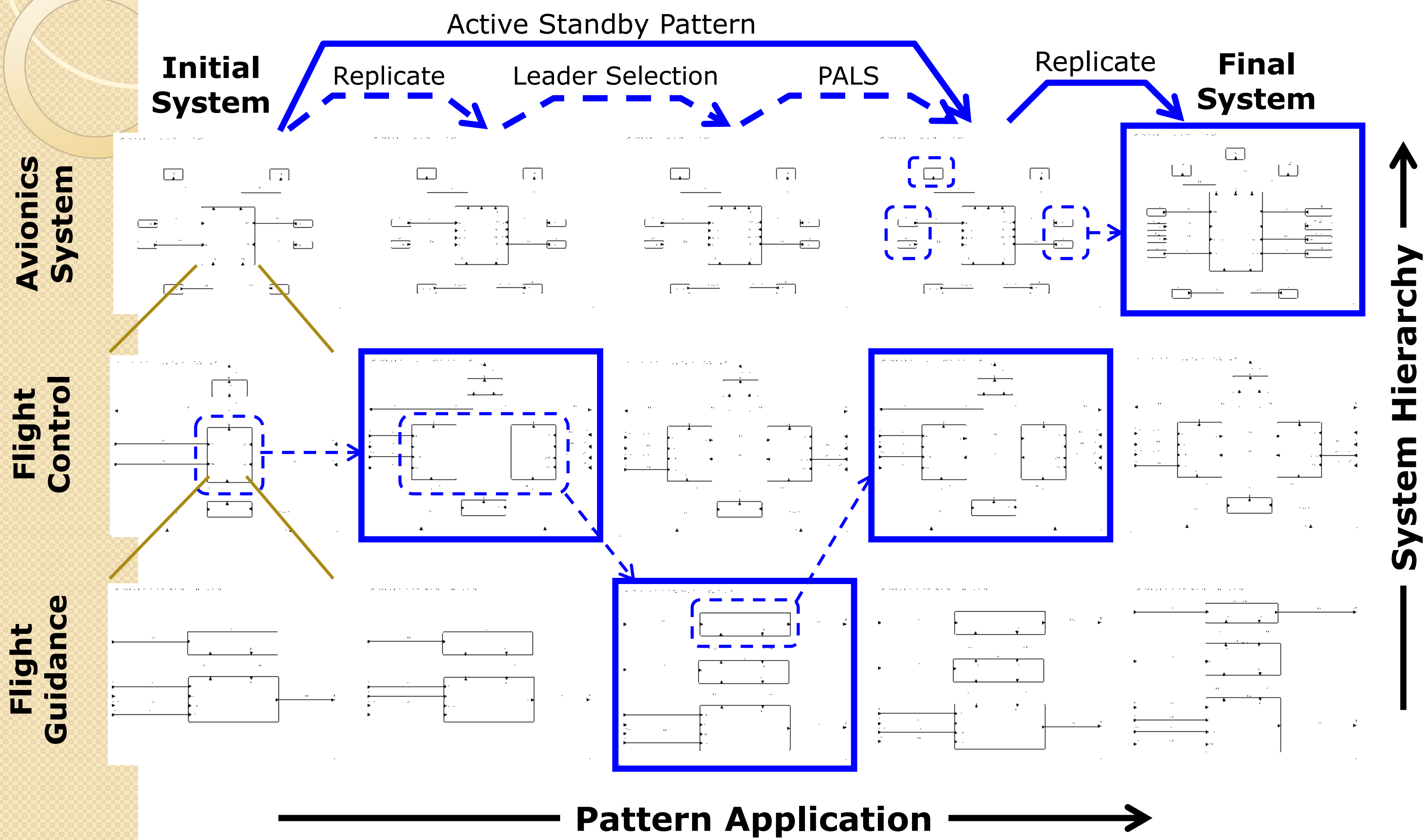
3
- Reason about system behavior based on contracts and system design model structure
- Compositional approach scales to large software systems

Complexity-Reducing Architectural Design Patterns

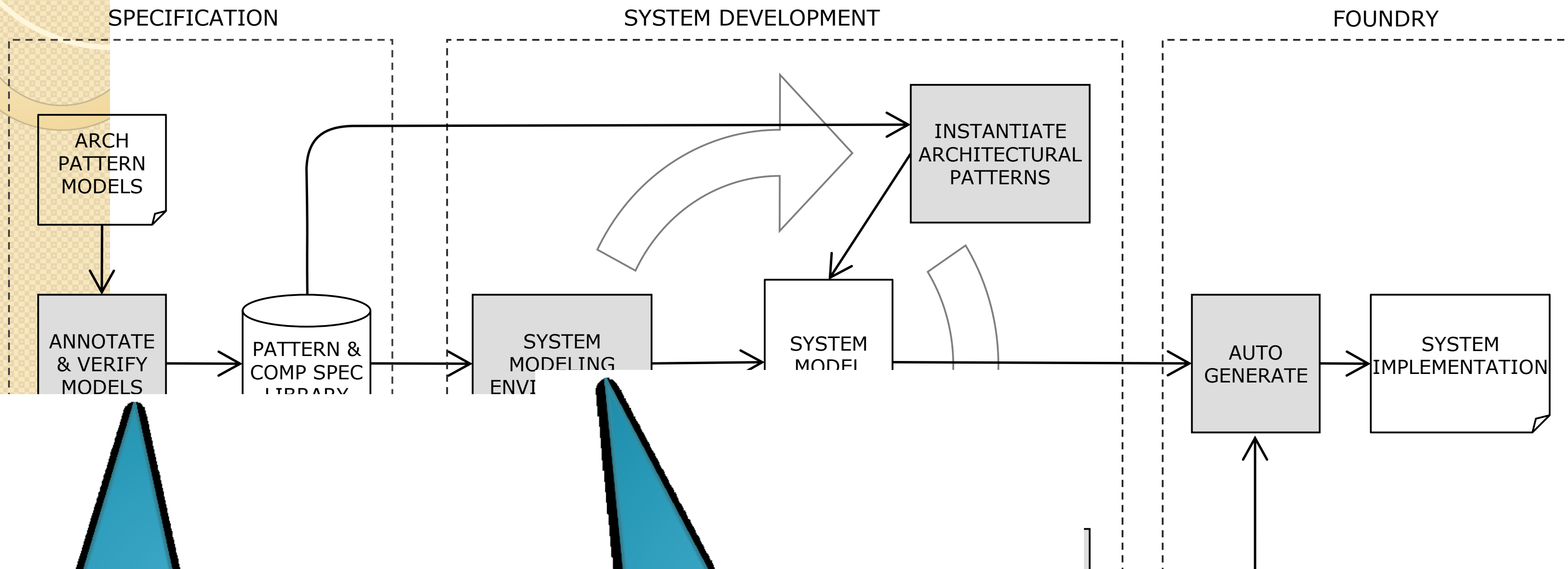
- Design pattern = model transformation
  - $p : \mathcal{M} \rightarrow \mathcal{M}$ (partial function)
  - Applied to system models
- Reuse of verification is key
  - Not software reuse
  - Guaranteed behaviors associated with patterns (and components)
- Reduce/manage system complexity
  - Separation of concerns
  - System logic vs. application logic (e.g., fault tolerance)
  - Process complexity vs. design complexity
- Encapsulate & standardize good solutions
  - Raise level of abstraction
  - Codify best practices



System Design Through Pattern Application



System verification



Reusable Verification:
Proof of component and pattern requirements (guarantees) and specification of context (assumptions)

Instantiation:
Check structural constraints, Embed assumptions & guarantees in system model

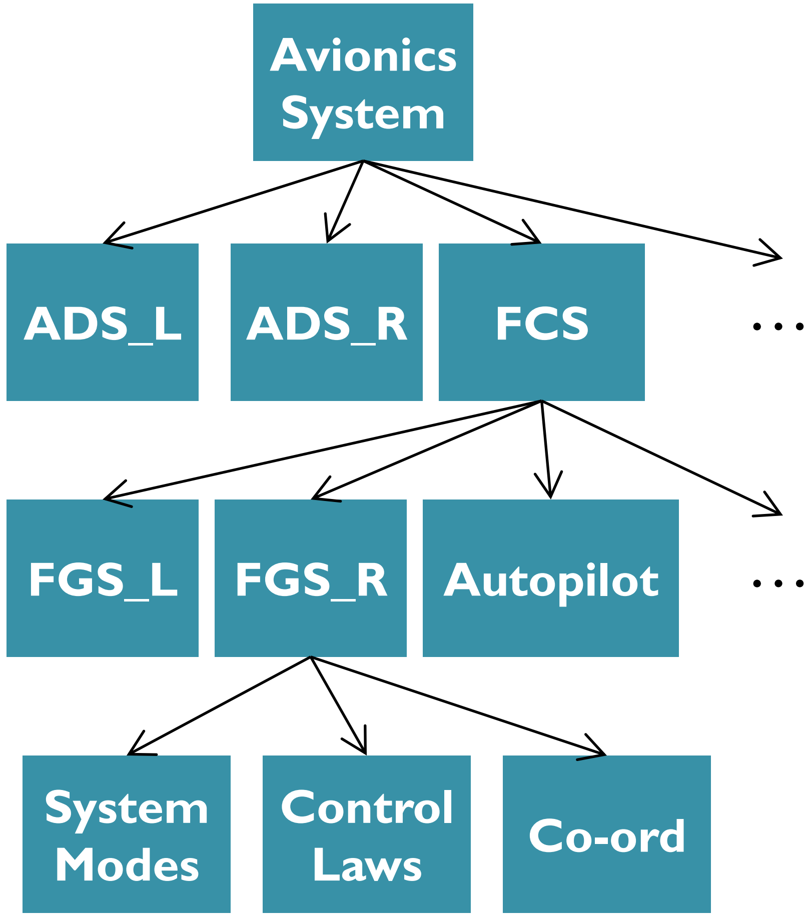
Compositional Verification:
System properties are verified by model checking using component & pattern contracts

Hierarchical reasoning about systems

- Avionics system requirement

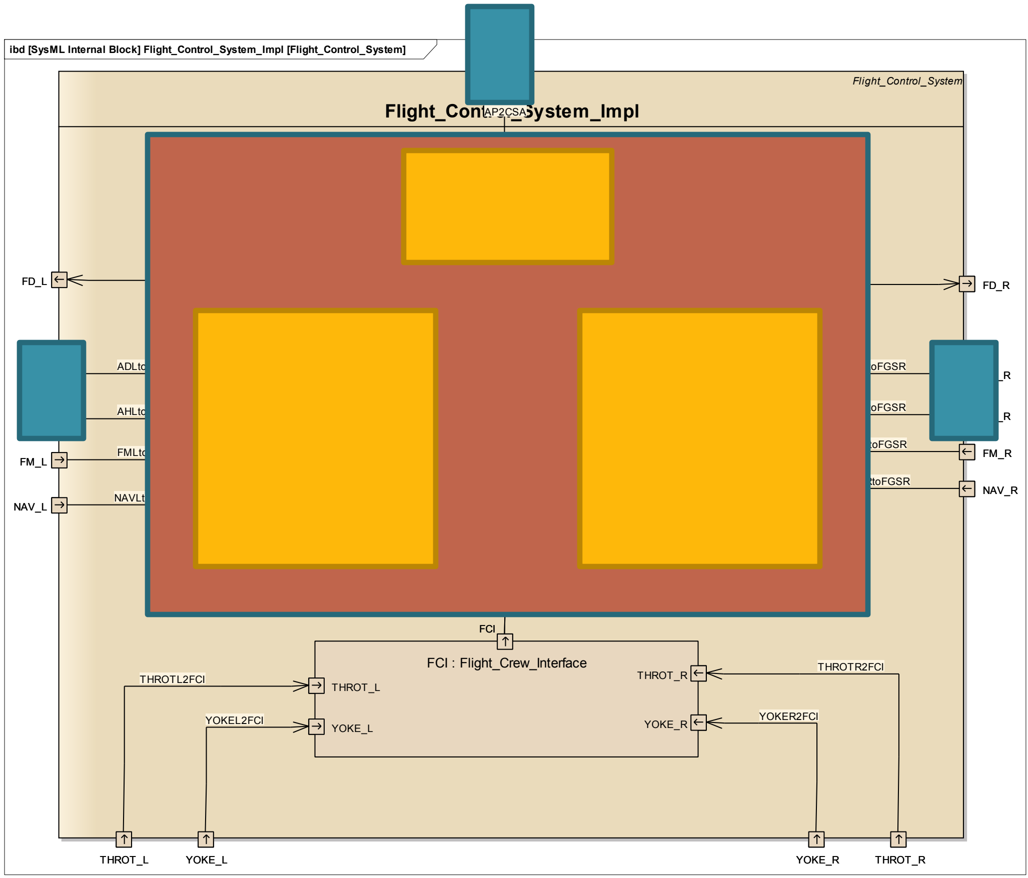
Under single-fault assumption, GC output transient response is bounded in time and magnitude

- Relies upon
  - Accuracy of air data sensors
  - Control commands from FCS
    - Mode of FGS
    - FGS control law behavior
    - Failover behavior between FGS systems
    - ….
  - Response of Actuators
  - Timing/Lag/Latency of Communications



Compositional Reasoning for Active Standby

- Want to prove a **transient response** property
  - The autopilot will not cause a sharp change in pitch of aircraft.
  - Even when one FGS fails and the other assumes control
- Given assumptions about the **environment**
  - The sensed aircraft pitch from the air data system is within some absolute bound and doesn't change too quickly
  - The discrepancy in sensed pitch between left and right side sensors is bounded.
- and guarantees provided by **components**
  - When a FGS is active, it will generate an acceptable pitch rate
- As well as **facts** provided by pattern application
  - Leader selection: at least one FGS will always be active (modulo one "failover" step)




```
transient_response_1 : assert true ->
  abs(CSA.CSA_Pitch_Delta) < CSA_MAX_PITCH_DELTA ;
transient_response_2 : assert true ->
  abs(CSA.CSA_Pitch_Delta - prev(CSA.CSA_Pitch_Delta, 0.0))
     < CSA_MAX_PITCH_DELTA_STEP ;
```

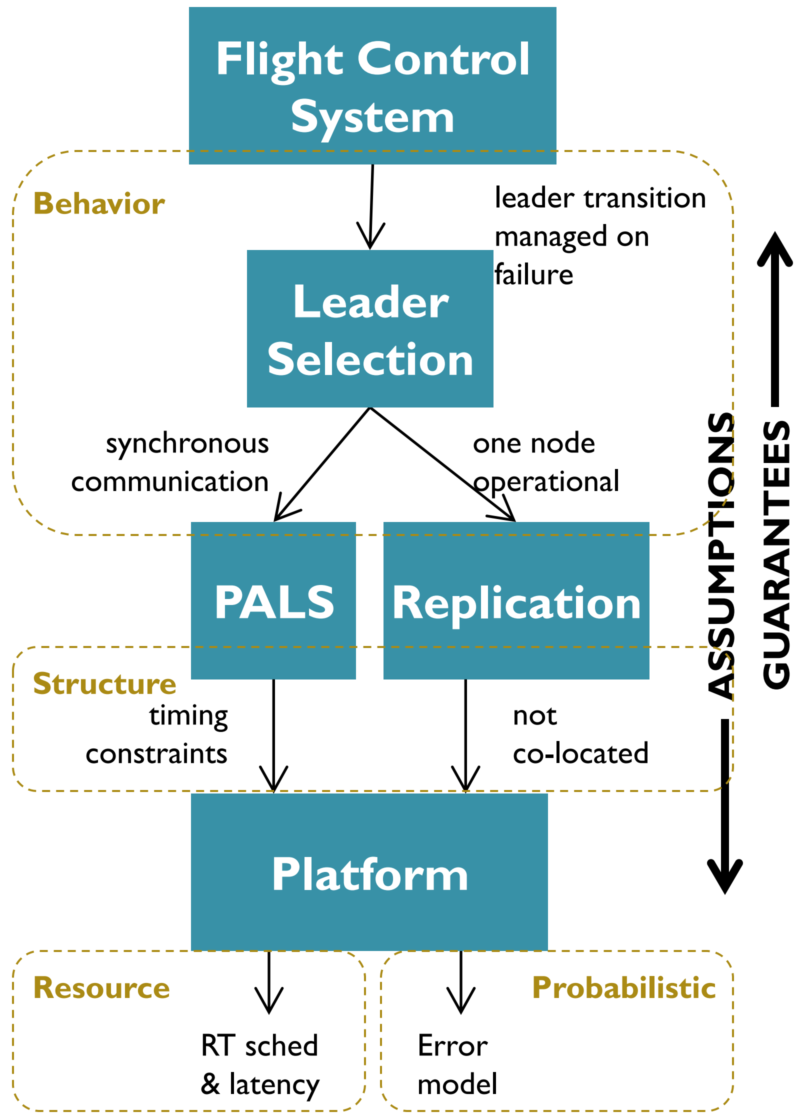
Hierarchical reasoning between analysis domains.

- Avionics system requirement

Under single-fault assumption, GC output transient response is bounded in time and magnitude

- Relies upon
  - Guarantees provided by patterns and components
  - Structural properties of model
  - Resource allocation feasibility
  - Probabilistic system-level failure characteristics

Principled mechanism for "passing the buck"



Contracts

- Derived from Property Specification Language (PSL) formalism
  - IEEE standard
  - In wide use for hardware verification
- Assume / Guarantee style specification
  - Assumptions: "Under these conditions"
  - Promises (Guarantees): "…the system will do X"
- Local definitions can be created to simplify properties

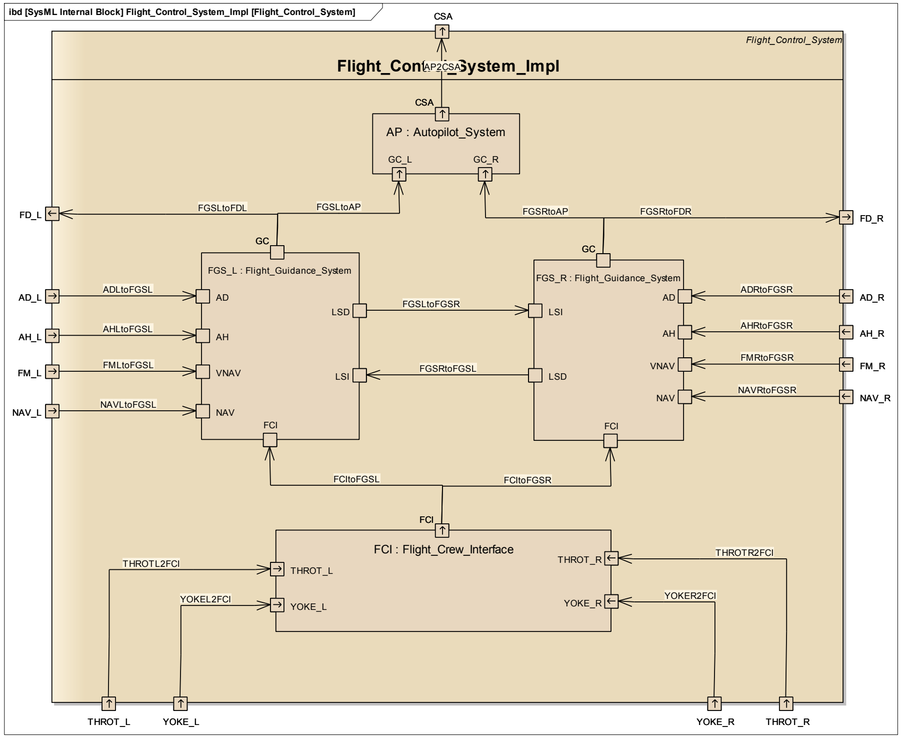
```
Contract:

fun abs(x: real) : real = if (x > 0) then x else -x ;

const ADS_MAX_PITCH_DELTA: real = 3.0 ;
const FCS_MAX_PITCH_SIDE_DELTA: real = 2.0 ;
const CSA_MAX_PITCH_DELTA: real = 5.0 ;
const CSA_MAX_PITCH_DELTA_STEP: real = 5.0 ;

property AD_L_Pitch_Step_Delta_Valid =
  true ->
    abs(AD_L.pitch.val - prev(AD_L.pitch.val, 0.0)) < ADS_MAX_PITCH_DELTA ;

property AD_R_Pitch_Step_Delta_Valid =
  true ->
    abs(AD_R.pitch.val - prev(AD_R.pitch.val, 0.0)) < ADS_MAX_PITCH_DELTA ;

property Pitch_lr_ok =
  abs(AD_L.pitch.val - AD_R.pitch.val) < FCS_MAX_PITCH_SIDE_DELTA ;

property some_fgs_active =
  (FD_L.mds.active or FD_R.mds.active) ;

active_assumption: assume some_fgs_active ;

transient_assumption :
  assume AD_L_Pitch_Step_Delta_Valid and
         AD_R_Pitch_Step_Delta_Valid and Pitch_lr_ok ;


transient_response_1 :
  assert true -> abs(CSA.CSA_Pitch_Delta) < CSA_MAX_PITCH_DELTA ;
transient_response_2 :
  assert true ->
      abs(CSA.CSA_Pitch_Delta - prev(CSA.CSA_Pitch_Delta, 0.0)) <
      CSA_MAX_PITCH_DELTA_STEP ;
```

Reasoning about contracts

- Notionally: *It is always the case that if the component assumption is true, then the component will ensure that the guarantee is true.*
  - $G(A \Rightarrow P)$;
- An assumption violation in the past may prevent component from satisfying current guarantee, so we need to assert that the assumptions are true up to the current step:
  - $G(H(A) \Rightarrow P)$;

Systems of Contracts

- Architectures are hierarchically composed in layers.
  - Visually: a box and line diagram
  - Formally you can view a layer as a *system* S:
    S = (A, P, C)
    - C is a finite set of component contracts
      C: ℙ (A x P)



Reasoning about Contracts

- Given the set of component contracts:

  $\Gamma = \{ G(H(A_c) \Rightarrow P_c) \mid c \in C \}$

- Architecture adds a set of obligations that tie the system assumption to the component assumptions

$$Q = \{H(A_s) \implies P_s\} \cup \{H(A_s) \implies A_c \mid c \in C\}$$

- This process can be repeated for any number of abstraction levels

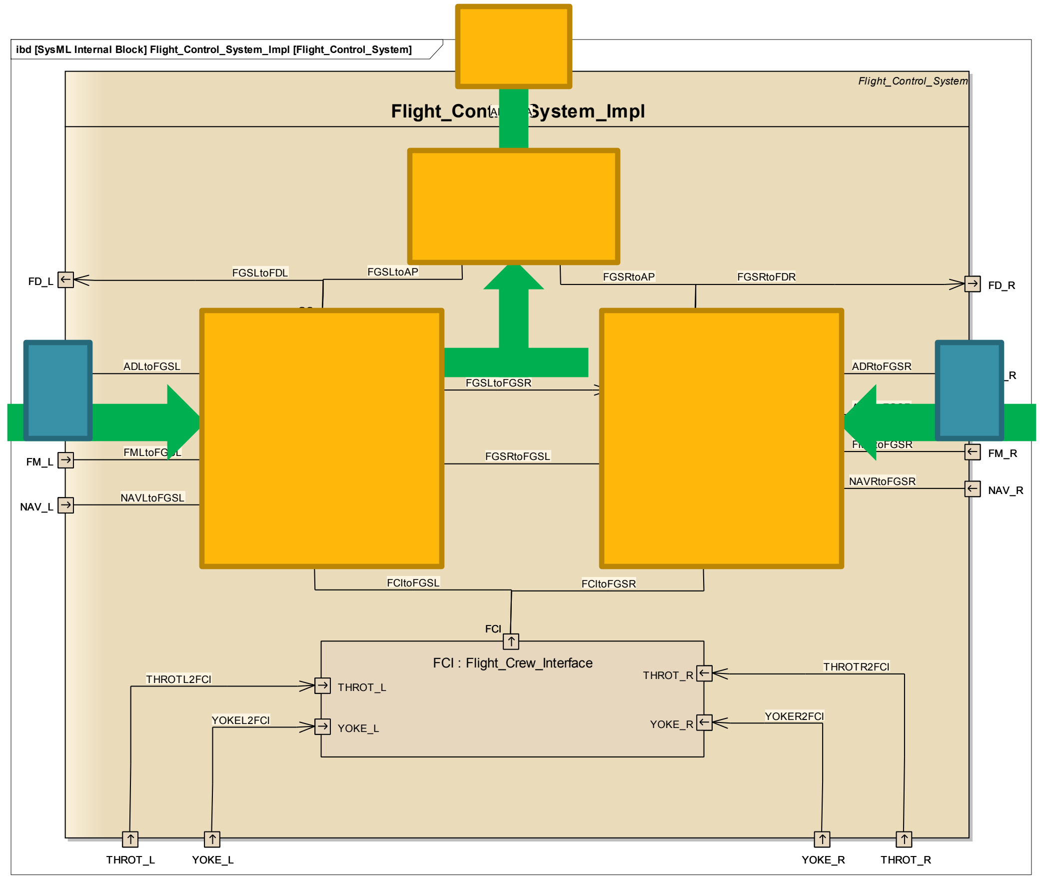
Composition Formulation

- Suppose we have
  - *Sets of formulas* Γ *and* Q
  - *A well-founded order* $\prec$ *on* Q
  - *Sets* $\Theta_q \subseteq \Delta_q \subseteq Q$, *such that* $r \in \Theta_q$ *implies* $r \prec q$
- Then if for all q ∈ Q
  - $\Gamma \Rightarrow G((Z(H(\Theta_q)) \wedge \Delta_q) \Rightarrow q)$
- Then:

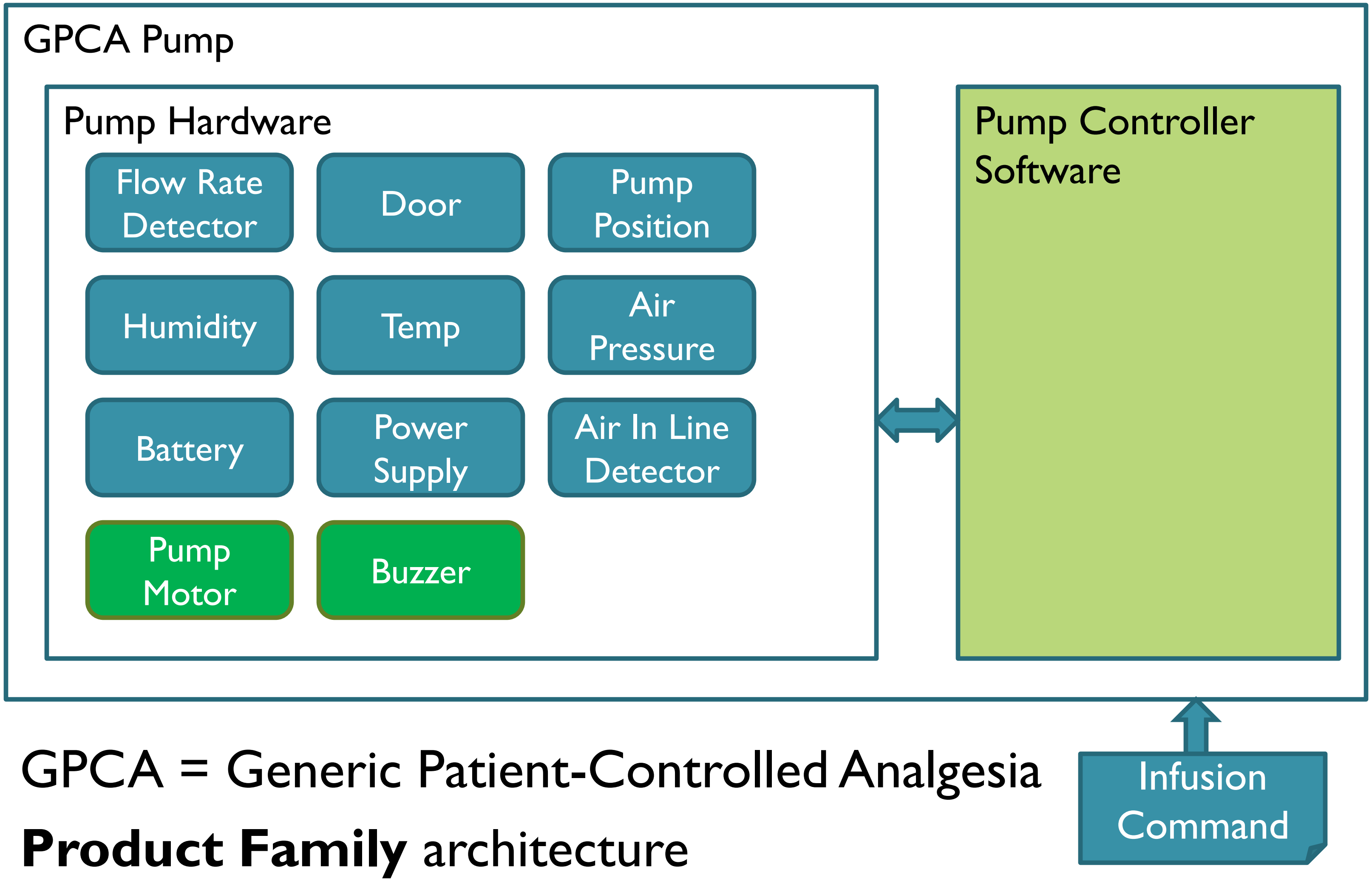
  G(q) for all q ∈ Q
- [Adapted from McMillan]

A concrete example

- Order of data flow through system components is computed by reasoning engine
  - {System inputs} → {FGS_L, FGS_R}
  - {FGS_L, FGS_R} → {AP}
  - {AP} → {System outputs}
- Based on flow, we establish four proof obligations
  - System assumptions → FGS_L assumptions
  - System assumptions → FGS_R assumptions
  - System assumptions + FGS_L guarantees + FGS_R guarantees → AP assumptions
  - System assumptions + {FGS_L, FGS_R, AP} guarantees → System guarantees
- System can also handle circular flows, but user has to choose where to break cycle



Architecture of Generic Infusion Pump



- GPCA = Generic Patient-Controlled Analgesia
- **Product Family** architecture

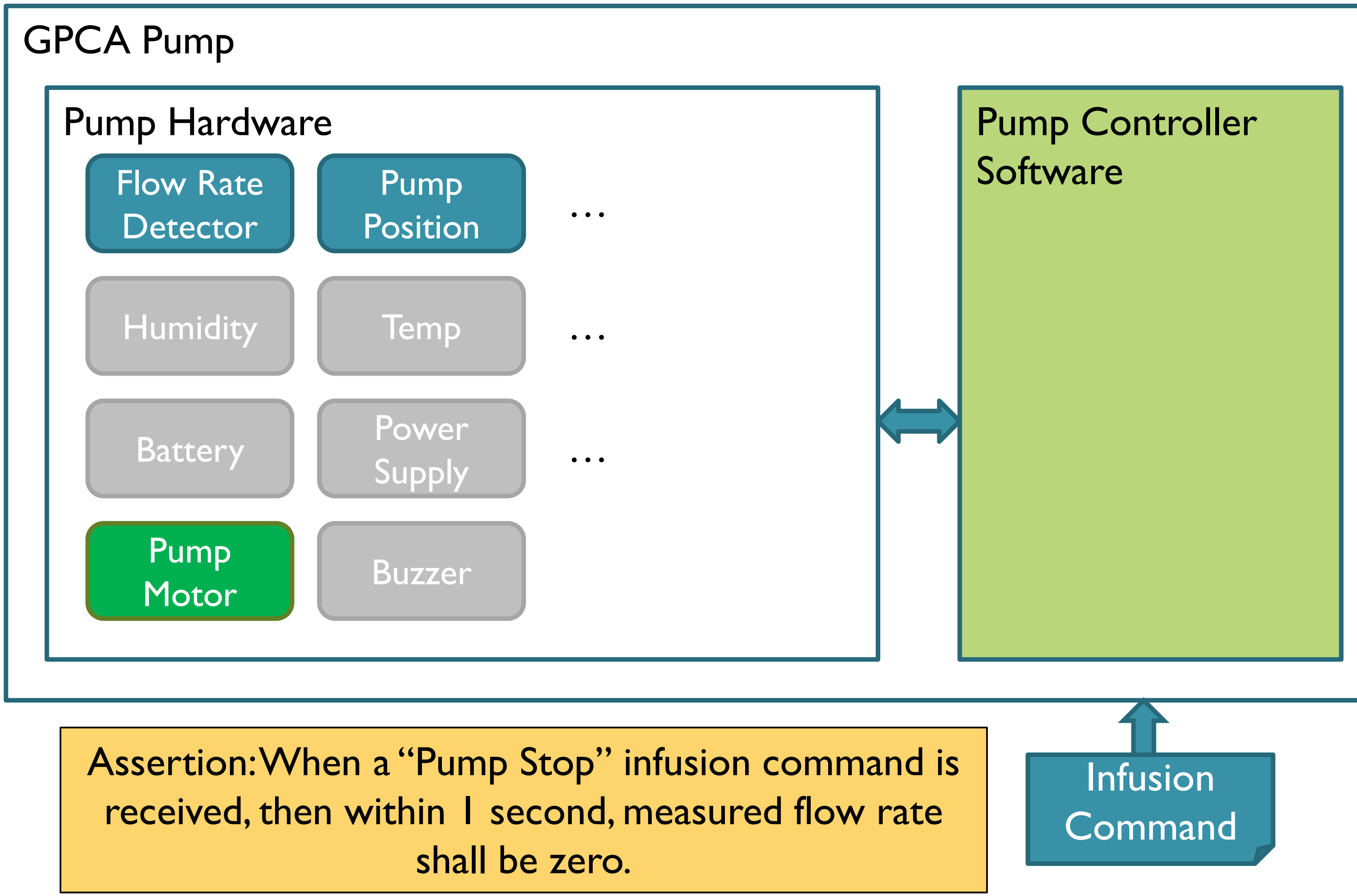
GPCA Pump Example

- Property of Interest:
  - If a "Pump Stop" command is received, then within 1 second, measured flow rate shall be zero.

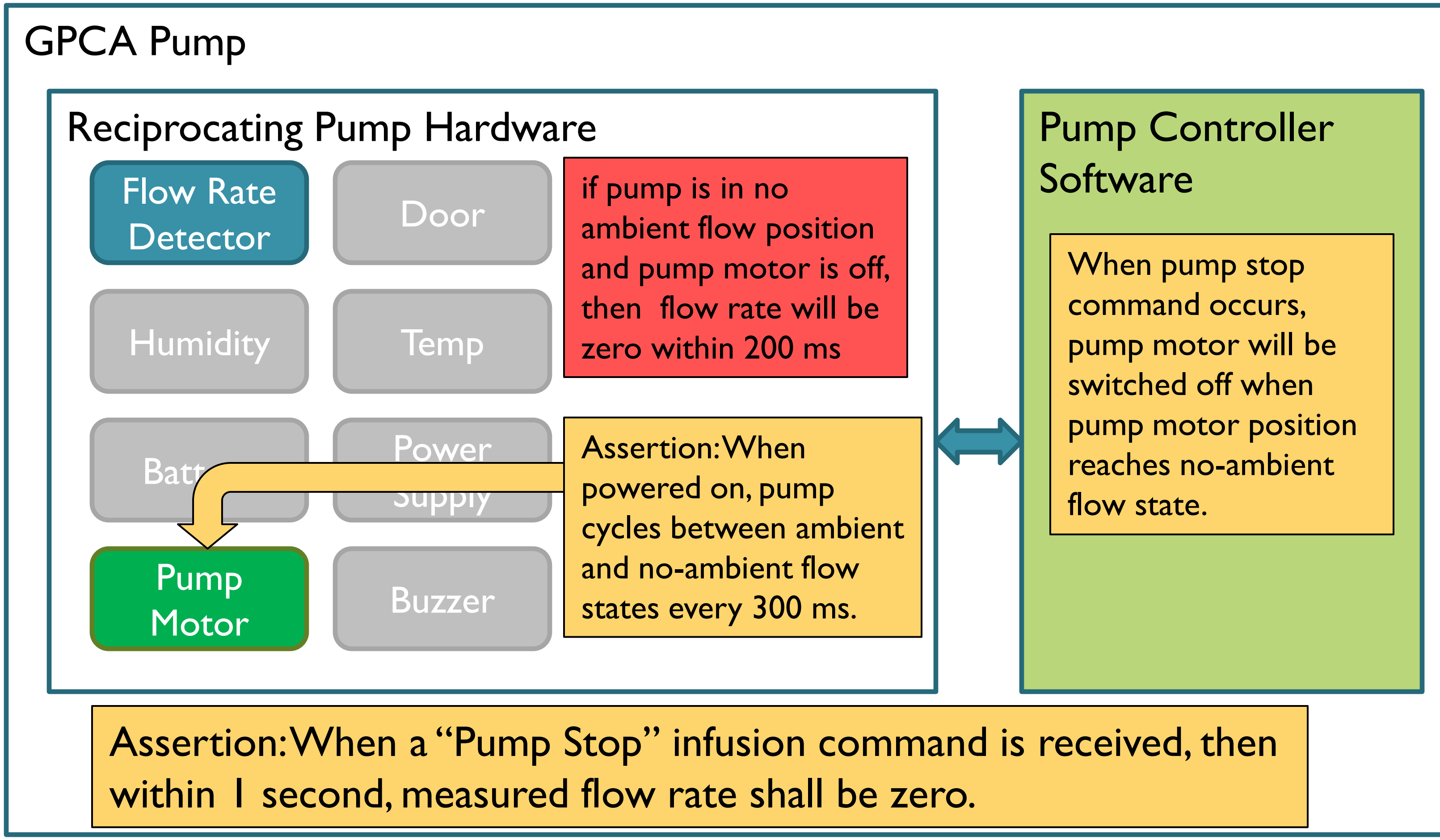
- We will prove this property compositionally based on the architecture of the Pump subsystem.

With exciting verification demo!

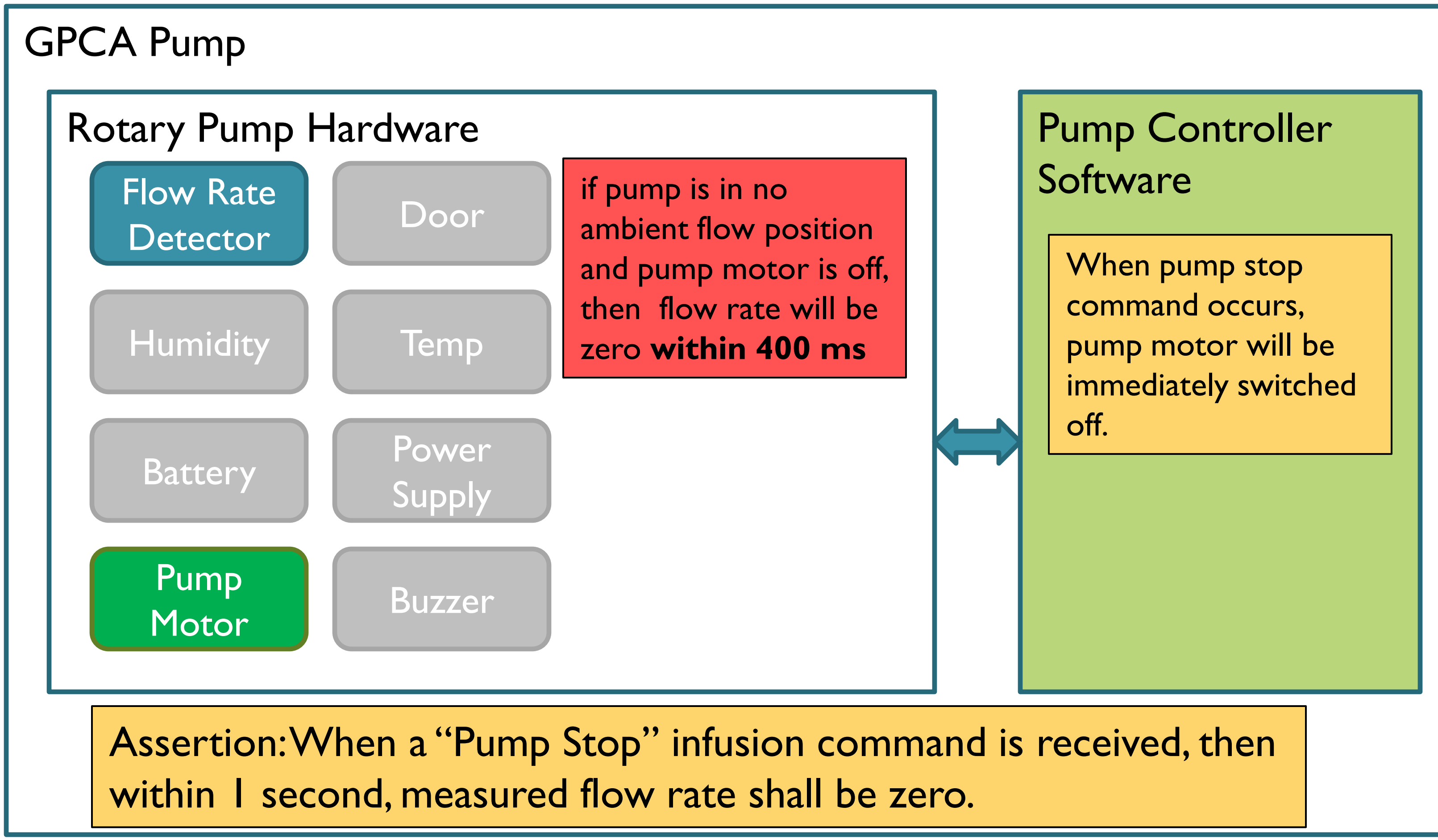
Proof of GPCA Pump



Proof of Reciprocating Pump



Proof of Rotary Pump



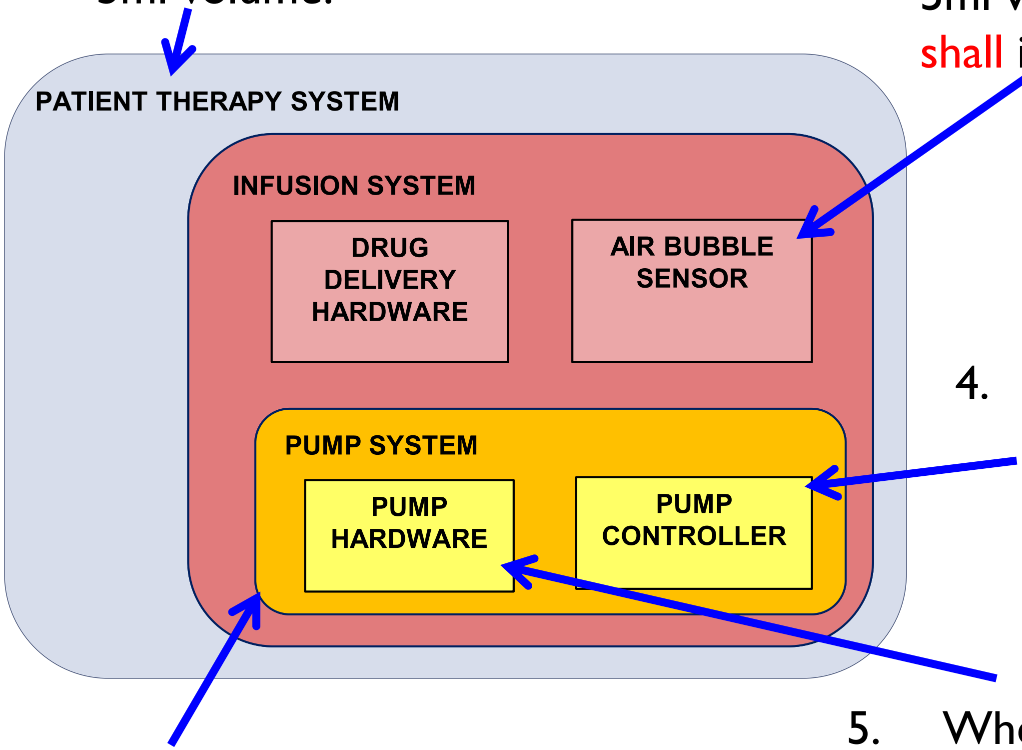
ARCHITECTURE AND REQUIREMENTS

Requirements or Design Information?

1. The patient shall never be infused with a single air bubble more than 5ml volume.
2. When a single air bubble more than 5ml volume is detected, the system shall stop infusion within 0.01 seconds.
3. When a single air bubble more than 5ml volume is detected, the system shall issue an air-embolism command.
4. When air-embolism command is true, the system shall stop infusion.
5. When air-embolism command is received, the system shall stop piston movement within 0.1 second.

A: Both

1. The patient shall never be infused with a single air bubble more than 5ml volume.



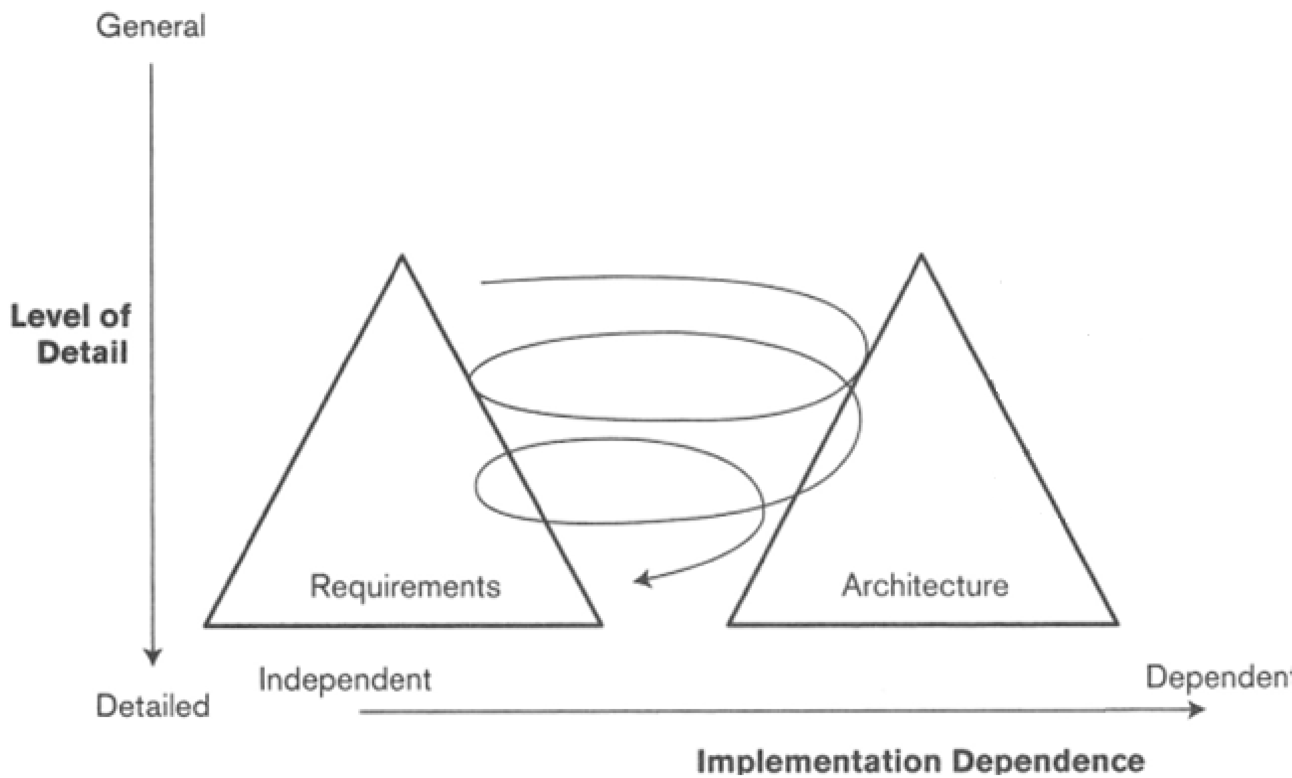
2. When a single air bubble more than 5ml volume is detected, the **system** shall stop infusion within 0.01 seconds.
3. When a single air bubble more than 5ml volume is detected, the **system** shall issue an air-embolism command.
4. When air-embolism command is true, the **system** shall stop infusion.
5. When air-embolism command is received, the **system** shall stop piston movement within 0.1 seconds


A380
A380

Your How is My What

- Systems are hierarchically organized
- Requirements vs. architectural design must be a matter of perspective
- Need better support for *N*-level decompositions for **requirements** and **architectural design**
  - Reference model support
    - How do elements "flow" between world, machine, and specification as we decompose systems?
  - Certification standard support (DO-178B/C)
    - Currently: two levels of decomposition: "high" and "low"

Twin Peaks

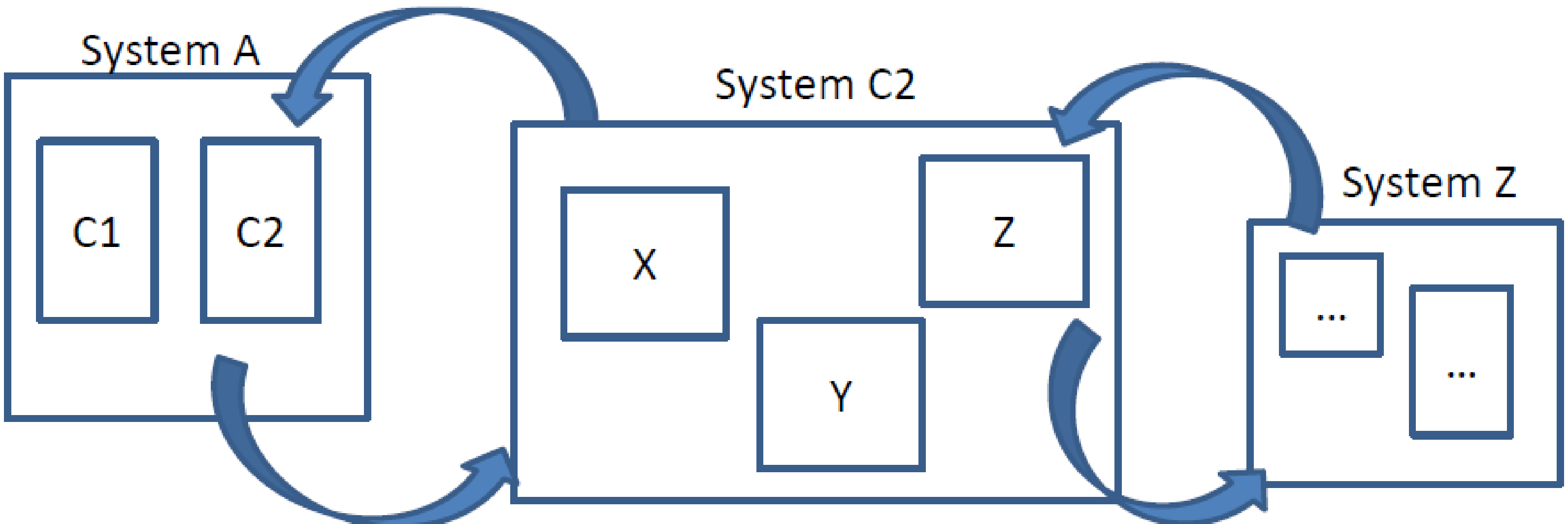

General
Level of
Detail
Detailed
Requirements
Architecture
Independent
Dependent
Implementation Dependence

Often, Architecture Comes First

- Candidate architectures from previous systems
  - Designer familiarity
  - Cost amortization
- Program families
- Certification or criticality requirements

Architectural choices often restrict set of achievable system requirements.

Flow is Bi-directional



- Determine subcomponents
- Allocate requirements to subcomponents
- Verify that subcomponent requirements establish system requirements

Requirements Validation and Verification

- Given hierarchical systems, where are the most serious problems with requirements?
  - At the component level?
  - At the top-level?
  - Somewhere in the middle?
- A hypothesis:
  - The most problematic are the layers in the middle
  - Errors in decomposing system requirements become integration problems.
- These are requirements to be both *verified* and *validated.*

STRUCTURAL PROPERTIES

Structural Properties

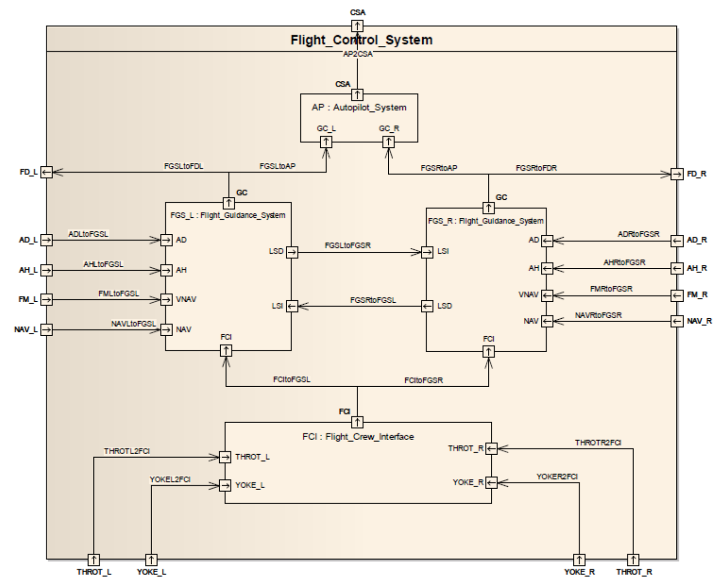
- Often, we are interested in properties about a model structure
  - Given the processor resources, is the system schedulable?
  - Is my software correctly distributed across different physical resources?
  - Are my end-to-end timing assumptions met?
- Often these involve checking the mapping between the software and the hardware.

Structural Properties

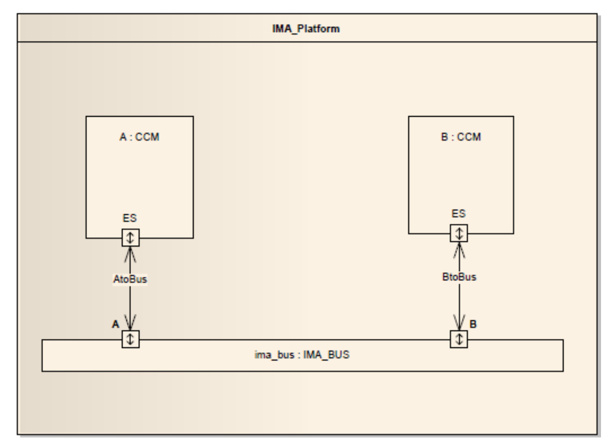
- Software + HW platform
  - Process, thread, processors, bus
- Ex: PALS vertical contract
  - PALS timing constraints on platform
  - Check AADL structural properties
- Guarantees
  - Sync logic executes at `PALS_Period`
  - `Synchronous_Communication => "One_Step_Delay"`
- Assumptions (about platform)
  - Causality constraint:

    Min(Output time) $\geq 2\varepsilon - \mu min$
  - PALS period constraint:

    Max(Output time) $\leq T - \mu max - 2\varepsilon$

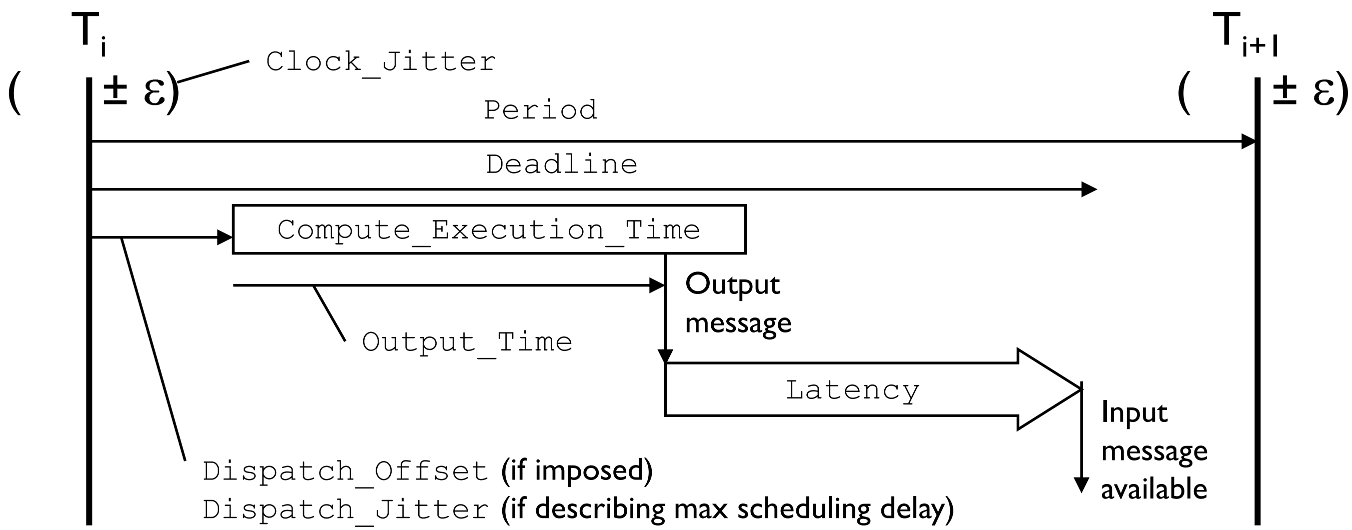


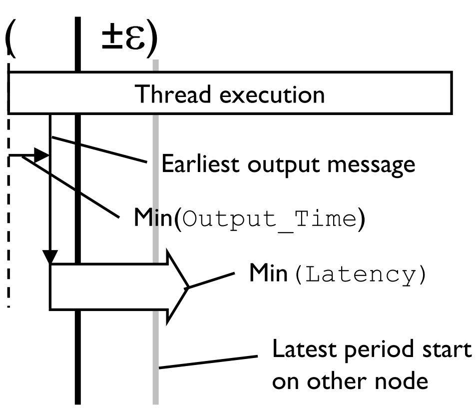
Software



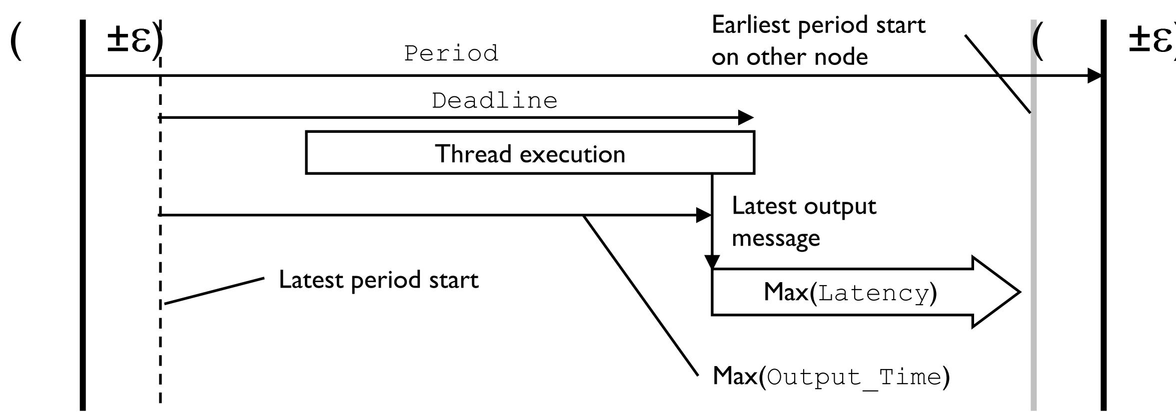
Platform

PALS assumptions in AADL





Causality Constraint



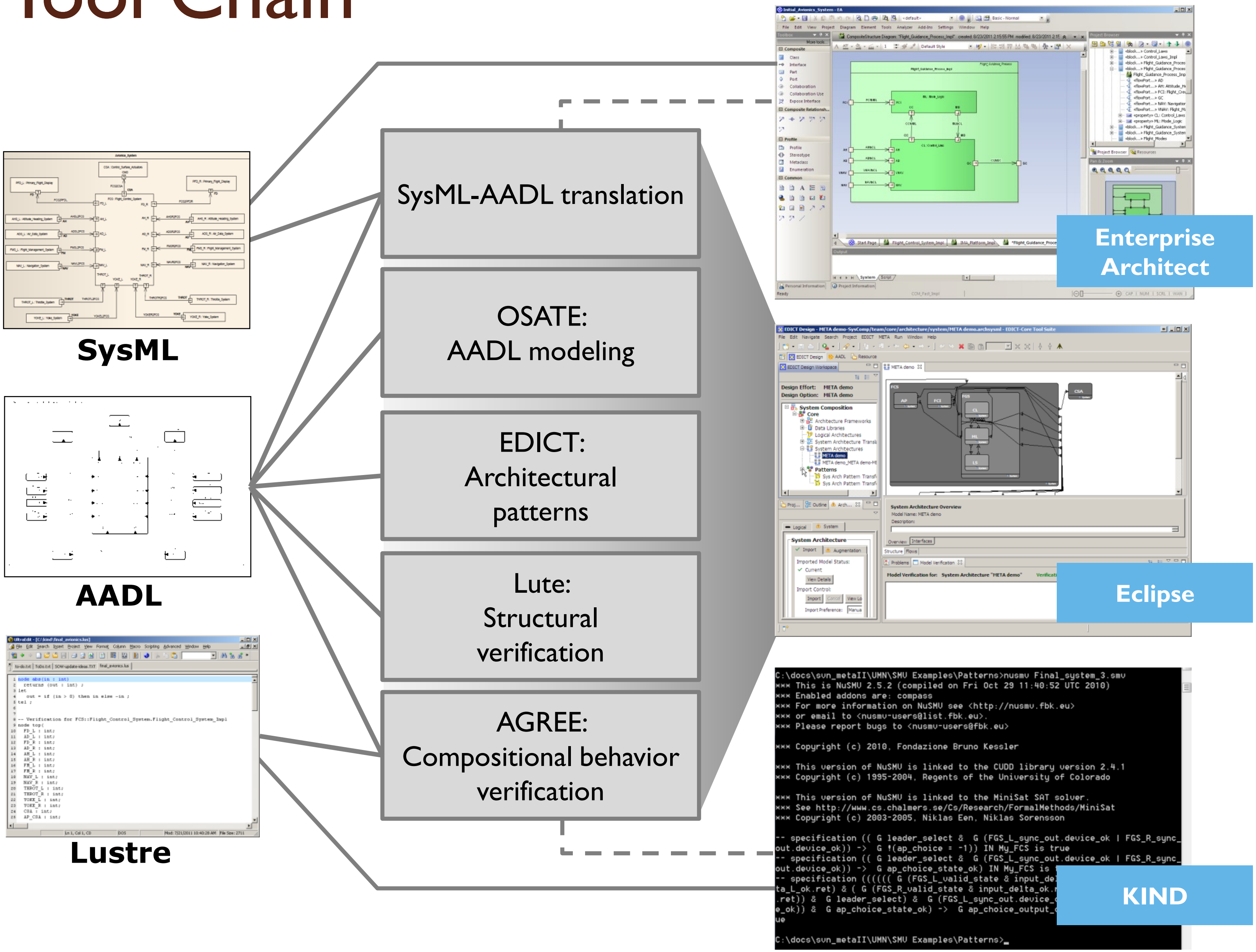
PALS Period Constraint

Structural property checks

- Contract
  - Platform model satisfies PALS assumptions
- Attached at pattern instantiation
  - Model-independent
  - Assumptions
  - Pre/post-conditions
- Lute theorems
  - Based on REAL
  - Eclipse plug-in
  - Structural properties in AADL model

```
PALS_Threads := {s in Thread_Set | Property_Exists(s,
"PALS_Properties::PALS_Id")};

PALS_Period(t) := Property(t, "PALS_Properties::PALS_Period");
PALS_Id(t) := Property(t, "PALS_Properties::PALS_Id");
PALS_Group(t) := {s in PALS_Threads | PALS_Id(t) = PALS_Id(s)};

Max_Thread_Jitter(Threads) :=
  Max({Property(p, "Clock_Jitter") for p in Processor_Set |
       Cardinal({t in Threads | Is_Bound_To(t, p)}) > 0});

Connections_Among(Set) :=
  {c in Connection_Set | Member(Owner(Source(c)), Set) and
                         Member(Owner(Destination(c)), Set)};

theorem PALS_Period_is_Period
  foreach s in PALS_Threads do
    check Property_Exists(s, "Period") and
          PALS_Period(s) = Property(s, "Period");
end;

theorem PALS_Causality
  foreach s in PALS_Threads do
    PALS_Group := PALS_Group(s);
    Clock_Jitter := Max_Thread_Jitter(PALS_Group);
    Min_Latency := Min({Lower(Property(c, "Latency")) for
                        c in Connections_Among(PALS_Group)});
    Output_Delay := {Property(t, "Output_Delay") for t in PALS_Group};
    check (if 2 * Clock_Jitter > Min_Latency then
             Min(Output_Delay) > 2 * Clock_Jitter - Min_Latency
           else
             true);
end;
```

Tool Chain


SysML
AADL
Lustre
SysML-AADL translation
OSATE:
AADL modeling
EDICT:
Architectural
patterns
Lute:
Structural
verification
AGREE:
Compositional behavior
verification
Enterprise
Architect
Eclipse
KIND
C:\docs\svn_metaII\UMN\SMU Examples\Patterns>nusmv Final_system_3.smv
*** This is NuSMV 2.5.2 (compiled on Fri Oct 29 11:40:52 UTC 2010)
*** Enabled addons are: compass
*** For more information on NuSMV see <http://nusmv.fbk.eu>
*** or email to <nusmv-users@list.fbk.eu>.
*** Please report bugs to <nusmv-users@fbk.eu>
*** Copyright (c) 2010, Fondazione Bruno Kessler
*** This version of NuSMV is linked to the CUDD library version 2.4.1
*** Copyright (c) 1995-2004, Regents of the University of Colorado
*** This version of NuSMV is linked to the MiniSat SAT solver.
*** See http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat
*** Copyright (c) 2003-2005, Niklas Een, Niklas Sorensson
C:\docs\svn_metaII\UMN\SMU Examples\Patterns>

Research Challenges

Structural and Behavioral Properties

Structural (Non-functional) Properties: Analyze conformance, optimization properties for hardware resources and model structure.

Assertion: My system is schedulable using Rate Monotonic Scheduling.

```
Theorem RMA
foreach e in Processor_Set do
  Proc_Set(e) := { x in Process_Set |
      Is_Bound_To(x, e) } ;
  Threads := { x in Thread_Set |
    Is_Subcomponent_Of(x, Proc_Set) }

  check (sum
    (get_property_value (Threads
        "RTOS_properties::Utilization")) <=
    (Cardinal (Threads) *
       (2 ** (1 / Cardinal (Threads))) -1 ) ) ;
End RMA ;
```

Checkable with Lute

Behavioral (functional) Properties: Analyze system behavior. Behavioral properties may use structural properties.

Assertion: If a "Pump Stop" command is received, then within 1 second the measured flow rate shall be zero.

```
PSL_contract

property no_flow_after_stop :
  after
    (not (infusion_control_in.Pump_On))
  (exists
     flow_rate_detector_out.Rate = 0
   within
     STEPS_PER_SECOND *1)  ;

  assert (no_flow_after_stop) ;
end PSL_contract;
```

Checkable with AGREE

Are these the “right” logics?

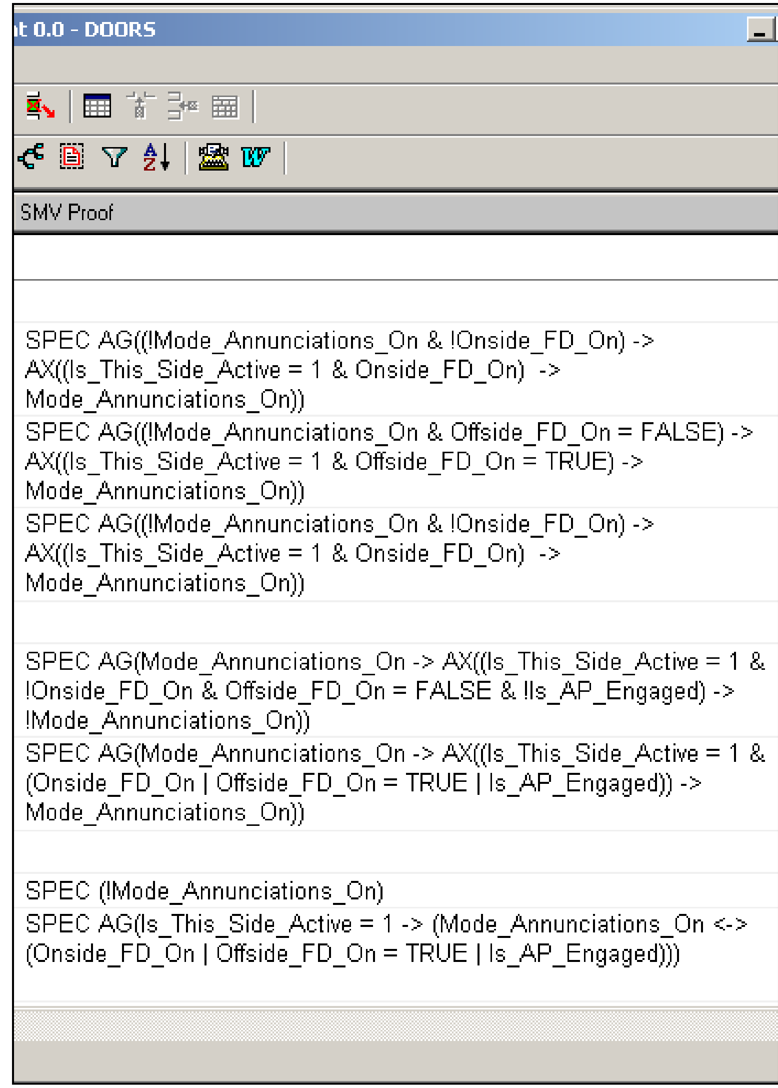
- Simpler logics have benefits
  - Primary benefit: much simpler to analyze
  - AADL error annex is (mostly) propositional
    - Makes analysis simpler
    - Supports useful categorization of errors
  - Datalog-style logics support “timeless” analysis
    - The Lute checker is essentially a datalog interpreter
- More complicated logics are necessary for certain properties
  - Richer types (e.g., algebraic types for XML messages)
  - Quantification

Dealing with Time

- Pure synchrony or asynchrony
- Uniform discrete time
  - Choose fixed time quantum between steps
  - This quantum need not be the same between layers
  - Adjust process behavior and requirements with clocks.
- Non-uniform discrete time
  - Calendar/Timeout automata advance system to next interesting instant
- Dense time

Scaling

- What do you do when systems and subcomponents have hundreds of requirements?
  - FGS mode logic: 280 requirements
  - DWM: >600 requirements
- Need to create automated slicing techniques for predicates rather than code.
  - Perhaps this will be in the form of counterexample-guided refinement



Assigning blame

- Counterexamples are often hard to understand for big models
- It is much worse (in my experience) for property-based models
- Given a counterexample, can you automatically assign blame to one or more subcomponents?
- Given a "blamed" component, can you automatically open the black box to strengthen the component guarantee?

Signal	Step... 0	1	2	3	4	5
AD_L.pitch.val	-0.91	-1.83	-2.74	-3.65	-4.35	-4.39
AD_L.pitch.valid	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
AD_R.pitch.val	0.83	-0.09	-1.00	-1.91	-2.83	-3.74
AD_R.pitch.valid	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE
AP.CSA.csa_pitch_delta	0.00	0.13	0.09	0.26	0.74	-4.26
AP.GC_L.cmds.pitch_delta	0.00	-4.91	-4.65	-4.57	-4.74	-4.35
AP.GC_L.mds.active	TRUE	FALSE	FALSE	FALSE	FALSE	TRUE
AP.GC_R.cmds.pitch_delta	0.00	0.83	-4.43	-4.48	4.91	4.83
AP.GC_R.mds.active	TRUE	TRUE	FALSE	FALSE	FALSE	FALSE
Assumptions for AP	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Assumptions for FCI	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Assumptions for FGS_L	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
Assumptions for FGS_R	TRUE	TRUE	TRUE	TRUE	TRUE	TRUE
FGS_L.GC.cmds.pitch_delta	-4.91	-4.65	-4.57	-4.74	-4.35	0.09
FGS_L.GC.mds.active	FALSE	FALSE	FALSE	FALSE	TRUE	FALSE
FGS_L.LSO.leader	2	2	3	2	1	3
FGS_L.LSO.valid	FALSE	TRUE	FALSE	TRUE	TRUE	FALSE
FGS_R.GC.cmds.pitch_delta	0.83	-4.43	-4.48	4.91	4.83	3.91
FGS_R.GC.mds.active	TRUE	FALSE	FALSE	FALSE	FALSE	FALSE
FGS_R.LSO.leader	0	0	1	0	1	1
FGS_R.LSO.valid	TRUE	FALSE	TRUE	FALSE	FALSE	TRUE
leader_pitch_delta	0.00	0.83	0.83	0.83	0.83	-4.35
System level guarantees	TRUE	TRUE	TRUE	TRUE	TRUE	FALSE

"Argument Engineering"

- Disparate kinds of evidence throughout the system
  - Probabilistic
  - Resource
  - Structural properties of model
  - Behavioral properties of model
- How do we tie these things together?
- Evidence graph, similar to proof graph in PVS
  - Shows evidential obligations that have not been discharged
- SRI is working on this: *Evidential Tool Bus (ETB)*
  - This seems to be a reasonable approach for tying tool results together
  - Declarative (like make or ant), but more powerful (uses Datalog)

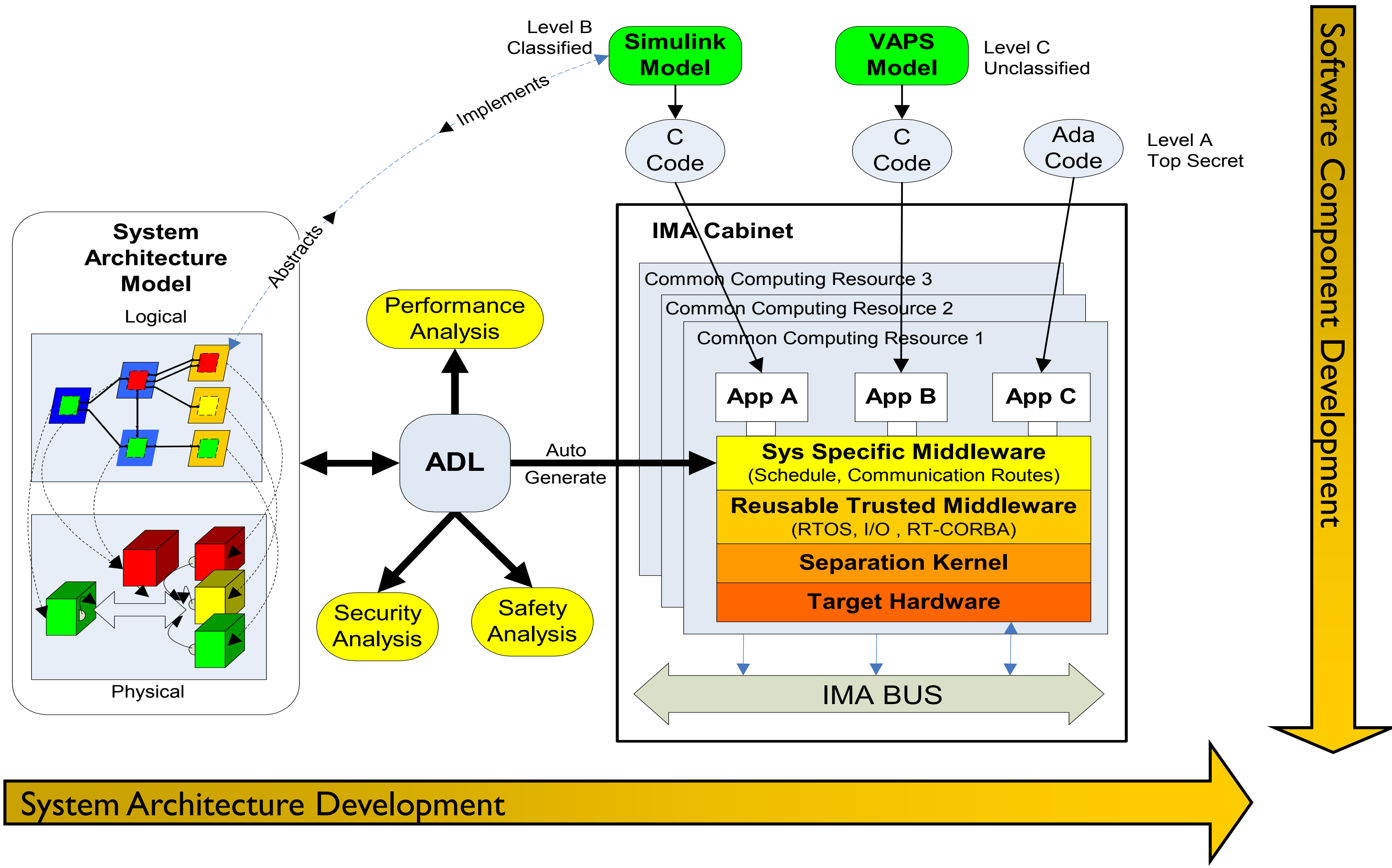
Integration with AADL

- Type representations
  - Currently we use "homebrew" property set for typing information
  - AADL data modeling annex?
- Inheritance and Refinement
  - **`Extends`** from same AADL class
  - **`Implements`** from different AADL class
  - Contracts should preserve behavioral subtyping
    - Weaken assumptions
    - Strengthen guarantees
  - Some subtleties:
    - For existential properties over traces (CTL), this refinement is generally **unsound**.
    - Probably only want to support **universal properties** (like LTL)
- Binding of logical system to physical system
  - Contracts are built on many assumptions involving physical system involving resources. Currently these are not addressed in the temporal logic, but externally
  - How do we represent physical failures in logical contracts?

Conclusions

- AADL is very nice for designing systems
  - Good way to describe hardware and software
  - Lots of built-in analysis capabilities
- Allows new system engineering approaches
  - Iteration between reqs and design
  - Specification and use of architectural patterns
- Looking at behavioral and structural analysis
  - Still *lots* of work to do!
  - ..but already can do some interesting analysis with tools
  - Sits in a nice intersection between requirements engineering and formal methods
  - Starting to apply this to large UAV models for security properties in the SMACCM project

System Architectural Modeling & Analysis



Thank you!


Ευχαριστώ
Merci
Grazie
Díky
Gracias
Vielen Dank
Obrigado!
شكراً
धन्यवाद
Teşekkürler