Compositional Verification of Architectural Models

High Confidence Software & Systems Conference

9 May 2012
Darren Cofer

Rockwell Collins

Outline

- What is META?
- Project vision
- Tool environment
- Technologies
  - System-level modeling and translation
  - Complexity-reducing architectural patterns
  - Compositional verification
- Next steps

Rockwell Collins

Team

- Rockwell Collins / Advanced Technology Center
  - Darren Cofer, Steven Miller, Andrew Gacek
  - System modeling & analysis, tooling, integration
- UIUC
  - Lui Sha
  - Design pattern development
- University of MN
  - Michael Whalen
  - Pattern verification, compositional analysis
- WWTG
  - Chris Walter, Brian LaValley
  - Pattern implementation & analysis tools

ILLINOIS
UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

UNIVERSITY OF MINNESOTA

WW TECHNOLOGY GROUP
The Dependability Solution Provider ™

What is META?

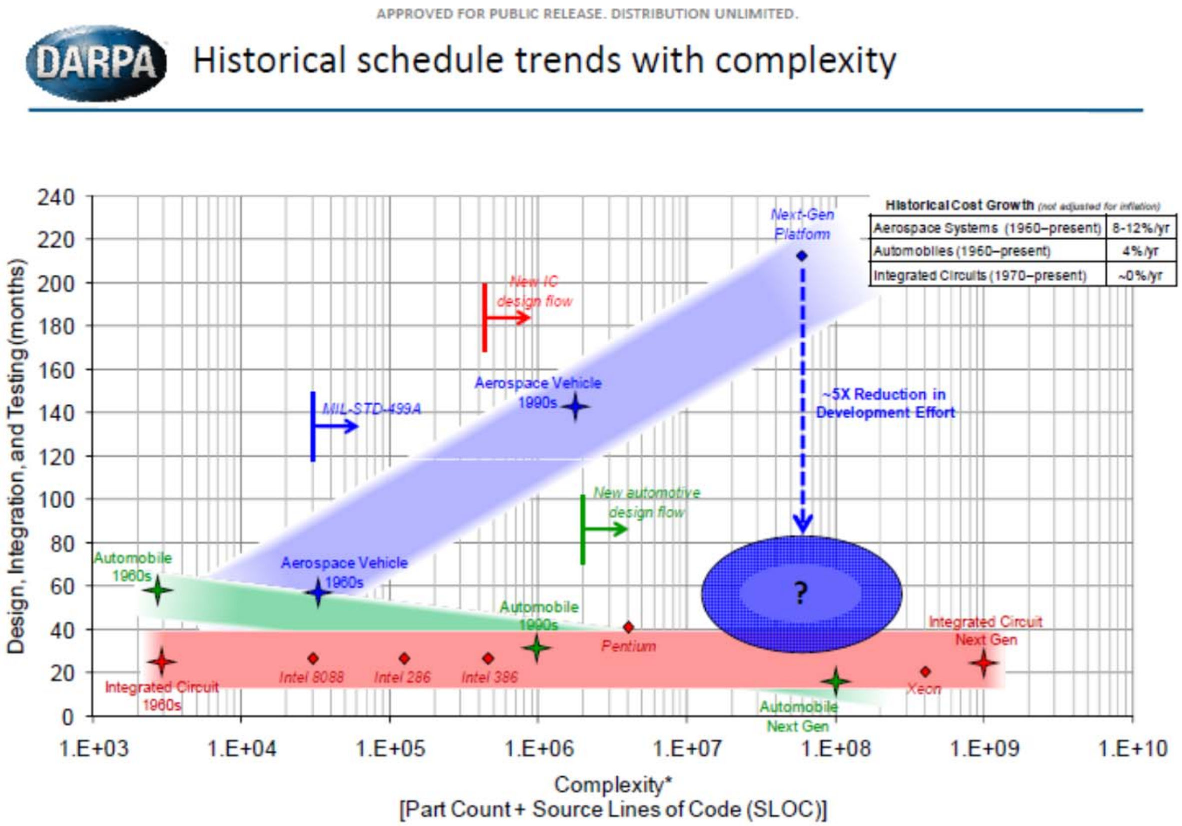
- Devise, implement, and demonstrate a radically different approach to the design, integration/manufacturing, and verification of defense systems/vehicles

- Enhance designer's ability to manage system complexity

- "Foundry-style" model of manufacturing

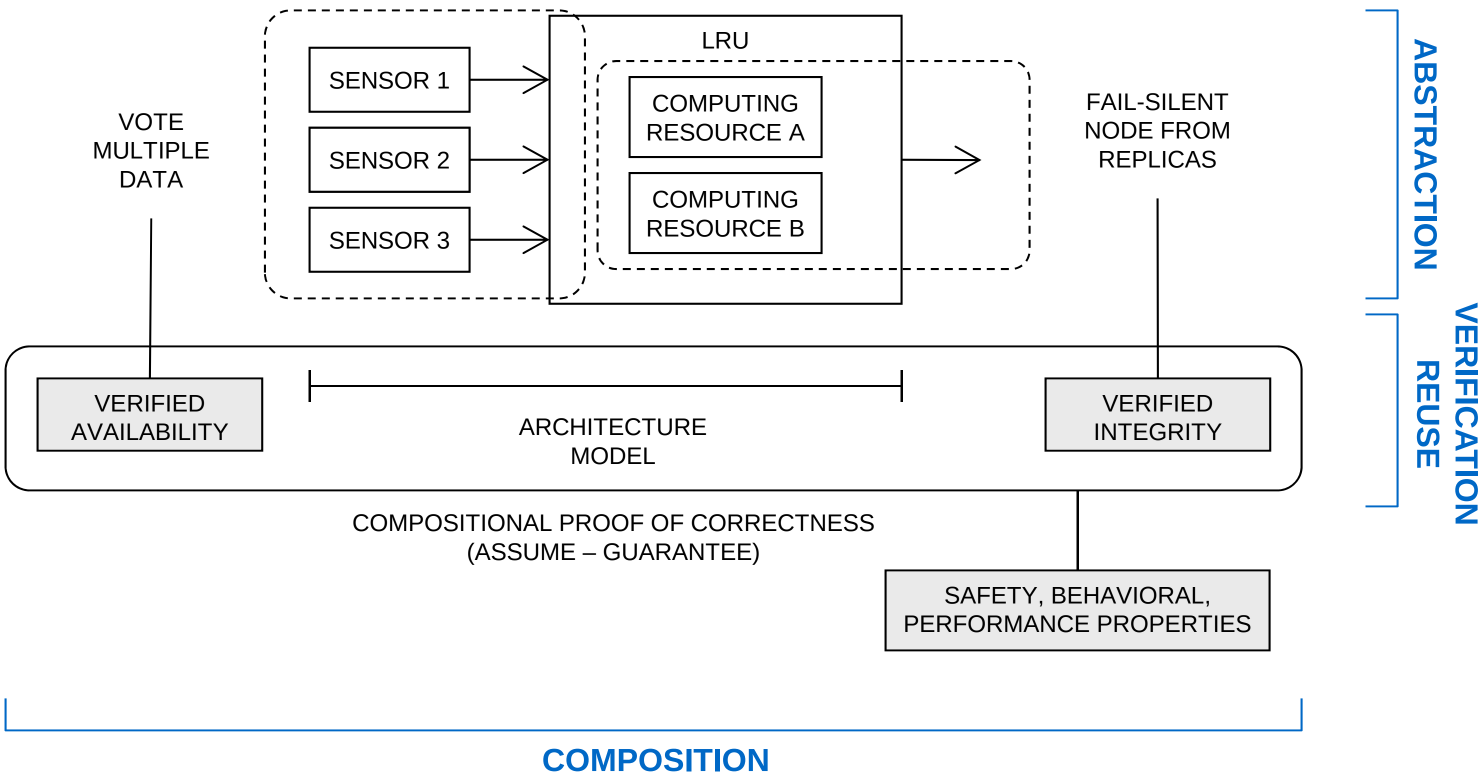
- Five technical areas

  1. Metrics of complexity
  2. Metrics of adaptability
  3. Meta-language for system design
  4. Design flow & tools
  5. Verification flow & tools

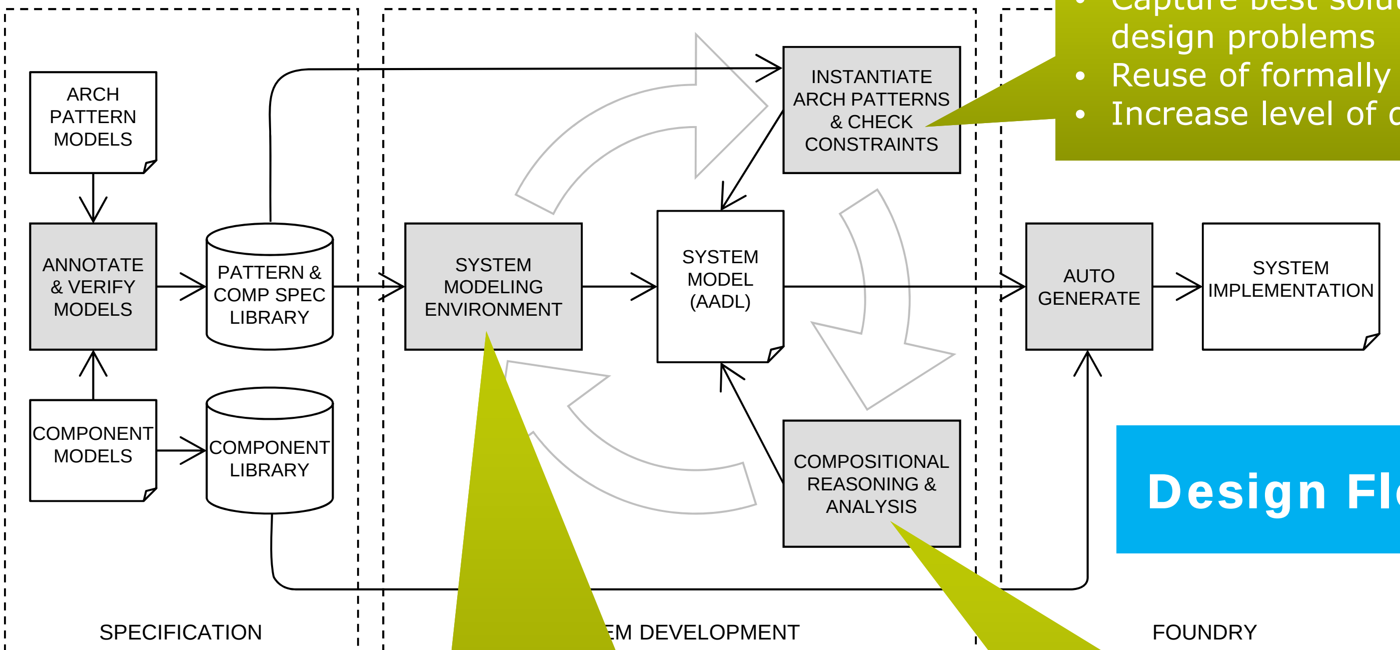


Vision

- Improve effectiveness and scalability of system design and verification through pre-verified design patterns and compositional reasoning



Approach



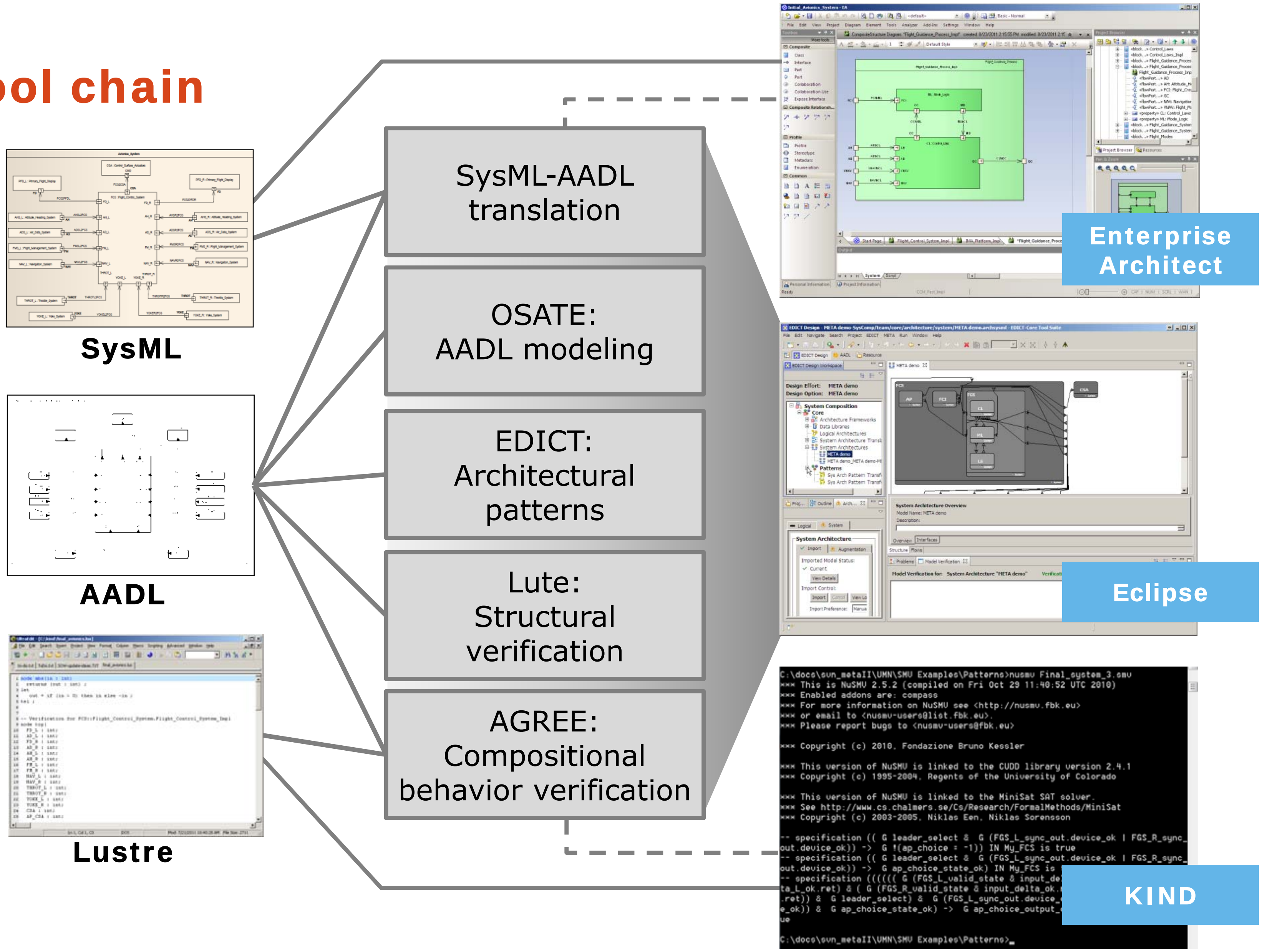
Complexity-reducing design patterns 2
- Capture best solutions to architectural design problems
- Reuse of formally verified solutions
- Increase level of design abstraction

System architecture modeling 1
- Apply formal specification and analysis tools to system-level design
- Separate component specification and implementation
- Automated model translation

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

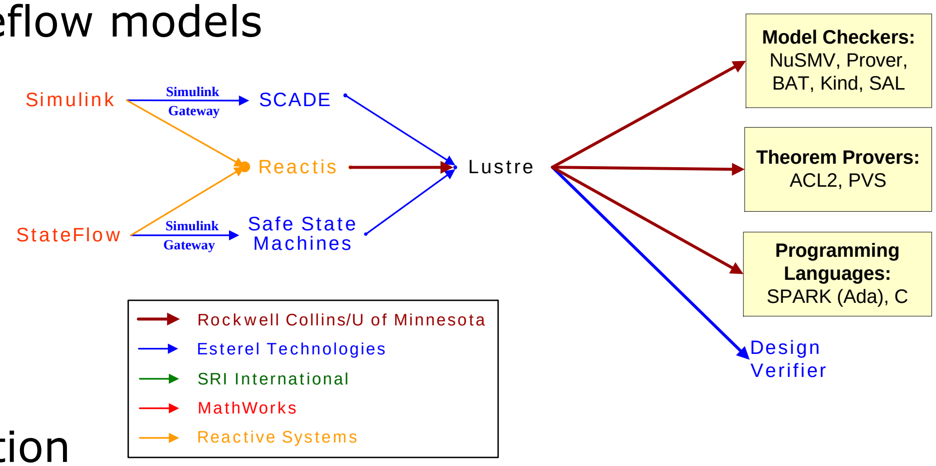
Rockwell Collins

Tool chain



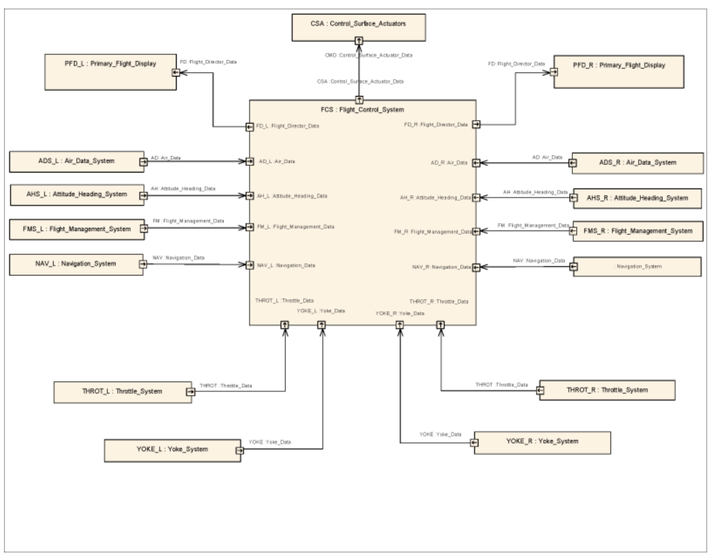
System architecture modeling

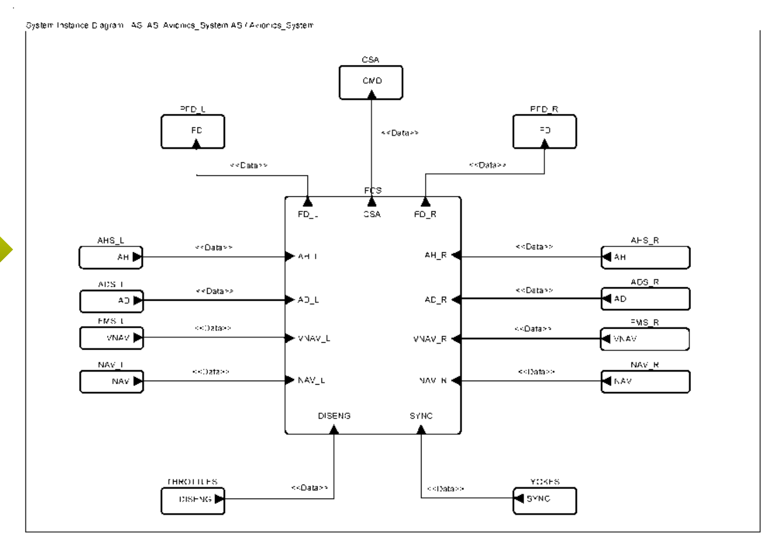
- We have been very successful at applying formal methods to software components produced in model-based development environments
  - Gryphon translation framework
  - Verification of Simulink/Stateflow models
- Objective
  - Leverage this knowledge and apply formal methods to the system design process
- Issues
  - Modeling language and tools
  - Different models of computation
  - Scalability



System modeling and translation

- AADL is a good fit and provides sufficiently formal notation
  - Available tools do not provide stable graphical environment
  - OSATE: open source, Eclipse-based
- SysML is being adopted by many organizations for system design
  - But has no formal semantics
  - No common textual representation across tools
- Solution: Eclipse plugin that provides bidirectional translation
  - Based on Enterprise Architect SysML tool used by Rockwell Collins
  - Define block stereotypes that correspond to AADL objects



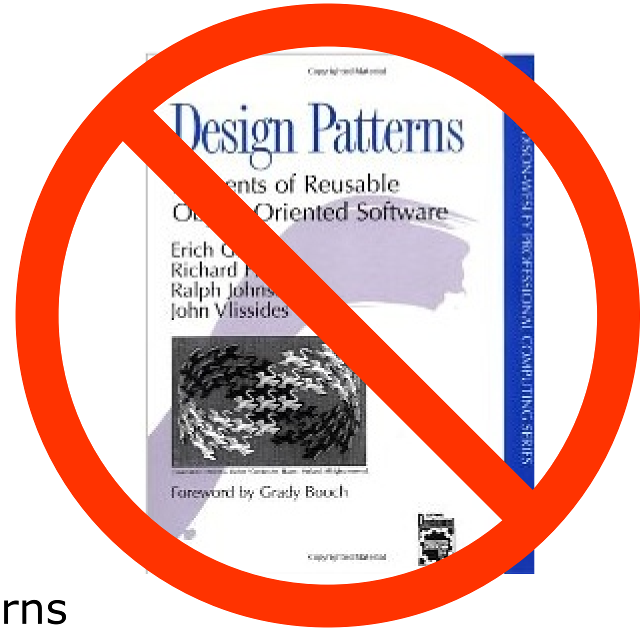


Scale and composition

- Architectural model does not capture implementation details
  - Component descriptions, interfaces, interconnections
- Assume/guarantee contracts provide the information needed from other modeling domains to reason about system-level properties
  - Guarantees correspond to the component requirements
  - Assumptions correspond to the environmental constraints that were used in proving the component requirements
  - Contract specifies precisely the information that is needed to reason about the component's interaction with other parts of the system
  - Supports hierarchical decomposition of verification process
- Contract can be applied to both components and design patterns
  - Mechanism for *verification reuse*
  - More about this later

Complexity-Reducing Architectural Design Patterns

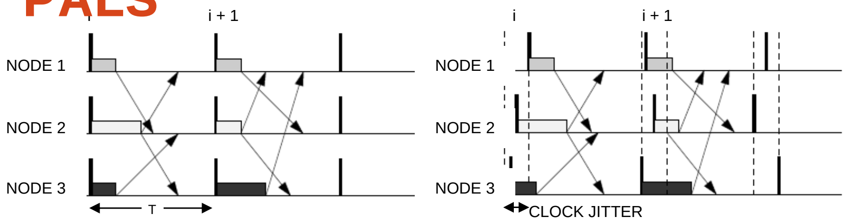
2



- 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

Rockwell Collins

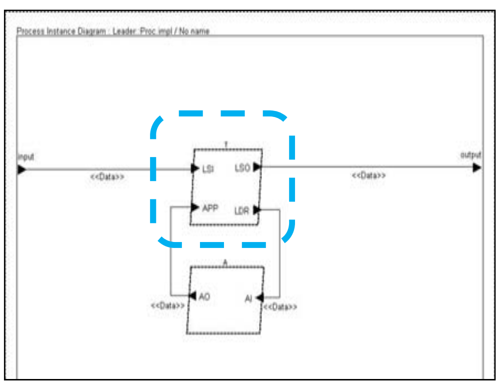
PALS



SYNCHRONOUS NETWORK ASYNCHRONOUS BOUNDED DELAY NETWORK WITH PALS

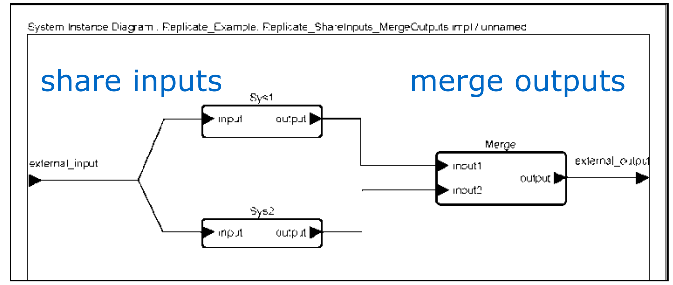
- Provide virtual synchrony for parts of async system
- Assumptions
  - Structural preconditions on system model (bounded jitter, computation, message delivery…)
  - Required data connections exist
- Guarantees
  - Sync logic executes with period T
  - Data from step *i* consumed in step *i* +1

Leader Selection



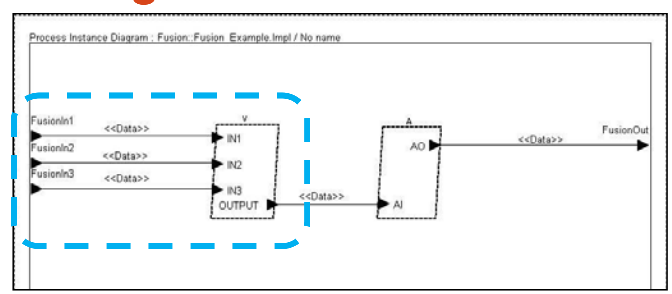
- Create leader for group of nodes
- Assumptions
  - Nodes communicate synchronously
  - At least one non-failed node
- Guarantees
  - All non-failed nodes agree on leader
  - If leader fails, new leader in next step
  - Non-failed node remains leader

Replication



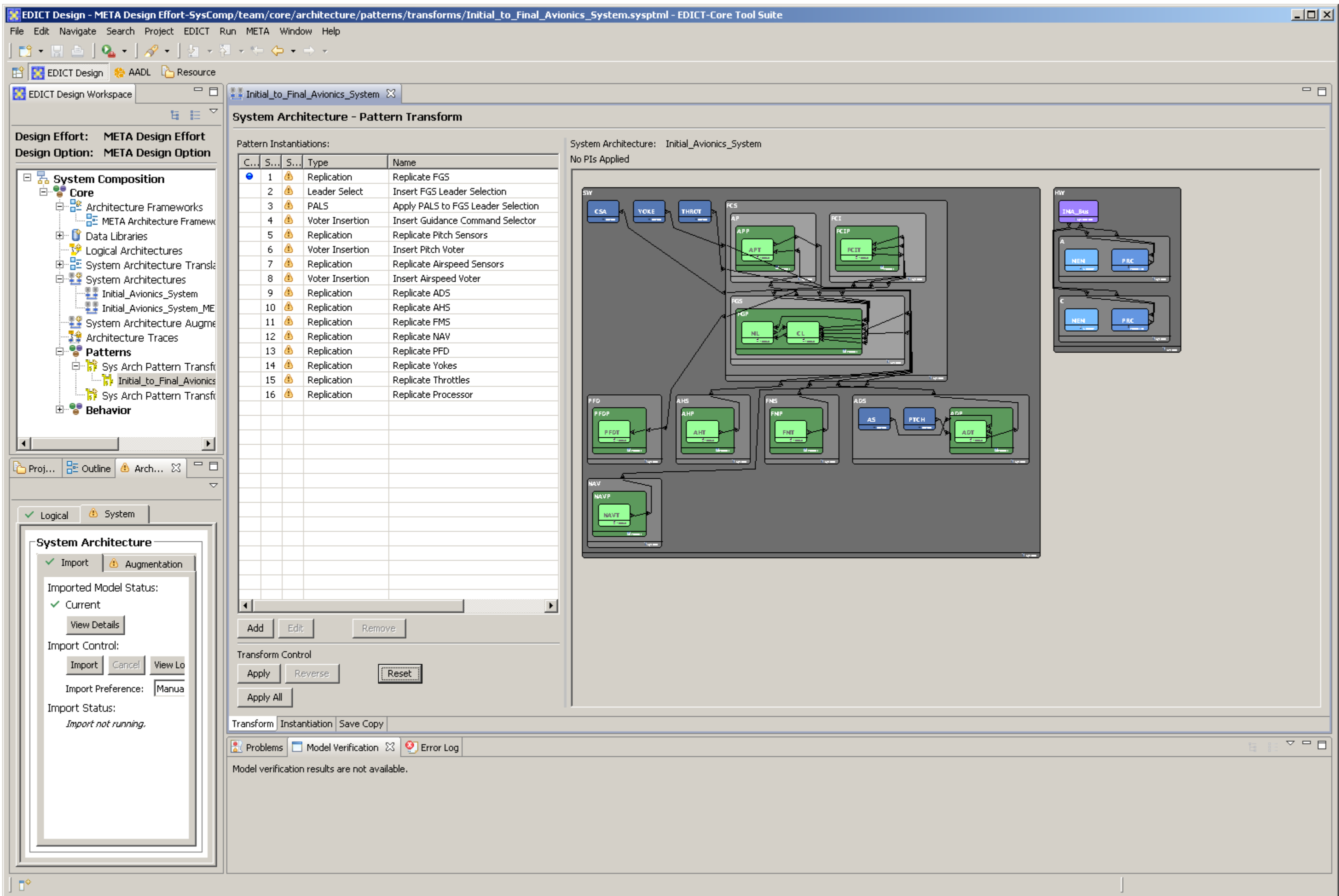
- Create identical copies of portions of the system
- Assumptions
  - Replicas hosted on platform HW with independent failure modes
- Guarantees
  - One or replicas will operate normally in the event of a single fault

Voting / Fusion

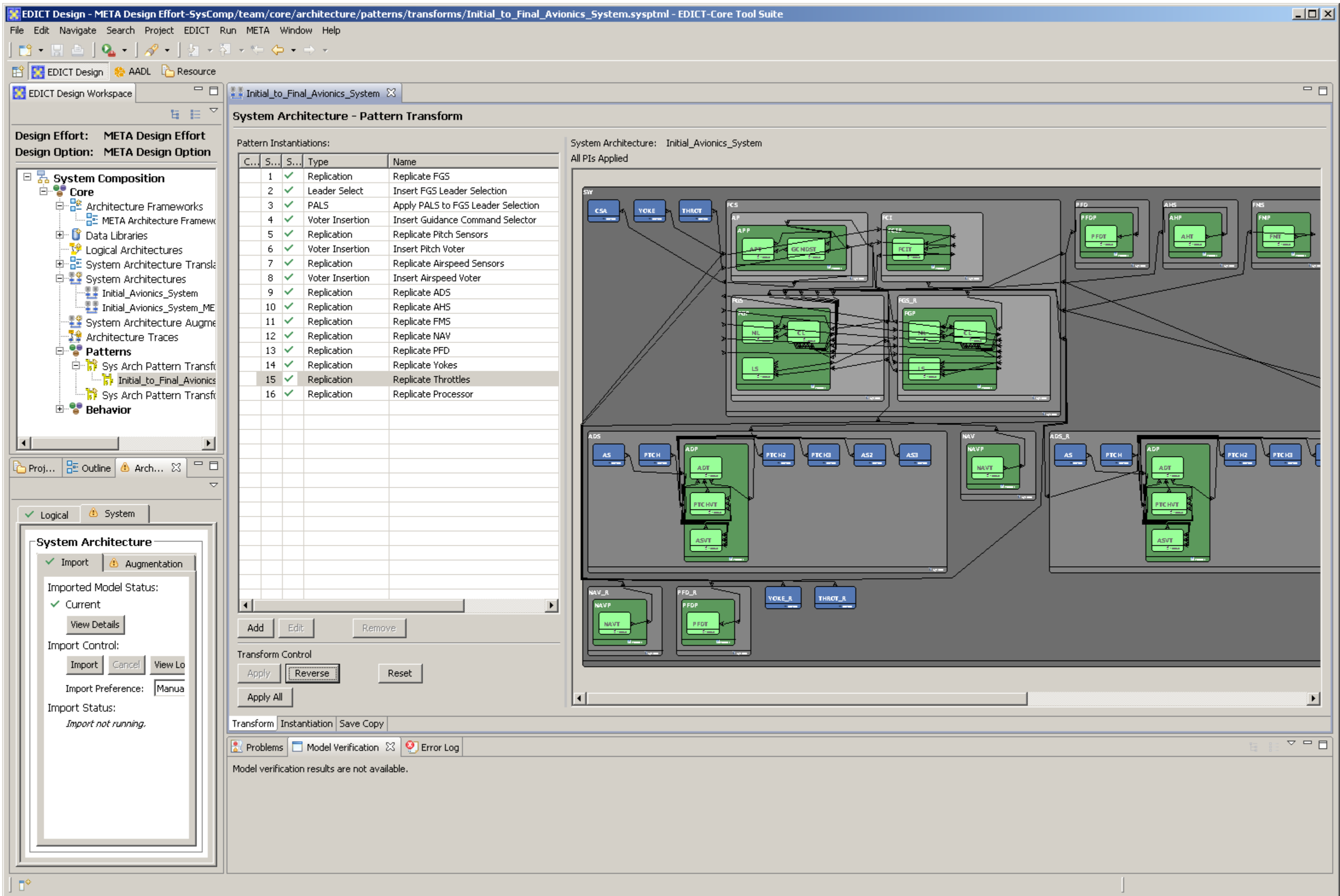


- Combine several component interfaces
- Assumptions
  - Interfaces terminate at same destination component
  - Interfaces have same data type
- Guarantees
  - Varies with component type
  - Agreement, mid-value select, output select, average

Initial Avionics System

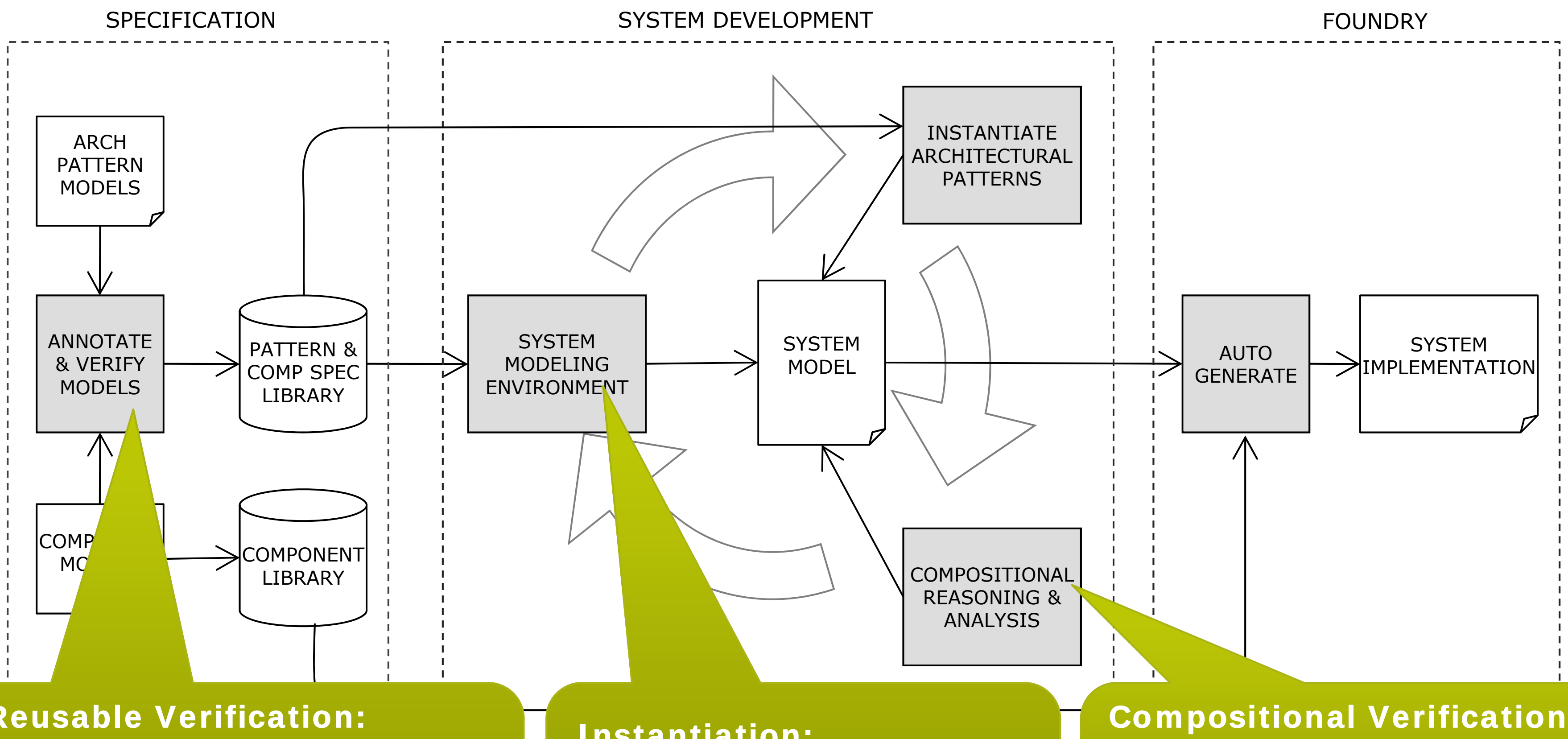


Final Avionics System (after pattern transformations)



System verification

3



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

Categories of system properties

- Structural/static
  - Properties of the transformed model
  - Pattern assumptions, post-conditions
  - Specified and checked using Lute
  - *PALS period constraint*
    `Deadline < PALS_Period - Max_Latency - 2*Clock_Jitter`
- Behavioral/dynamic
  - Pattern and component interactions
  - Specified in PSL, verified by AGREE using model checking
  - *Failed node will not be leader in next step*
    `G(!device_ok[j] -> X(leader[i] != j)) ;`
- Resource allocation
  - RT schedulability, memory allocation, bandwidth allocation
  - ASIIST tool (UIUC/RC)
  - *Threads can be scheduled to meet their deadlines*
- Probabilistic
  - Failure analysis of system
  - Behavior and failure rates described using AADL error annex
  - PRISM/PRISMATIC (SIFT/RC)
  - *P(all sensors failed)* $< 10^{-9}$

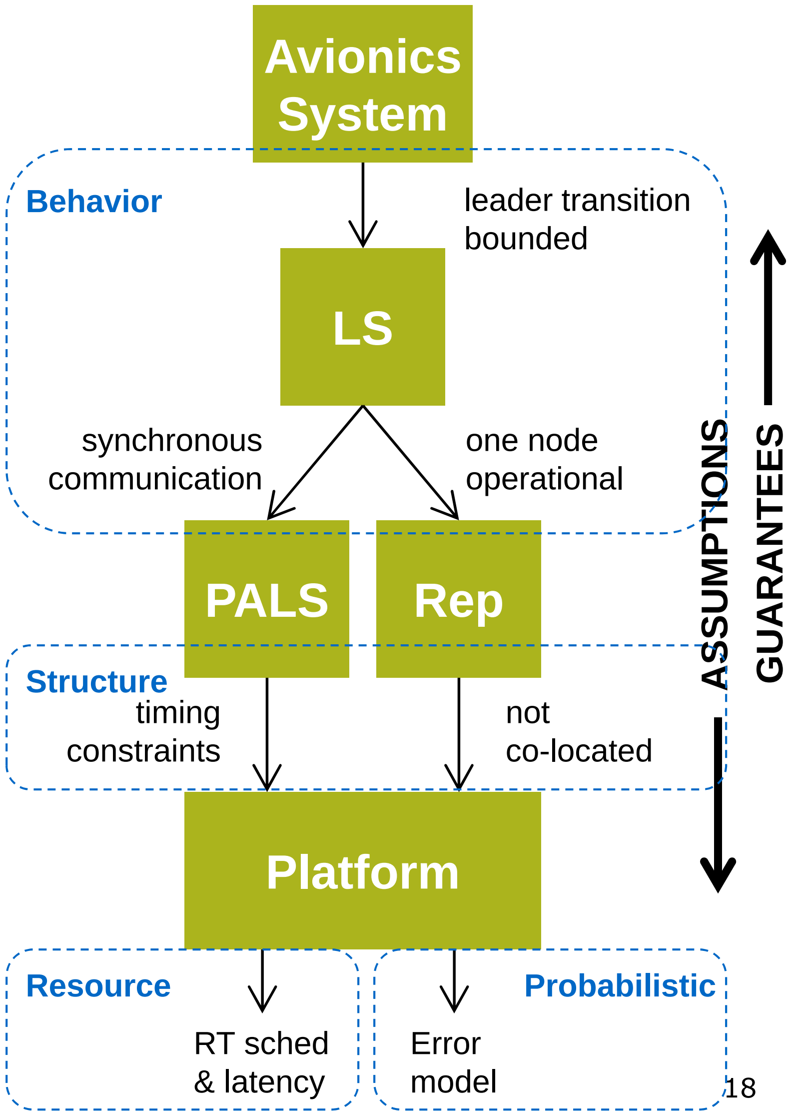
Contracts between patterns and components

- 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"

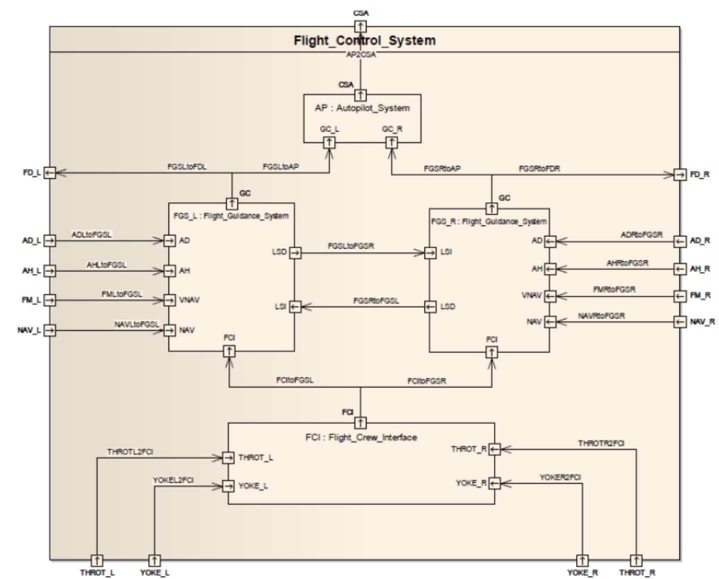


Structural / static properties

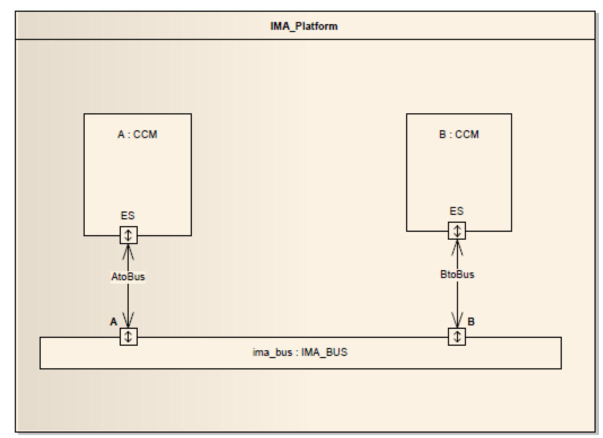
- Software + HW platform
  - Process, thread, processors, bus
- Ex: PALS vertical contract
  - PALS timing constraints on platform
  - Check AADL structural properties
- Guarantees
  - Sync logic executes at `PALS_Period`
  - `Synchronous_Communication => "One_Step_Delay"`
- Assumptions (about platform)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

    Min(Output time) ≥ 2ε – μmin
  - PALS period constraint:

    Max(Output time) ≤ T - μmax - 2ε



Software



Platform

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;
```

Compositional behavior verification

- Given
  - Assumptions for system
  - Assumptions/Guarantees for components (A, P)
- Prove
  - System guarantees (requirements)
- New analysis plug-in (AGREE)
  - Automatic translation of model structure, contracts, and verification conditions
  - Verify via k-induction model checker (KIND - Tinelli @ Univ. of Iowa)

Contract compliance:
$G(H(A) \Rightarrow P)$

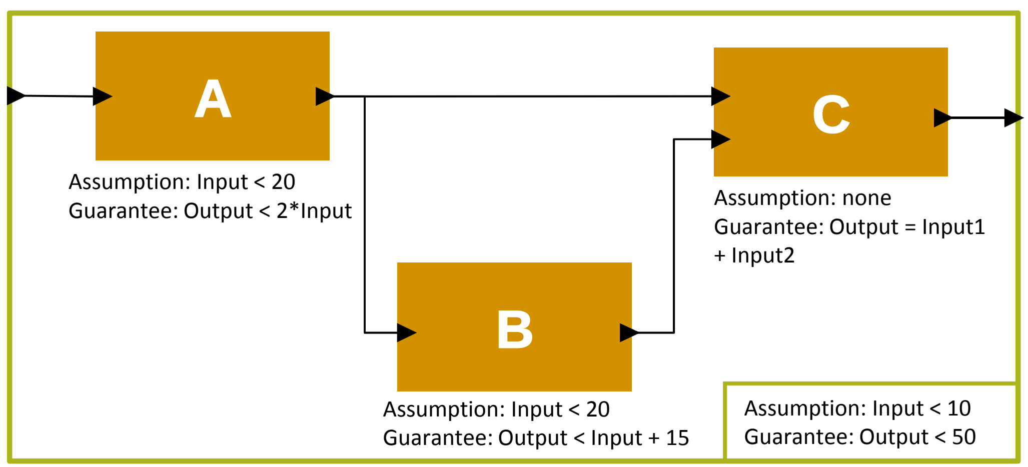
Example (to prove)

$A_S \rightarrow A_A$

$A_S \wedge P_A \rightarrow A_B$

$A_S \wedge P_A \wedge P_B \rightarrow A_C$

$A_S \wedge P_A \wedge P_B \wedge P_C \rightarrow P_S$



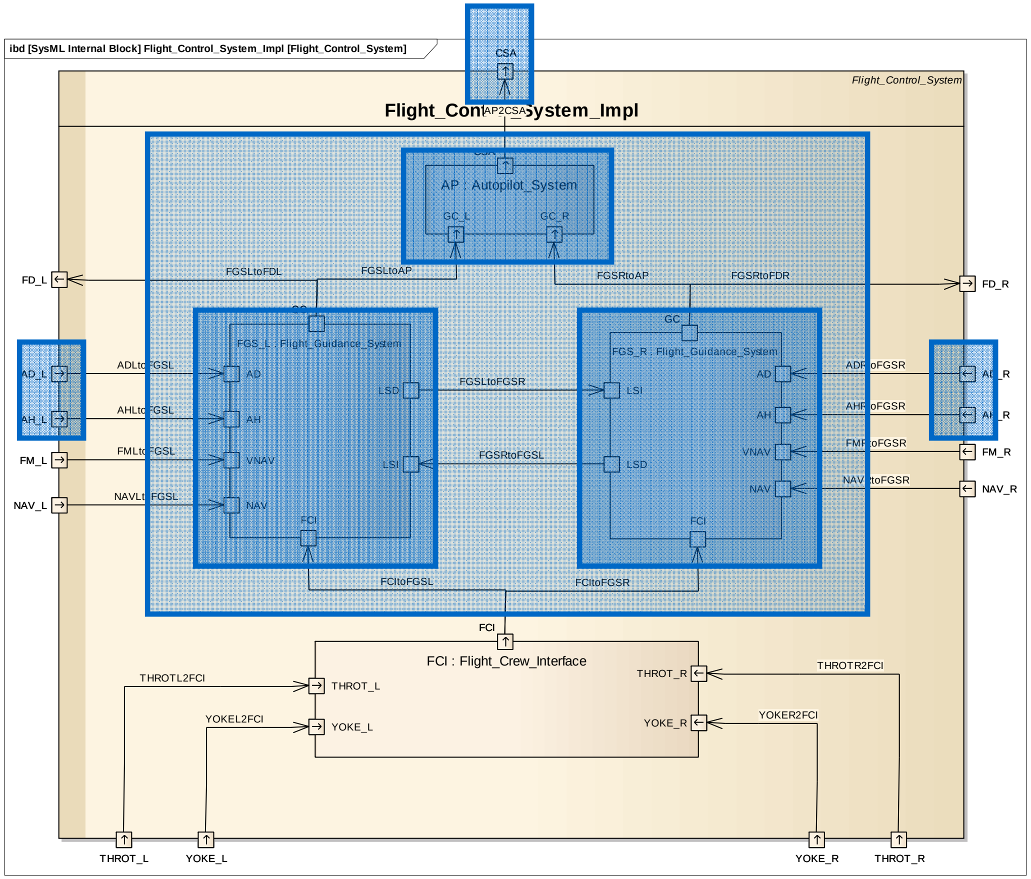
Contract specification in AADL

- Derived from Property Specification Language (PSL) formalism
  - IEEE standard
  - In wide use for hardware verification
- Assume / Guarantee style specification
  - Assumptions: “Under these conditions”
  - Guarantees: “…the system will do X”
- Local definitions can be created to simplify properties
- For now, this is an AADL string property

```
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 ;
```

Compositional reasoning for FCS

- 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 ;
```

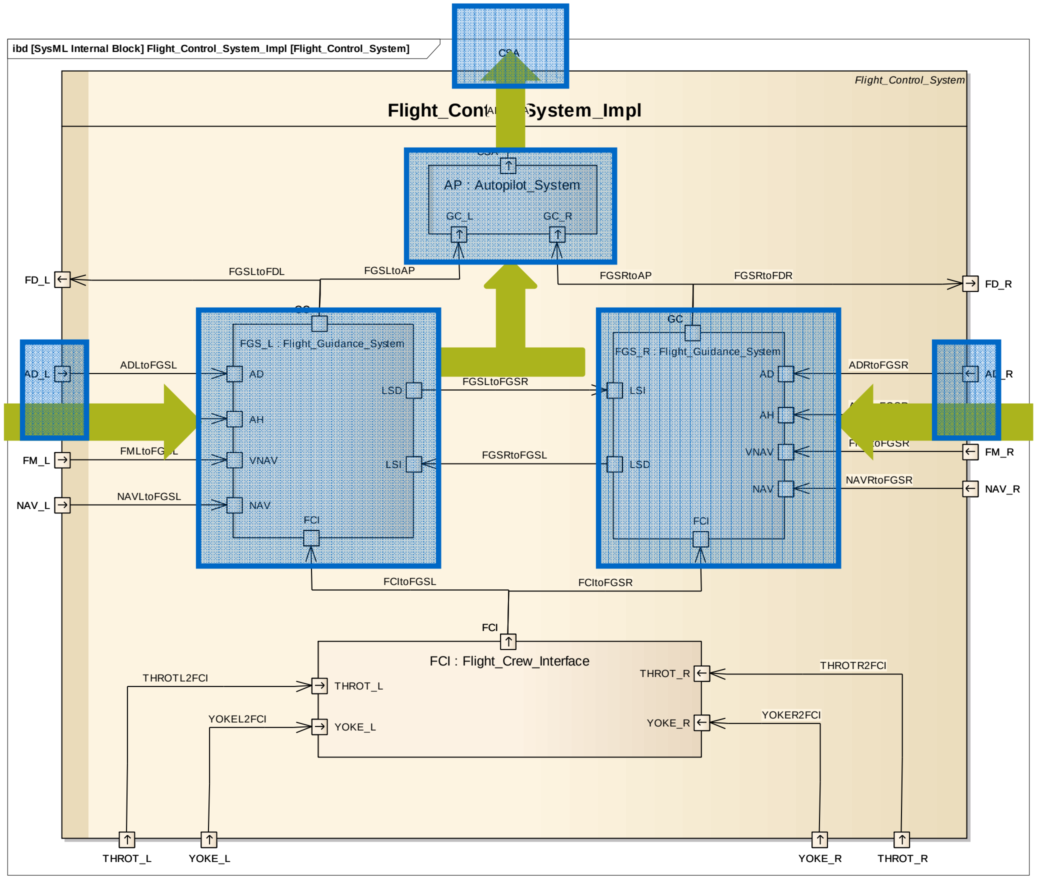
Compositional Reasoning and Patterns

- Guarantees provided by pattern are encoded as **facts**
- Attached at pattern instantiation
  - Model-independent
  - Assumptions
  - Pre/post-conditions
- Describe relationships between several components
  - In this example, the *Leader* and *Valid* fields for the left and right FGSs.

```
pattern_instance Leader_Select_1 :

  -- sync single-step delay between elements
  assume single_step_delay_comm(FGS_L, FGS_R);
  assume single_step_delay_comm(FGS_R, FGS_L);

 -- All non-failed nodes agree on who is the leader
 leader_agreement:
    assert (FGS_L.LSO.Valid and FGS_R.LSO.Valid) =>
      FGS_L.LSO.Leader = FGS_R.LSO.Leader;

 -- If a node fails, leadership is transferred to a non-failed node
 leader_transfer_1:
   assert (prev(not(FGS_L.LSO.Valid), false) =>
    (FGS_R.LSO.Valid =>
      FGS_R.LSO.Leader != Get_Property(FGS_L, Leader_Select_ID)));

 leader_transfer_2:
   assert prev(not(FGS_R.LSO.Valid), false) =>
     (FGS_L.LSO.Valid =>
        FGS_L.LSO.Leader != Get_Property(FGS_R, Leader_Select_ID));

  -- If any non-failed nodes exist, one of them will be the leader
  leader_existence:
   assert (prev(FGS_L.LSO.Valid or FGS_R.LSO.Valid, false)) =>
     (( FGS_L.LSO.Valid => (FGS_L.LSO.Leader >= 1 and FGS_L.LSO.Leader <= 2)) and
      ( FGS_R.LSO.Valid => (FGS_R.LSO.Leader >= 1 and FGS_R.LSO.Leader <= 2)));

  -- If the leader does not fail, it shall remain the leader.
  leader_persistence_1: assert
    (prev(FGS_L.LSO.Valid and
     FGS_L.LSO.Leader = Get_Property(FGS_L, Leader_Select_ID), false)) =>
       (FGS_L.LSO.Valid =>
          FGS_L.LSO.Leader = Get_Property(FGS_L, Leader_Select_ID));

  leader_persistence_2: assert
   (prev(FGS_R.LSO.Valid and
    FGS_R.LSO.Leader = Get_Property(FGS_R, Leader_Select_ID), false)) =>
      (FGS_R.LSO.Valid =>
         FGS_R.LSO.Leader = Get_Property(FGS_R, Leader_Select_ID));
end pattern_instance Leader_Select_1 ;
```

Proof Process

- Order of data flow through system components is computed by AGREE
  - {System inputs} → {FGS_L, FGS_R}
  - {FGS_L, FGS_R} → {AP}
  - {AP} → {System outputs}
- Based on flow, we establish four proof obligations
  1. System assumptions → FGS_L assumptions
  2. System assumptions → FGS_R assumptions
  3. System assumptions + FGS_L guarantees + FGS_R guarantees → AP assumptions
  4. 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 (usually a time delay)

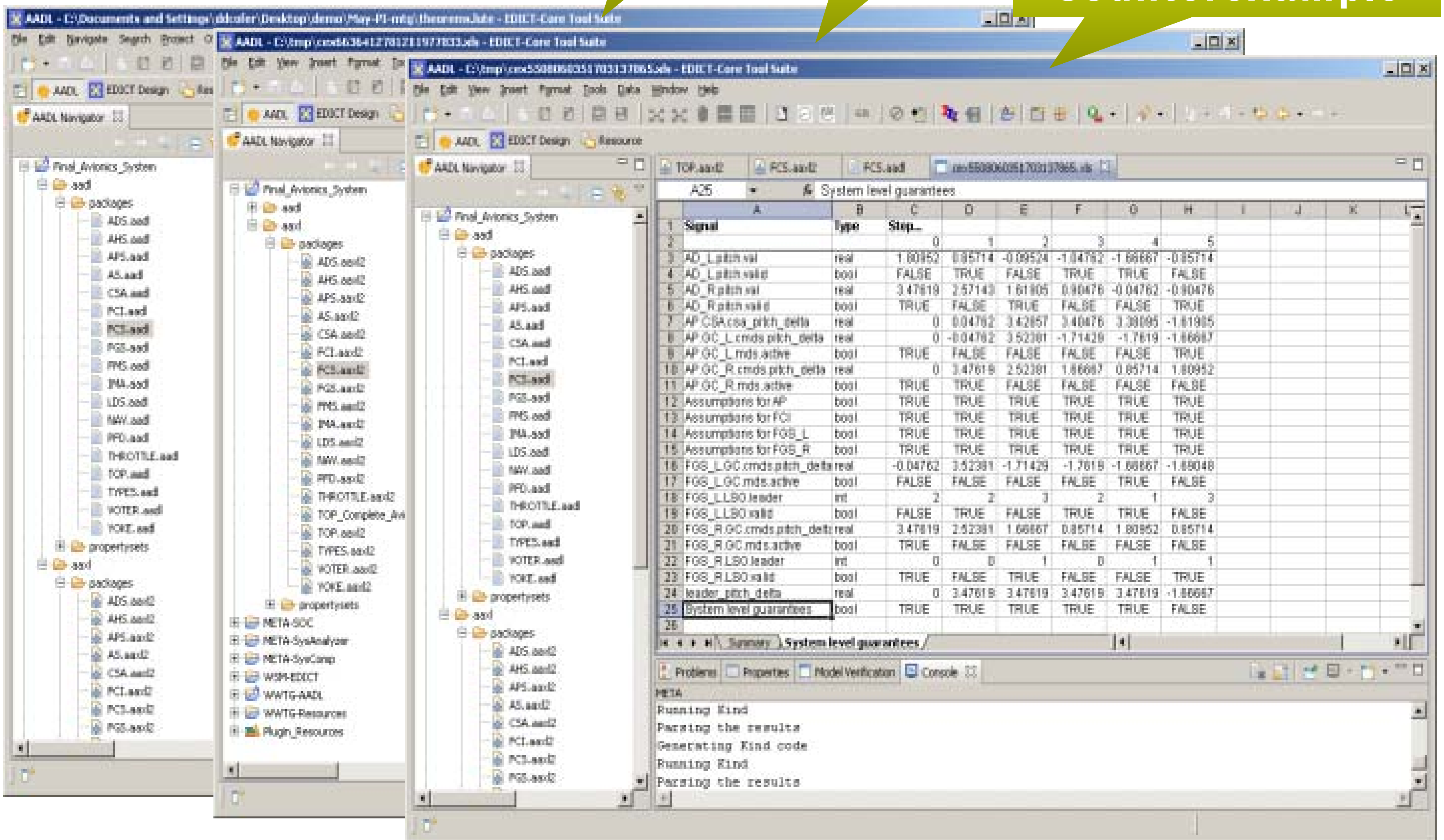


Verification tools

Lute

AGREE

Counterexample



Next steps

- Extend compositional verification to more complex models of computation
  - Multiple rates, delays, asynchrony
- Incorporate additional design patterns in library
  - Especially fault tolerance patterns with existing verification artifacts
- Improved annotation of contracts in architecture models
  - AADL annex? Alternate representations (e.g., sequence diagrams?)
- More general mechanism for composing evidence from multiple sources
  - Evidence graph, assurance case

Download

- AADL Tools wiki
  - https://wiki.sei.cmu.edu/aadl/index.php/RC_META