# OM-AADL Modeling

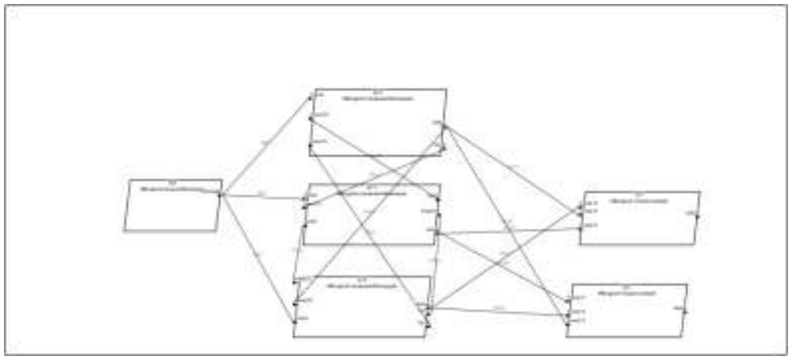
AFFIRM TAG-UP 2/11/15

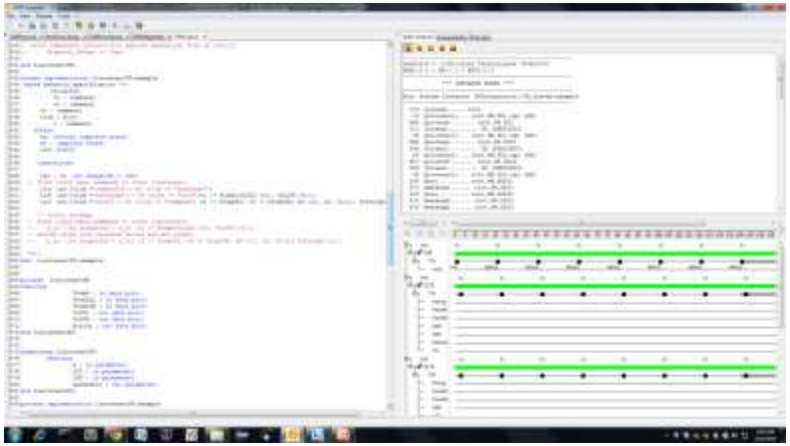
# OM-Modeling

- Motivation : Hand code AADL model of OM to contrast with Tower generated model
  - Understand capture of synchronous system composition
    - Contrast to initial Tower event driven model
  - Introduce error-models
  - Investigate SAL frame-work for Integrated formal model capture
    - Platform Dispatch & Local Thread Behavior
    - Communication Behavior
    - Error /Fault manifestations
  - Generate feedback for DSL synthesis path
- Status :
  - Initial Model Created using Standard AADL incorporating Behavior Annex and Error Annex Annotations

# OM-Model Description

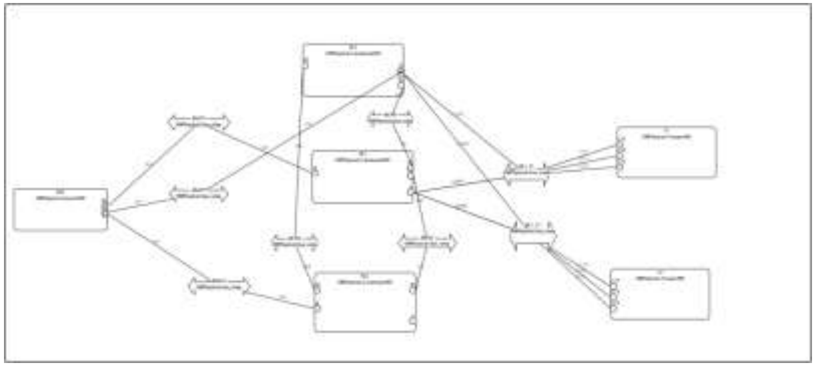
- ARP nominally assumes faults are introduced by hardware. So our model comprised to parts
  - Logical model representing software
    - Supplemented with Behavioral Annotations





To Check legality if platform and behavioral semantics we executed them using a behavioral simulation. Still debugging

  - Platform model representing hardware
    - Supplemented with Error Annex annotations
    - Evaluated two types of error annotations
      - Declarative port propagation
      - Binding propagation



We may integrate with different bus connectivity and error models to explore influence of faults

# Error Model Annotations

AADL Inspector ( C:/Users/E094226/Desktop/2015/Projects/NASA-AFFIRM/svn/trunk/casestudies/OM/HandCode-AADL/OM-Example/packages/OMErrorLibrary.aadl )

File View Wizards Tools ?

OMPhysical | HonErrorLibrary | OMErrorLibrary | OMIntegration | OMLogical

```
package OMErrorLibrary
public
        with ErrorLibrary;

annex EMV2 {**
        error types
                Failure : type ;
                FailStop : type extends Failure;
                FailSymmentric  : type extends Failure;
                FailAsymmetic: type extends Failure;
                end types;


error behavior OMErrorBehavior
        use types OMErrorLibrary;
        events
        Failure : error event {Failure};
        states
        Operational: initial state;
        Failed: state {Failure};
        transitions
                FirstFailure : Operational -[Failure]-> Failed;

        -- with reference to the transofrmation discussion
        -- I am stil a little uncertain to what we need to do here
        -- but we can stop at one fault
        WhenFailedFailures :Failed -[Failure]-> Failed;
        end behavior;


        **};
```

Initially using very simple error mode
that is not linked to generic error library types
(See later discussion)

# Error Propagations

Verbose declarative error propagations

```
        g: in propagation  {ItemOmission,SymmetricValue,AsymmetricValue};
        ltr: out propagation {ItemOmission,SymmetricValue,AsymmetricValue};
        ltl: out propagation {ItemOmission,SymmetricValue,AsymmetricValue};
        o :  out propagation {ItemOmission,SymmetricValue,AsymmetricValue};
        end propagations;

component error behavior
    propagations

   -- there may be a better way to encode this
       Failed{FailStop}-[]->ltr{ItemOmission};
       Failed{FailStop}-[]->ltl{ItemOmission};
       Failed{FailStop}-[]->o{ItemOmission};
       Failed{FailSymmentric}-[]->ltr{SymmetricValue};
       Failed{FailSymmentric}-[]->ltl{SymmetricValue};
       Failed{FailSymmentric}-[]->o{SymmetricValue};
       Failed{FailAsymmetic}-[]->ltr{ASymmetricValue};
       Failed{FailAsymmetic}-[]->ltl{ASymmetricValue};
       Failed{FailAsymmetic}-[]->o{ASymmetricValue};

            -- when operational we propate what comes in
       operational -[g{ItemOmission}]->ltr{ItemOmission};
       operational -[g{ItemOmission}]->ltl{ItemOmission};
       operational -[g{SymmetricValue}]->ltr{SymmetricValue};
       operational -[g{SymmetricValue}]->ltl{SymmetricValue};
       operational -[g{ASymmetricValue}]->ltr{SymmetricValue};
       operational -[g{ASymmetricValue}]->ltl{SymmetricValue};
   end component;
        **};
end LieutenantHW.verbose;
```

Bindings approach

```
system implementation LieutenantHW.concise
-- (Pierre 27.01.15) adding a Processor subcomponent
subcomponents
  cpu : processor cpu {Scheduling_Protocol => (RMS); };

annex EMV2 {**
        use types OMErrorLibrary,ErrorLibrary;
  use behavior OMErrorLibrary::OMErrorBehavior;
        error propagations
                bindings: out propagation {ItemOmission, SymmetricValue,ASymmetricValue};
        flows
                f1 : error source bindings{ItemOmission, SymmetricValue,ASymmetricValue};
        end propagations;

component error behavior
    propagations
                Failed{FailStop}-[]->bindings{ItemOmission};
                Failed{FailSymmentric}-[]->bindings{SymmetricValue};
                Failed{FailAsymmetic}-[]->bindings{AsymmetricValue};
   end component;
```

# Behavioral Annotations

```
thread implementation LieutenantTM.example
 annex behavior_specification {**
         variables
          v1 : command;
          v2 : command;
      v3 : command;
      tick : slot;
          o : command;
    states
      si: initial complete state;
      s0 : complete state;
      inc: state;

     transitions

      tsi : si -[on dispatch]-> inc;
  -- first round send commands to other lieutenants
      tic: inc-[tick ="command"]-> s0 {tick := "exchange"};
      tie: inc-[tick ="exchange"]-> s0 {tick := "vote";v1 := FromG;ToLTL!(v1); ToLTR!(v1)};
      tiv: inc-[tick ="vote"]-> s0 {tick := "command"; v2 := FromLTL; v3 := FromLTR; mv!(v1, v2, v3,o); ToTroops!(o)};
```

Behavior expressed using global schedule

Start up issues not covered in initial model

```
data command
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("null","attack", "retreat", "hold position");
    Data_Model::Representation => ("00", "01", "10", "11");
end command;

data slot
  properties
    Data_Model::Data_Representation => Enum;
    Data_Model::Enumerators => ("command","exchange", "vote");
    Data_Model::Representation => ("00", "01", "10", "11");
end slot;
```

Subprogram

```
subprogram maj_vote
-- quick draft just looks for two agreeing values assuming 1 fault
-- we may want to re-code this to more closely reflect SAL
features
  x : in parameter command;
  y : in parameter command;
  z : in parameter command;
  o : out parameter command;

 annex behavior_specification {**
         states
      s: initial final state;
        transitions
         t0 : s -[x = y]-> s { o:= x};
         t1 : s -[x = z]-> s { o:= x};
         t3 : s -[y = z]-> s { o:= y};
         t4 : s -[y != z and y != x ]-> s { o:= "null"};

  **};

end maj_vote;
```

# Thoughts Related to Model Construction -I

- Synchronous execution model of AADL appears to make the logic error consideration simpler
  - Much easier to encode omission error model once the periodic rates capture an expected arrival time
- Formal model needs to integrate platform dispatch behavior and thread local behavior
  - We working hybrid calendar based abstraction in SAL.
- The declarative error propagation “encodes” assumptions of the error propagation
- How is this kept consistent with real behavior?
  - This is interesting for logical error propagations
    - Consider TTP/Membership.
  - How and where is the captured?
    - To complex to capture using the EA
  - Can we deduce the abstract error propagation from the fusion of the error model and behavioral model using the AFFIRM approach?
  - Do we need error propagation or do we need only local fault models?
- **Hunch** *Bindings maybe a simpler mechanism to support error and behavior integration.*
  - Logical model is “overridden” by fault propagation at binding intersection point

Since AFFIRM is a targeting synthesis from levels above the implementation model, what can we learn from the AADL model capture.

- To drive consistency - Is the path to the formal model through the implementation model or orthogonal to it

# Thoughts Related to Model Construction -II

- In learning the background of the AADL model we looked at alternative representations
  - COMPASS SLIM
    - Integrated Error/Behavior model
  - MILS_AADL
    - Interesting refinement using known libraries for Distributed MILS implementation
  - BLESS-BA
    - Clean semantics for behavioral annotation
      - Frozen inputs and outputs during execute states
    - Much easier to compose with dispatch behavior
      - Chosen as exploratory basis of calendar-based SAL abstraction (in Progress)
  - BLESS bindings to platform model
    - Bless requires dispatch mechanisms that are not yet part of core standard **
      - Timed Port (See attached)
    - Exploratory thoughts of modeling BRAIN flooding protocol and TTP/ START-UP indicated that this type of construct may be required
    - What can we learn here?
      - Do we have a set of synthesis platform primitives related to behavior and platform level support?

* *We have already closed these loops with the AADL working groups

# Modeling Faults

- In order to integrate errors  in to fuse behavior and error we need a cleaner semantics and error primitives to allow for cross annex fusion
  - Working to redefined error library

    – [Value] {Synatic Error, Semantic Error}
    – [Time] {OrdinalTimingError, CardinalTimingError}
    – [Persistence] {PermanentError, TransientError, RepetitiveError}
    – [Symmetry] {SymmetricError, AsymmetricError}
    – [Detectability] {Locally Detectable Not-locally Detectable Error }

    Using our simpler ontology higher-level fault-error manifestations are formed by taking the product of each if the contributing themes. For example

    • StuckValue => PermanentError * ValueError
    • SubtleValueError => NonSelfDetectable * ValueError;

    A babbling idiot (assuming incomprehensible noise transmissions) would be

    • Babbling Idiot => Permanent * * Symmetric * * SyntacticError

    Note for such products to be clean, it is essential that the themes and base types are suitably orthogonal.

- Type products used to compose behaviors
  - This may enable a simpler mapping. Each error type would introduce a behavior into the logical model
  - Can we end up with Error Insertions scripts, that allow for non-deterministic  fault injections
  - Do we need equivalence class mapping to keep state-space manageable ?
  - Does AFFIRM calculate the equivalence classes?

# What does this all mean to AFFIRM?

- Working at AADL model is equivalent to Level 5 of our original abstraction hierarchy
  - Our main intent was to work bottom up to get an handle on the following
    - Execution platform primitives
    - Integration of fault , architecture and nominal behavior models
  - Raising to the next level of abstraction may allow for abstraction properties to explored
    - Consider bus abstraction for communication action
    - How are communication properties communicated ?
    - How is the abstraction reflected in the DSL?
  - Many other issues encountered during this work
    - Consider start-up to nominal transition
    - Often such issues are separated in formal analysis?
    - In AFFIRM we need to integrate these aspects into the DSL?
    - What primitives of behavior will be need to achieve this?
  - We may also learn from DMILS Synthesis approach (still processing)
    - DMILS syntheses using characterized known libraries
      - E.g. TTEthernet
    - AFFRIM may need to surpass this
      - In place of known libraries we may create proof obligations for the lower levels of refinement