Lessons and Experiences Learned Applying Model Based Engineering to Safety Critical FPGA Designs

A US-DOE Funded Project under the NEET Program

11th International Workshop on the Application of FPGAs in NPPs

Dr. Carl Elks, Co-PI, Virginia Commonwealth University

Dr. Ashraf Tantawy, Research Professor, Virginia Commonwealth University

Matt Gibson, Program PI, EPRI

Rick Hite, Smitha Gautham, Chris Deloglos, Athira Jayakumar, Shawkat Khairullah - Virginia Commonwealth University

Jason Moore, MathWorks

Andrew Nack, Paragon Inc.

Oct. 8th 2018

Who we are:

- Virginia Commonwealth University Department of Electrical and Computer Engineering
  - Located in Richmond, VA, USA
  - +30,000 enrolled students in university
- **Dependable and Secure Cyber-Physical Systems Lab**
  - **Dr. Carl R. Elks, Ph.D.**
    - **Emal: crelks@vcu.edu**
  - Our lab is multidisciplinary, comprising faculty and research members whose expertise spans across hardware architectures, dependability and cyber-security, Unmanned Autonomous Systems, Nuclear Energy Instrumentation and Control, MEMs devices, and advanced sensors.
  - Our lab has a strong focus on experimental methods and practices, we develop techniques and methods that are applied to real world systems or verified with actual data.

U.S. DEPARTMENT OF ENERGY

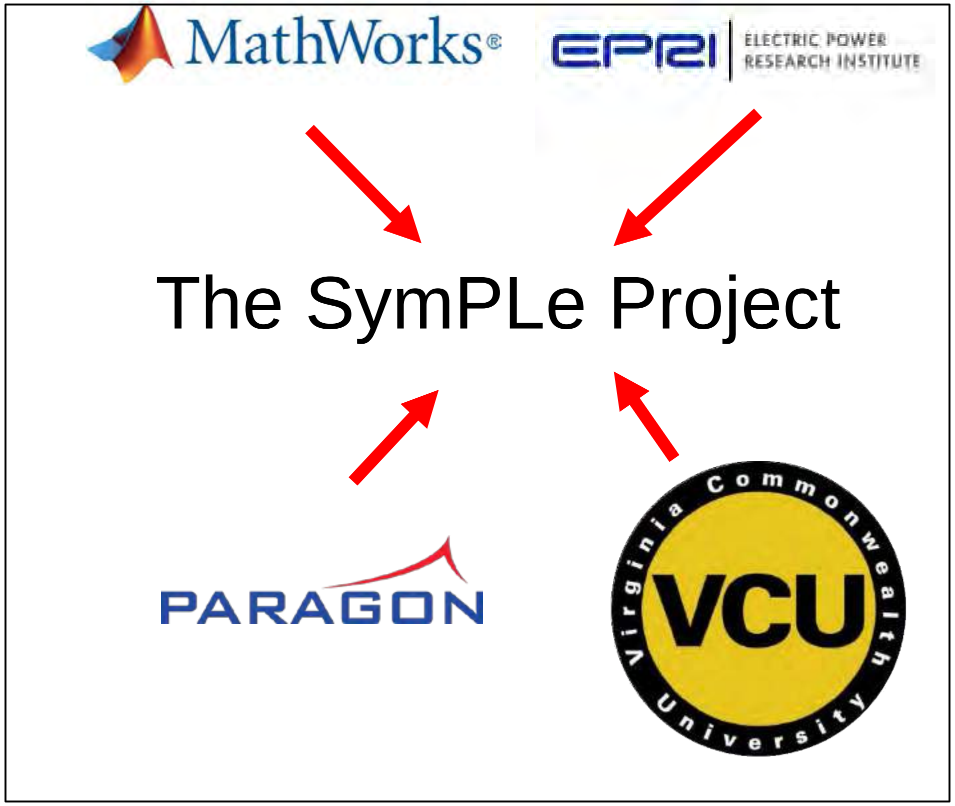
Office of Science

Idaho National Laboratory

ELECTRIC POWER RESEARCH INSTITUTE

REALIZING VERIFIABLE I&C AND EMBEDDED DIGITAL DEVICES FOR NUCLEAR POWER

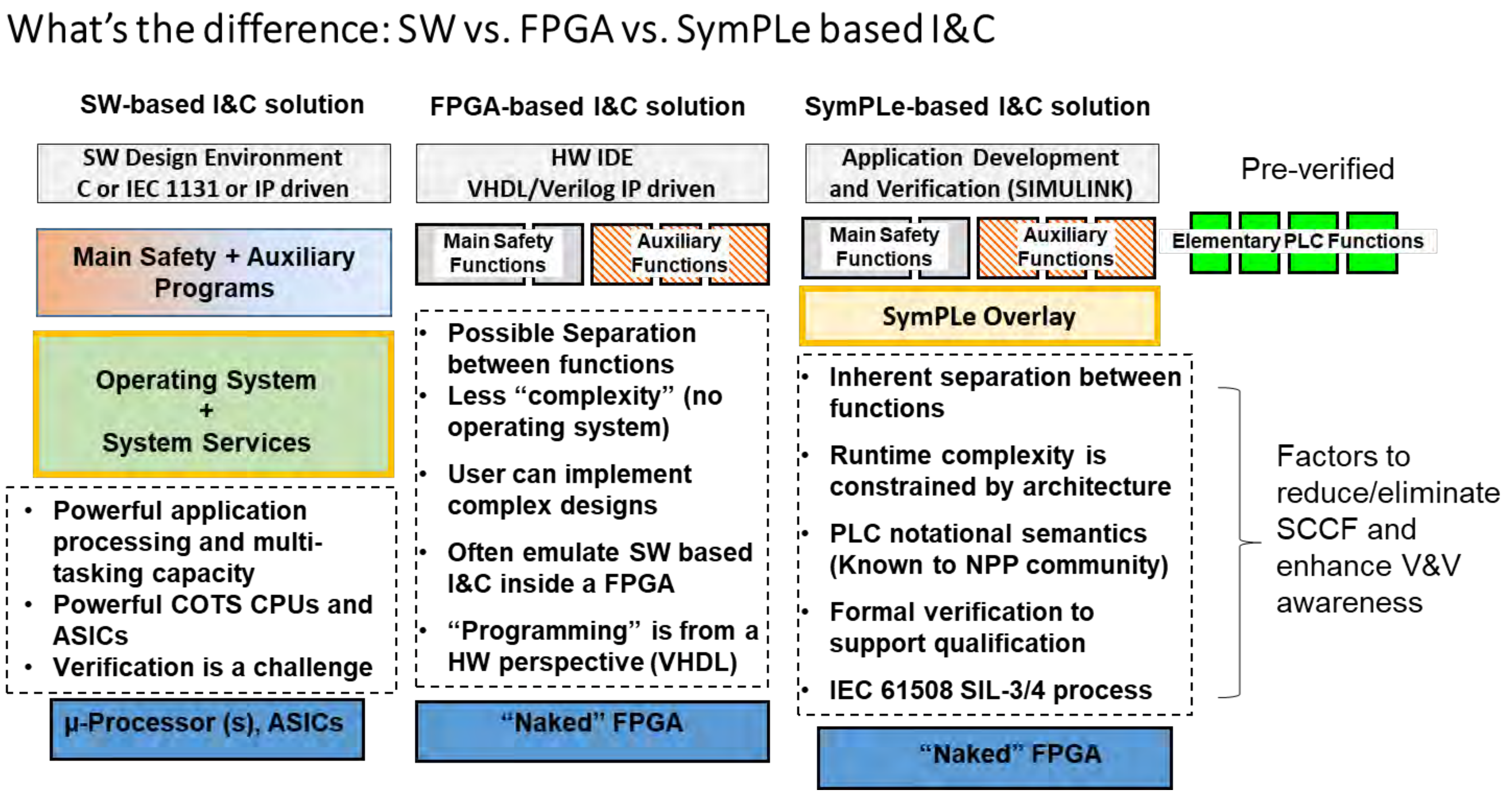
- Context: A DOE NEET-2 project led by EPRI, VCU principle research organization
- 2015-2019
- Goal: Develop science-based technologies and approaches for NPP I&C systems that show the potential for:
  - Reducing qualification burden of I&C systems
  - Reducing complexity to enhance V&V awareness
  - Address SW Common Cause Failures issues associated with digital I&C systems.
- Participants
  - Matt Gibson, Program PI, Electric Power Research Institute
  - Dr. Carl Elks, PI, Virginia Commonwealth University
  - Rick Hite, Smitha Gautham, Chris Deloglos, Athira Jayakamar-Virginia Commonwealth University
  - Dr. Ashraf Tantawy, Virginia Commonwealth University
  - Jason Moore, Mathworks


MathWorks
EPRI ELECTRIC POWER RESEARCH INSTITUTE
The SymPLe Project
PARAGON
Virginia Commonwealth University VCU

Research Perspective

- What’s the problem we are trying to address?
  - Safety Critical SW based systems expensive to develop
  - Certification/Licensing is even more expensive
- I&C systems in the context of nuclear power <u>may not need to be derivatives of software intensive systems</u> and by extension, not carrying the complexity associated with the SW intensive systems.
- Our approach called SymPLe is to rethink digital I&C from a perspective of three views: Simplicity, Extensibility and Verifiability.

Comparative View of I&C "Stacks"


What's the difference: SW vs. FPGA vs. SymPLe based I&C
SW-based I&C solution
FPGA-based I&C solution
SymPLe-based I&C solution
SW Design Environment C or IEC 1131 or IP driven
HW IDE VHDL/Verilog IP driven
Application Development and Verification (SIMULINK)
Pre-verified
Main Safety + Auxiliary Programs
Main Safety Functions
Auxiliary Functions
Main Safety Functions
Auxiliary Functions
Elementary PLC Functions
SymPLe Overlay
Operating System + System Services
μ-Processor (s), ASICs
"Naked" FPGA
"Naked" FPGA
Factors to reduce/eliminate SCCF and enhance V&V awareness

- Powerful application processing and multi-tasking capacity
- Powerful COTS CPUs and ASICs
- Verification is a challenge

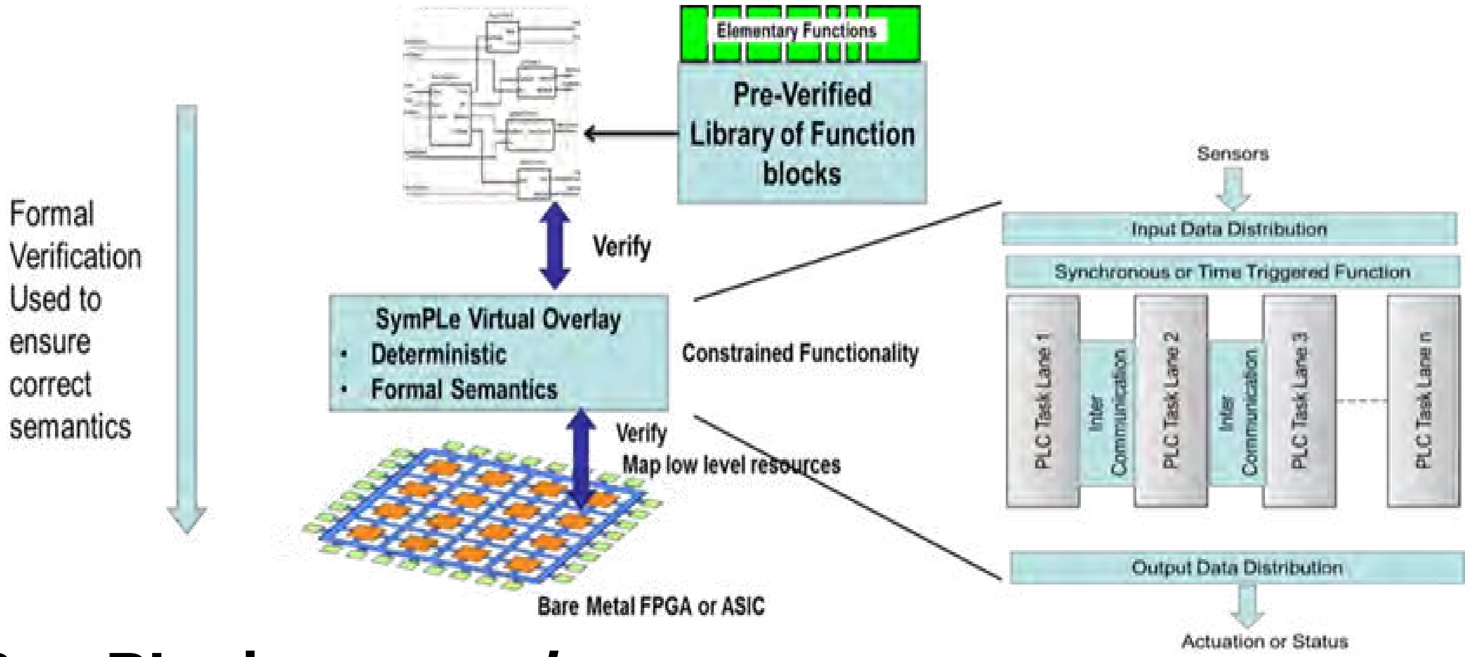
- Possible Separation between functions
- Less "complexity" (no operating system)
- User can implement complex designs
- Often emulate SW based I&C inside a FPGA
- "Programming" is from a HW perspective (VHDL)

- Inherent separation between functions
- Runtime complexity is constrained by architecture
- PLC notational semantics (Known to NPP community)
- Formal verification to support qualification
- IEC 61508 SIL-3/4 process

SymPLe Architectural Concepts

- HW FPGA based design
  - PLC-like Function Block programming
- Constrain design to favor <u>verifiable</u> execution behavior
- Constrain program <u>composability</u> rules to favor testing and comprehension
- Design for <u>predictability</u> - Well understood behaviors
- Well formed semantics – no side effects
  - *Engineer Accessible*: SymPLe is explicitly specified in a manner (e.g., language; structure) that is comprehensible to the community of its users and reviewers.
    - Function Blocks

SymPLe Concept

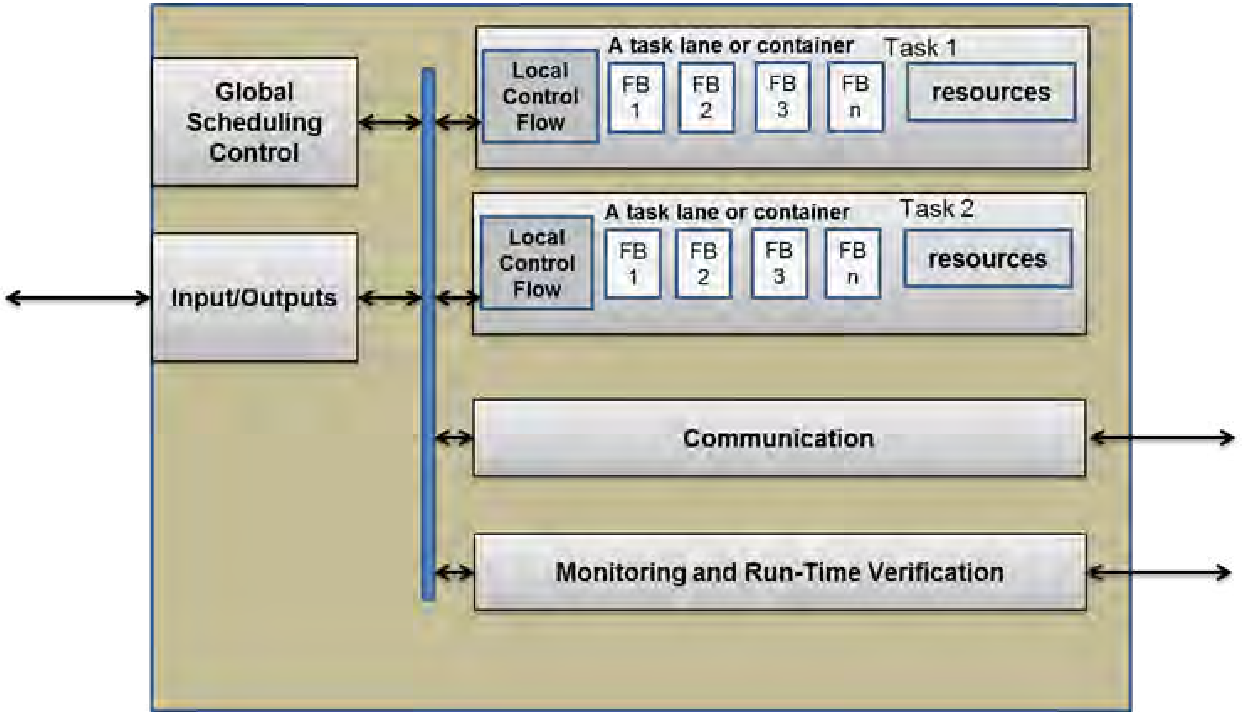


SymPLe is an *overlay*

- *SymPLe* is an architectural viewpoint that seeks to maximize reasoning, transparency and safety evidence while avoiding unnecessary complexity.
- *Engineer Accessible*: By adopting "PLC like" overlay architecture, we hope to make SymPLe extensible like a CPU based architecture – function blocks are the execution units.
- SymPLe is limited in what it can do – it trades computational power for verifiability.

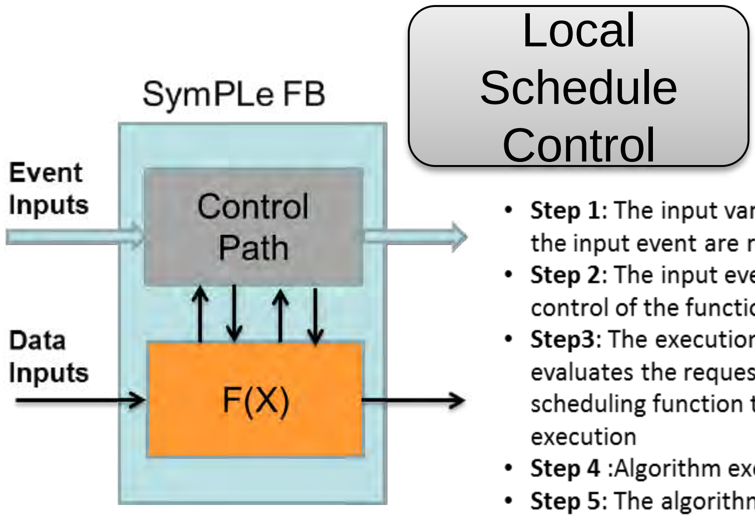
SymPLe Architecture

- Function Blocks – elementary functions used to create safety I&C functions.
- Local Control-responsible for dispatching:
  - the sequence of FB execution
  - marshaling inputs and outputs
  - and managing local state.
- global controller - provides global coordination and synchronization of task lanes
- Runtime Verification – Formal checking of executions, I/O, and non-interference.
- True Concurrency between Lanes



Function Block Architecture

- Common architecture for all SymPLe function blocks
  - Separation of control and dataflow with clear and defined interconnections
  - Formal Semantics
  - Inspired by IEC-61499
- SymPLe architecture variants differ in the control path of the generic function block
  - Autonomous Function Block
  - Lite Function Block

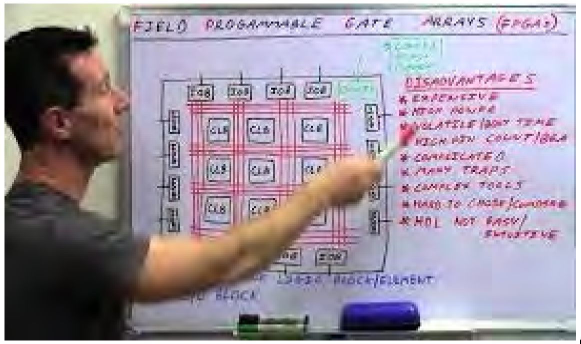


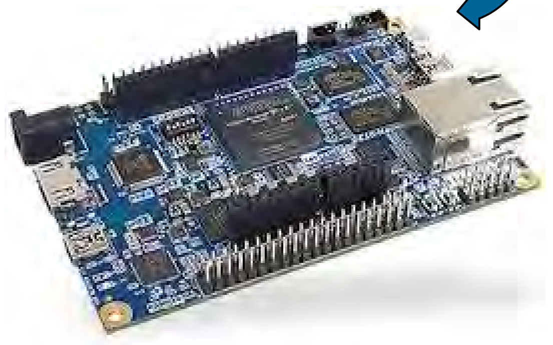
- **Step 1**: The input variable values relevant to the input event are registered and available.
- **Step 2**: The input event occurs, the execution control of the function block is triggered
- **Step3**: The execution control function evaluates the request and notifies the scheduling function to schedule algorithm for execution
- **Step 4** :Algorithm execution begins.
- **Step 5**: The algorithm completes the establishment of values for the output variables associated with the event output by the WITH qualifier
- **Step 6**: The resource scheduling function is notified that algorithm execution has ended.
- **Step 7**: The scheduling function invokes the execution control function.
- **Step 8**: The execution control function signals event at the event output.

Instruction	Description
AND, OR, NOT, XOR, NAND, NOR	Logical Operators
AND, OR, NOT, XOR, NAND, NOR	Bitwise Logical Operators
MAX, MIN, MUX	Selection Operators
GT, GE, EQ, LT, LE, NE	Comparison Operators
ADD, SUB, MUL, DIV	Arithmetic Operators
SLL, SLR	Bit-shift Operators
MOVE	System Operators

Ok, How do we go from a concepts to implementation – in 2 years?

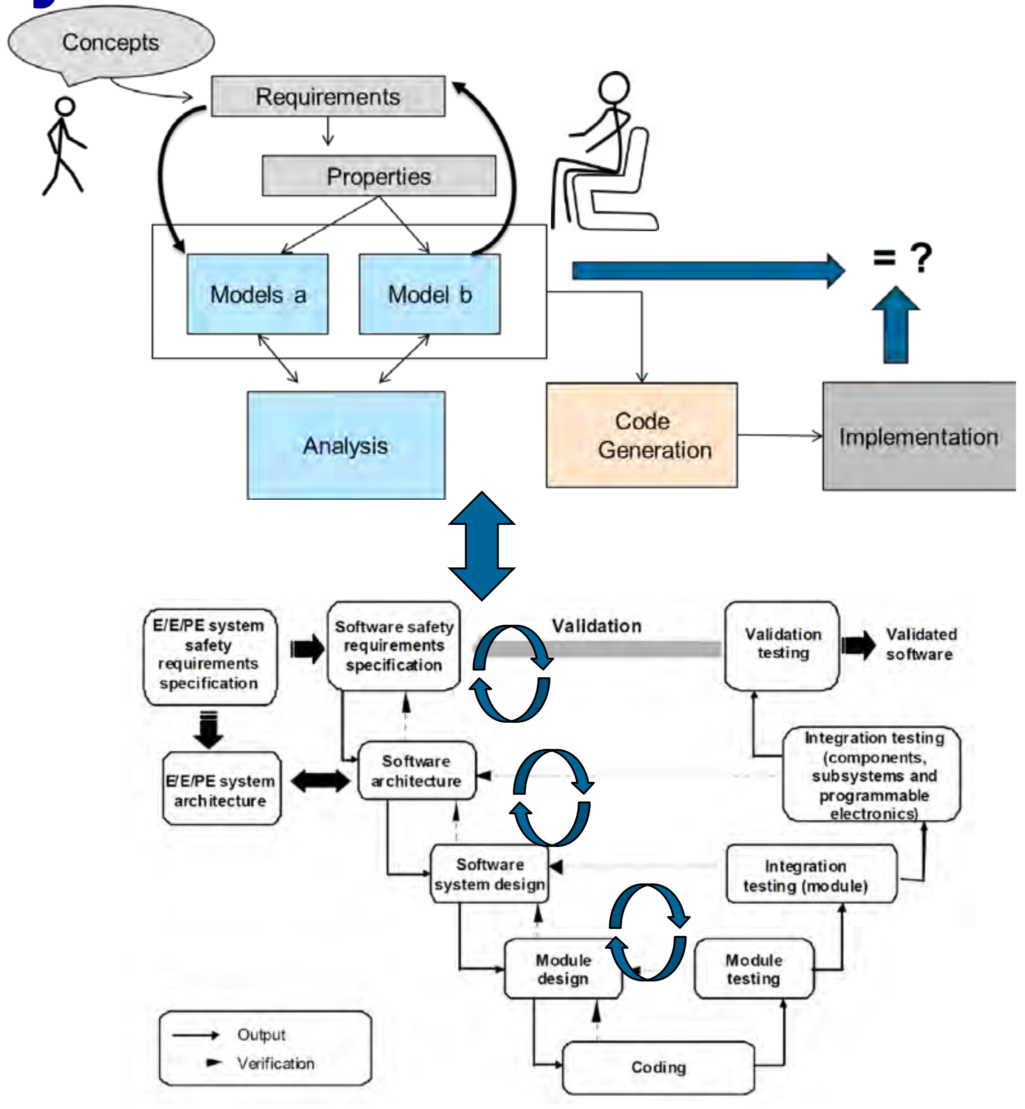
- VCU had extensive experience with HW Synthesis tools
  - Xilinx Vivado, Altera Quartus, full Mentor Graphics tool chains
  - All of our students use Xilinx tools for realizing embedded systems using SoC FPGAs.
- Formal verification will be a large part of the effort
  - Must have accessible formal verification
- We also want to produce “evidence” that is accessible to stakeholders
- Given the above, model based engineering approach seemed appealing.
  - Wide-spread use in Automotive and Aviation
  - Real example of reduced costs and improved quality and safety
  - But, MBD & E not widely used in FPGA or HW based designs… yet…
  - Maybe a big opportunity, or big risk?





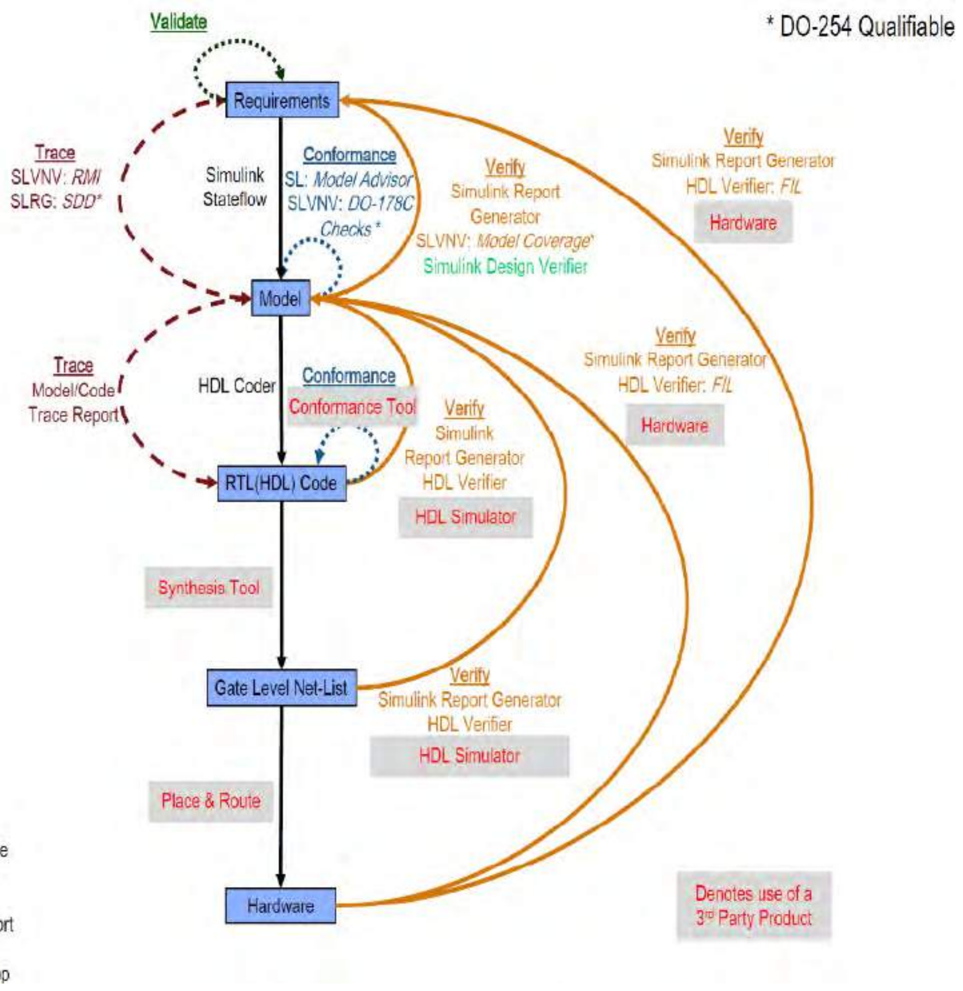
Your What's are my How's

- It is often the case that system development does not proceed in an orderly top-down fashion.
- But rather through co-evolution of requirements and architectural design
- This is especially true for concept to implementation efforts.
- Model based design is very good, in this regard.



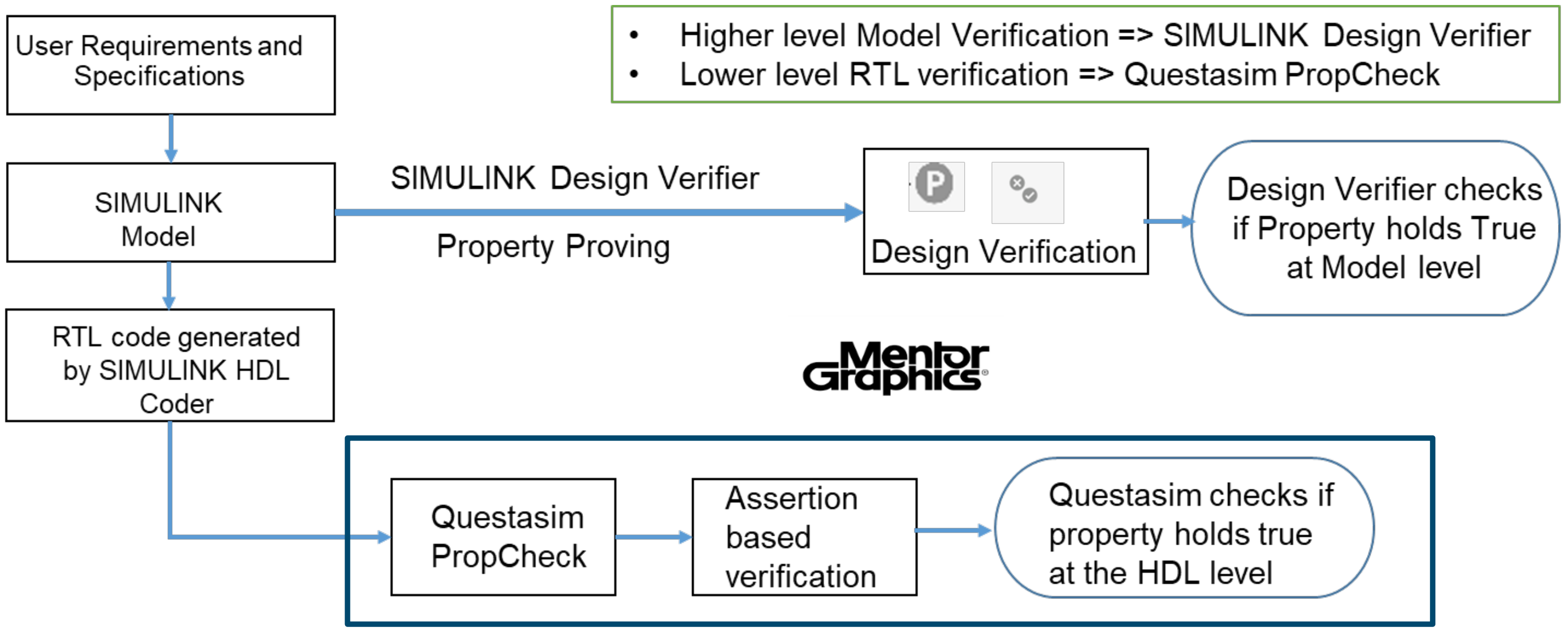
SymPLe Development and Design Assurance Workflow

- Currently following IEC 61508 SIL 3/4
- Verification at each step from requirements to implementation
- Math works Tools used throughout V&V workflow
  - **Prevents gaps in V&V from oversight or misinterpretation**
- Critical toolboxes are TUV certified



Abbreviations
SL: Simulink
SLVNV: Simulink Verification and Validation
RMI: Requirements Management Interface
SDD: System Design Description
SLRG: Simulink Report Generator
FIL: FPGA-in-the-Loop

End-to-End Property Verification using Simulink DV and MENTOR GRAPHICS Questasim



- Higher level Model Verification => SIMULINK Design Verifier
- Lower level RTL verification => Questasim PropCheck
User Requirements and Specifications
SIMULINK Model
RTL code generated by SIMULINK HDL Coder
SIMULINK Design Verifier
Property Proving
Design Verification
Design Verifier checks if Property holds True at Model level
Mentor Graphics
Questasim PropCheck
Assertion based verification
Questasim checks if property holds true at the HDL level

24

Verifies that actual SymPLe HW has inherited the proven property of models

Tool Options We Considered

- Primary criteria –
  - Legacy with real safety critical system design
  - Tools must be cleanly integrated into a IDE
  - Must generate clean HDL from models
  - Formal verification tools
  - Bi-directional traceability from low-level code to requirements
  - Commercially supported
  - Tool qualification (nice, but not strictly required)
- The contenders
  - SCADE – Ansys
  - Mathworks –Simulink, Design Verifier, and associated toolboxes
  - GME – Vanderbilt academic tool

Observation 1: The lots of moving parts problem

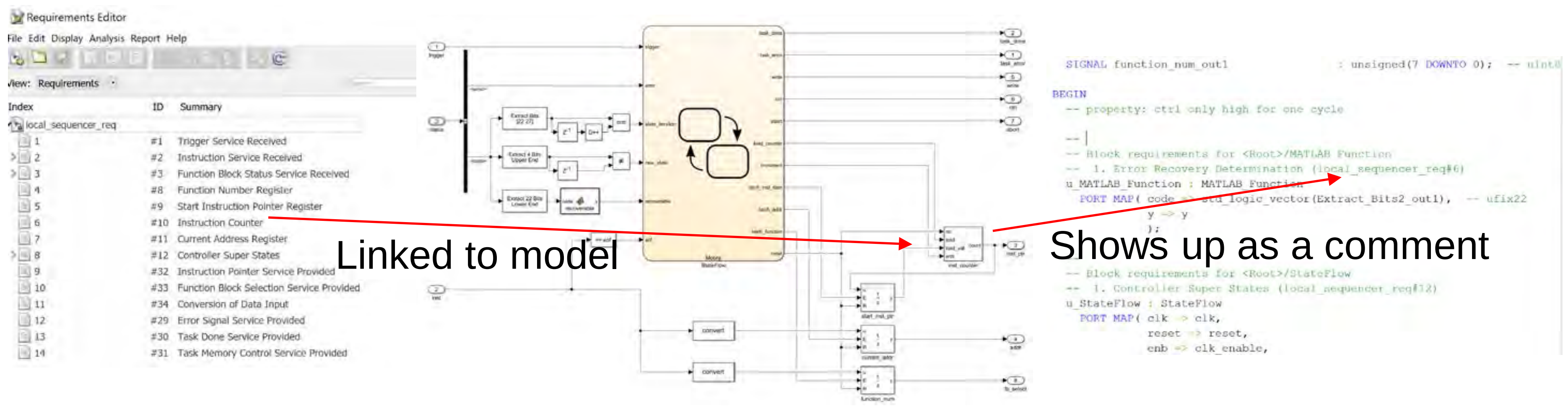
- The state of-the-art in Mathworks IDE is a powerful design and verification environment, but it comes at a cost .
- Lots of coupled toolboxes for automatic HDL and C code-generation, simulation coupled with requirement monitoring, co-simulation of heterogeneous models, model-based analysis including verification of compliance of requirements and specification, formal verification model-based test-generation, rapid prototyping, and virtual integration testing
- You have to invest time and training to unleash the power of these dedicated specialized tools.
  - Simulink
  - Design Verifier
  - Simulink Test
  - Simulink Report Generator
  - HDL Coder
  - Verification and Validation – Multiple toolboxes
  - HDL Verifier
  - IEC Certification Kit (61508)
  - Fixed-point Designer
  - Simulink Stateflow

80% of architectural design is here. These are the toolboxes you need for model-based FPGA design following IEC 61508

- Synthesis Tools
  - Xilinx Vivado
  - Mentor Graphic QuestaSim, Propcheck, Propcover
  - Custom Tool - MBAC: runtime verification

Observation 2: Vertical Integration – traceability of requirements to functions to implementations

- In general, work well. Requirements Management is now well-developed in Matlab Simulink.
- Automatic report test example generators are nice for standards..
- Lower level integration into HW synthesis tools (so far so good)
- Requirements captured in Simulink Requirements Editor or external requirements document.
- Requirements are linked to where they are implemented in the model with a bi directional link.
- Low level requirements can also be directly linked to high level/system level requirements directly.
- Linked requirements are then placed textually as comments in any generated code.
- Generated code comes in 2 versions: pure text and HTML. HTML version the linked requirements are links that will navigate to the requirements document.
- Linking produces traceability matrix

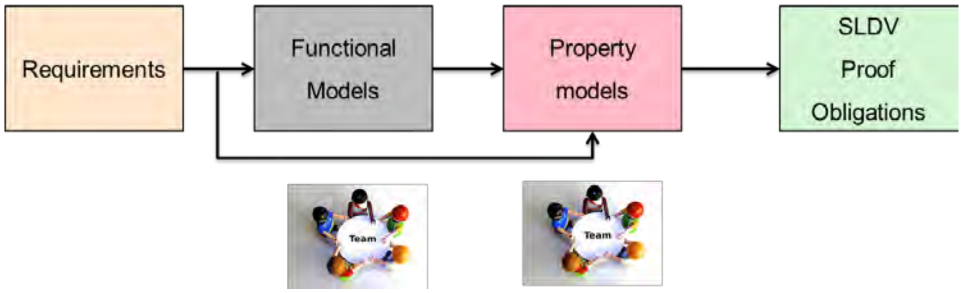


Index	ID	Summary
local_sequencer_req		
1	#1	Trigger Service Received
2	#2	Instruction Service Received
3	#3	Function Block Status Service Received
4	#8	Function Number Register
5	#9	Start Instruction Pointer Register
6	#10	Instruction Counter
7	#11	Current Address Register
8	#12	Controller Super States
9	#32	Instruction Pointer Service Provided
10	#33	Function Block Selection Service Provided
11	#34	Conversion of Data Input
12	#29	Error Signal Service Provided
13	#30	Task Done Service Provided
14	#31	Task Memory Control Service Provided

```
  SIGNAL function_num_out1                : unsigned(7 DOWNTO 0);  -- uint8

BEGIN
  -- property: ctrl only high for one cycle

  --|
  -- Block requirements for <Root>/MATLAB Function
  --  1. Error Recovery Determination (local_sequencer_req#6)
  u_MATLAB_Function : MATLAB_Function
    PORT MAP( code => std_logic_vector(Extract_Bits2_out1),  -- ufix22
              y => y
              );

  -- Block requirements for <Root>/StateFlow
  --  1. Controller Super States (local_sequencer_req#12)
  u_StateFlow : StateFlow
    PORT MAP( clk => clk,
              reset => reset,
              enb => clk_enable,
```

Observation 3: Horizontal Model Structuring

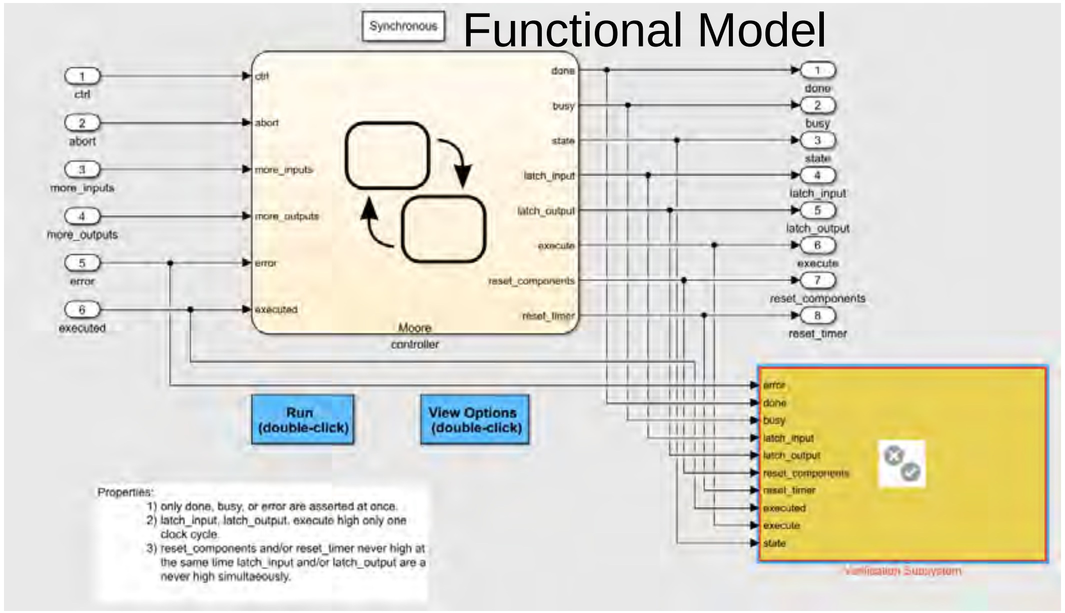
- You need to pay attention to this…
- Horizontal Models are models that support lifecycle activities. Determines the role the model plays in the development life cycle.
- It was not easy to see how to structure models at the beginning. We tended to mix models across domains.
- Good horizontal structuring facilitates parallel independent V&V and development via separation of concerns
  - As long as modelers agree on input and output interface semantics then teams can independently build prover models, testing models, and environment models – independent of the design model.
- Allows expert teams to focus on their domain
- We partition both horizontally and vertically to achieve maximal utility from models



Example: How the functional and Property models play together

Design Verifier

- A collection of verification tools, of which the strongest is formal model checking, and K-induction.
- Used for Property Proving: Properties are written for critical conditions that has to hold true in the design. Design verifier formally verifies by checking the properties in the <u>entire state space for all inputs</u>. If the properties are falsified, detailed report with counter examples is generated
- Design verifier can also be used to generate test cases to increase coverage

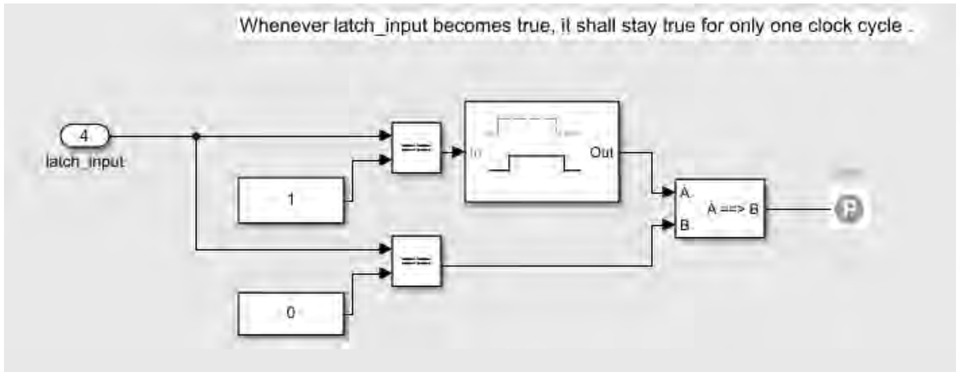


Properties:
1) only done, busy, or error are asserted at once.
2) latch_input, latch_output, execute high only one clock cycle.
3) reset_components and/or reset_timer never high at the same time latch_input and/or latch_output are a never high simultaeously.

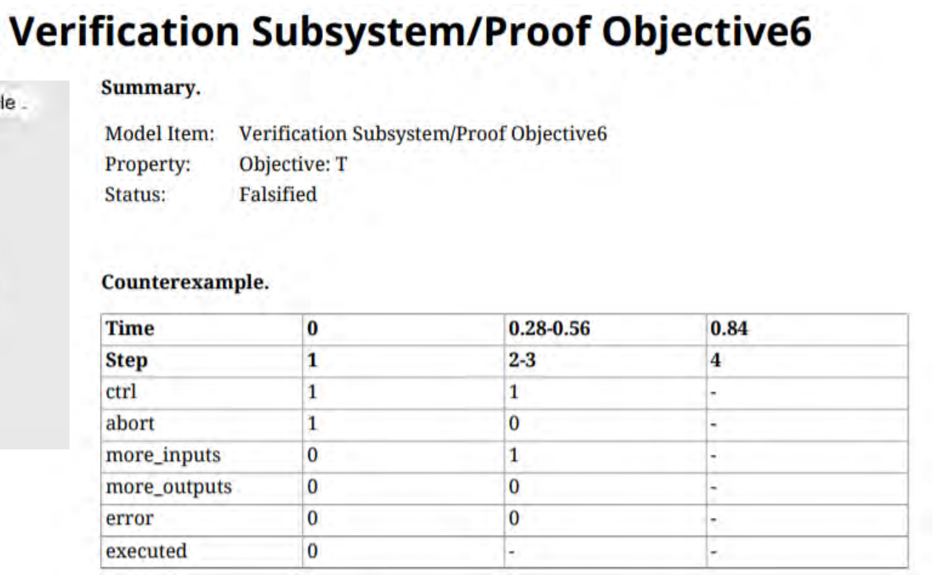
Model and the Verification Subsystem having the Properties that have to be verified by the Design Verifier

How the Horizontal models play together: Design Verifier

- One of the properties that we wanted to check was falsified and a detailed report was generated indicating the falsified property with a counterexample


Whenever latch_input becomes true, it shall stay true for only one clock cycle .
4
latch_input
==
1
==
0
In
Out
A
B
A ==> B
P

Property to check that the latch_input stays true for only one clock cycle



Verification Subsystem/Proof Objective6

Summary.

Model Item:	Verification Subsystem/Proof Objective6
Property:	Objective: T
Status:	Falsified

Counterexample.

Time	**0**	**0.28-0.56**	**0.84**
Step	**1**	**2-3**	**4**
ctrl	1	1	-
abort	1	0	-
more_inputs	0	1	-
more_outputs	0	0	-
error	0	0	-
executed	0	-	-

Design verifier report indicating the falsified property and a detailed counterexample

Example: How the models play together

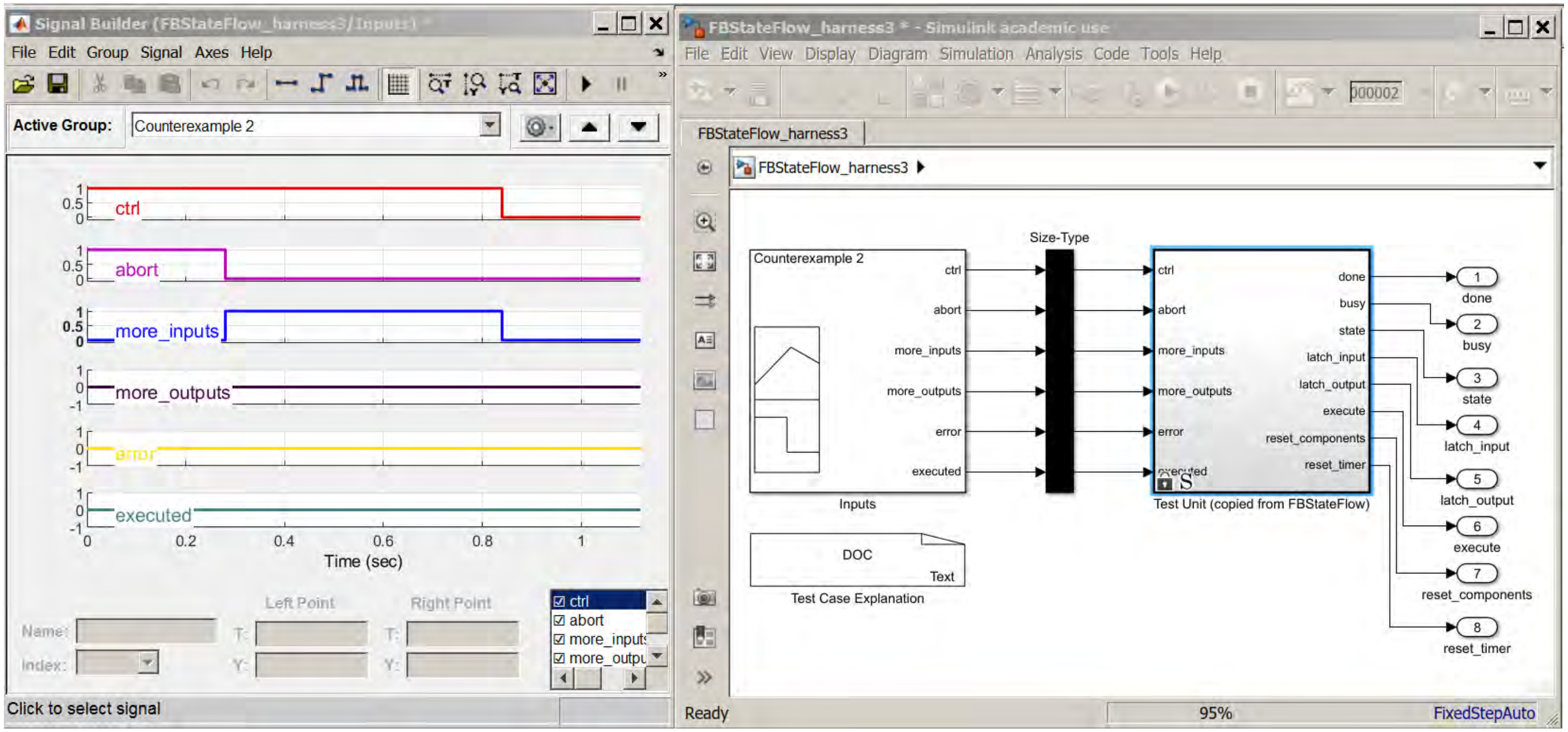
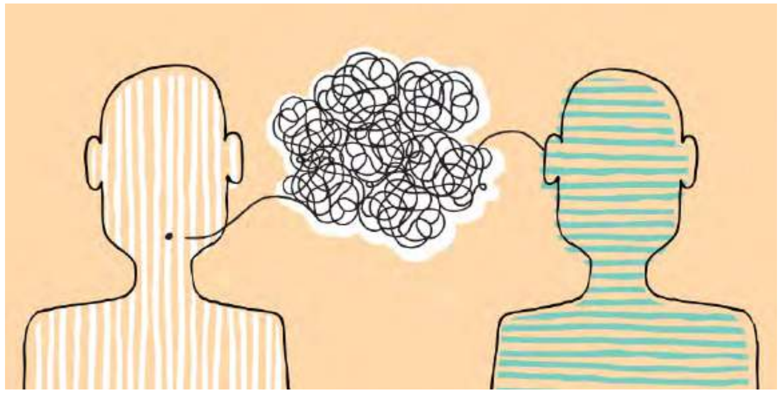


Fig: Counterexample indicating the input conditions for which the property failed

Observation 4: Usability Across Specialization Domains

- Does MBE tools help team members convey "specifics" of their domain to others?
- So far, we seem to think so…
- The ambiguity arising due to requirements communication via Natural Language seems to be reduced … precise models with assumptions help the translation…
- The inference is less chances of errors due to requirement misunderstandings by the implementer.
- Single, common algorithmic/model development environment allows different stakeholders into the picture early. Identification of missed or incomplete requirements is better facilitated.
- Model management can be a problem when you have lot's of changes..
- **Common environment for understanding**



Observation 5: Common environment for understanding

- Very well – could be a key to regulator/supplier evidence communication
- Single, common algorithmic/model development environment allows different stakeholders into the picture early. Identification of missed or incomplete requirements is better facilitated.
- No mix of prototyping languages
- Commonality fosters sharing, algorithm/utility reuse
- Use a single, secure, collaborative, web-based sharing repository (github/Gitlab, bitbucket, etc…)
- Functional models allow low-level designs and implementations to be critically reviewed with better comprehension.

Observation 6: Evidence based Artifacts are useful for reviews

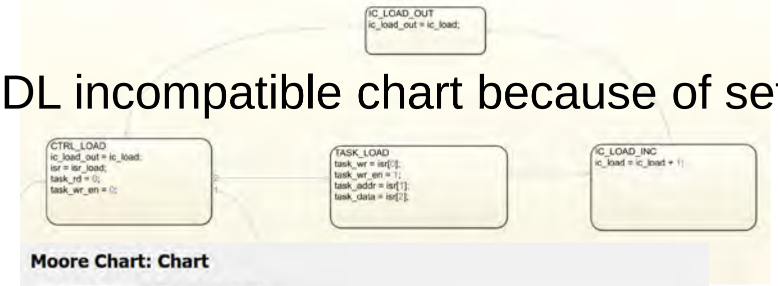
- Very well, needs some help from lower level Synthesis tools.
- We found that models, and the results/data are very good for confirming or refuting a claim.
- Requirements, sub-requirements and assumptions are explicitly chained in all models
- Testing, formal verification in the Simulink
- When you get to VHDL and transfer to HW synthesis you may need to check to see if you can trace bi-directionally.
- Cite preliminary results for Paragon.

- IMAGE REQ DOC TO THE MODEL TO THE TEST RESULTS TO THE PROVER RESULTS
- PROP CHECK IMAGE

Observation 7: Be careful with Models to HDL Code

- Not all blocks, block parameter settings, and design patterns in the Simulink libraries are HDL compatible
- Much time spent on development of a working model only to find HDL code generation incompatibilities.
- Simple solution: run script hdllib. Hides all incompatible tool boxes.

HDL incompatible chart because of setting below

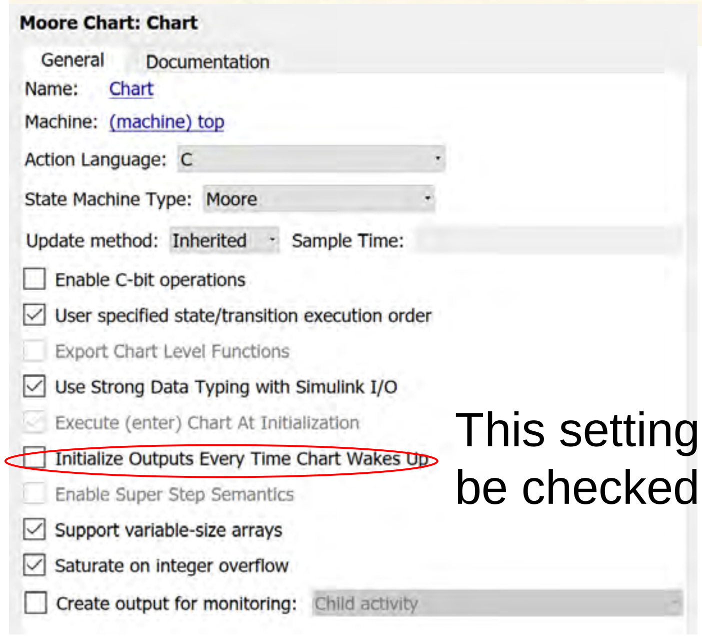


```
IC_LOAD_OUT
ic_load_out = ic_load;
```

```
CTRL_LOAD
ic_load_out = ic_load;
isr = isr_load;
task_rd = 0;
task_wr_en = 0;
```

```
TASK_LOAD
task_wr = isr[0];
task_wr_en = 1;
task_addr = isr[1];
task_data = isr[2];
```

```
IC_LOAD_INC
ic_load = ic_load + 1;
```



Moore Chart: Chart

General | Documentation

Name: Chart

Machine: (machine) top

Action Language: C

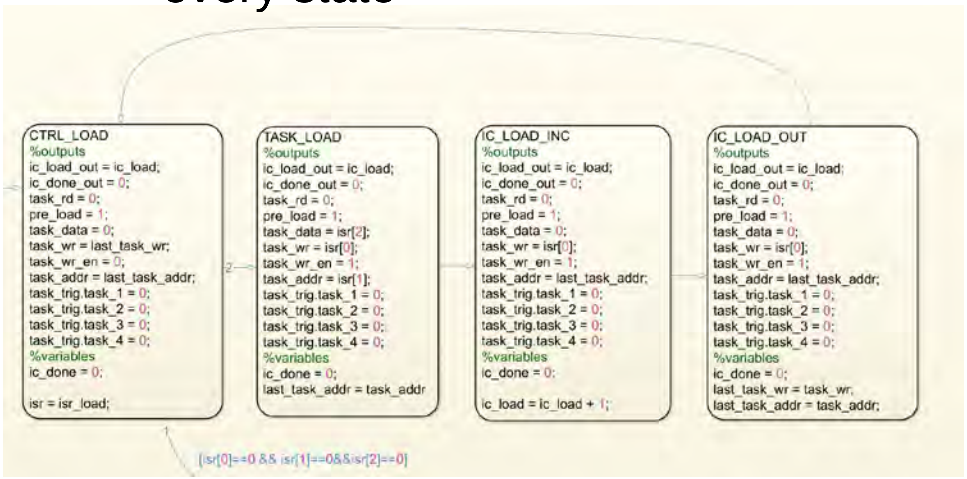
State Machine Type: Moore

Update method: Inherited   Sample Time:

- ☐ Enable C-bit operations
- ☑ User specified state/transition execution order
- ☐ Export Chart Level Functions
- ☑ Use Strong Data Typing with Simulink I/O
- ☑ Execute (enter) Chart At Initialization
- ☐ Initialize Outputs Every Time Chart Wakes Up
- ☐ Enable Super Step Semantics
- ☑ Support variable-size arrays
- ☑ Saturate on integer overflow
- ☐ Create output for monitoring: Child activity

This setting must be checked

Checking box required explicit definition of chart output at every state



```
CTRL_LOAD
%outputs
ic_load_out = ic_load;
ic_done_out = 0;
task_rd = 0;
pre_load = 1;
task_data = 0;
task_wr = last_task_wr;
task_wr_en = 0;
task_addr = last_task_addr;
task_trig.task_1 = 0;
task_trig.task_2 = 0;
task_trig.task_3 = 0;
task_trig.task_4 = 0;
%variables
ic_done = 0;

isr = isr_load;
```

```
TASK_LOAD
%outputs
ic_load_out = ic_load;
ic_done_out = 0;
task_rd = 0;
pre_load = 1;
task_data = isr[2];
task_wr = isr[0];
task_wr_en = 1;
task_addr = isr[1];
task_trig.task_1 = 0;
task_trig.task_2 = 0;
task_trig.task_3 = 0;
task_trig.task_4 = 0;
%variables
ic_done = 0;
last_task_addr = task_addr
```

```
IC_LOAD_INC
%outputs
ic_load_out = ic_load;
ic_done_out = 0;
task_rd = 0;
pre_load = 1;
task_data = 0;
task_wr = isr[0];
task_wr_en = 1;
task_addr = last_task_addr;
task_trig.task_1 = 0;
task_trig.task_2 = 0;
task_trig.task_3 = 0;
task_trig.task_4 = 0;
%variables
ic_done = 0;

ic_load = ic_load + 1;
```

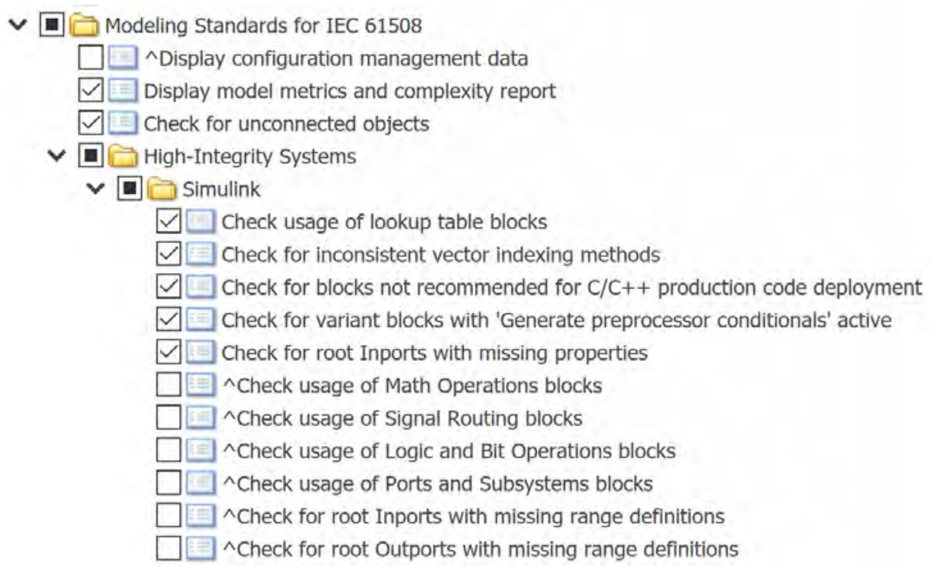
```
IC_LOAD_OUT
%outputs
ic_load_out = ic_load;
ic_done_out = 0;
task_rd = 0;
pre_load = 1;
task_data = 0;
task_wr = isr[0];
task_wr_en = 1;
task_addr = last_task_addr;
task_trig.task_1 = 0;
task_trig.task_2 = 0;
task_trig.task_3 = 0;
task_trig.task_4 = 0;
%variables
ic_done = 0;
last_task_wr = task_wr;
last_task_addr = task_addr;
```

```
[isr[0]==0 && isr[1]==0&&isr[2]==0]
```

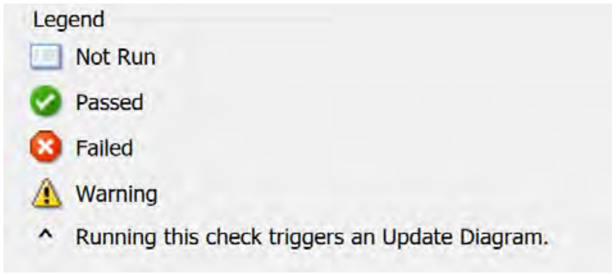
Observation 8: Risk management in the design space

- More complex designs (but potentially better performance) can be identified early and assessed in terms of verification risk.
- The model based verification tools can provide preliminary feedback
- We did this early on with function blocks designs.
- Incremental HDL code generation compatibly checks
- Incremental Design Verifier compatibility checks
- Model Advisor usage to perform incremental modeling standard compliance
- Management of design complexity to assist formal verification
  - Limit functionality of blocks to logical units
  - Prove properties about individual units
  - Use proven properties to build arguments that support higher level properties and requirements.
- Strict data typing defined in parameter setup file.
  - Ensures consistency of typing wherever used.
  - Also allows changing of the data type with a single change in the parameter file without the need to revise manually wherever the data type appears.
  - Data types in this context include composition of buses, Qmn format for data in architecture

Model Advisor Menu



- ■ Modeling Standards for IEC 61508
  - ☐ ^Display configuration management data
  - ☑ Display model metrics and complexity report
  - ☑ Check for unconnected objects
  - ■ High-Integrity Systems
    - ■ Simulink
      - ☑ Check usage of lookup table blocks
      - ☑ Check for inconsistent vector indexing methods
      - ☑ Check for blocks not recommended for C/C++ production code deployment
      - ☑ Check for variant blocks with 'Generate preprocessor conditionals' active
      - ☑ Check for root Inports with missing properties
      - ☐ ^Check usage of Math Operations blocks
      - ☐ ^Check usage of Signal Routing blocks
      - ☐ ^Check usage of Logic and Bit Operations blocks
      - ☐ ^Check usage of Ports and Subsystems blocks
      - ☐ ^Check for root Inports with missing range definitions
      - ☐ ^Check for root Outports with missing range definitions

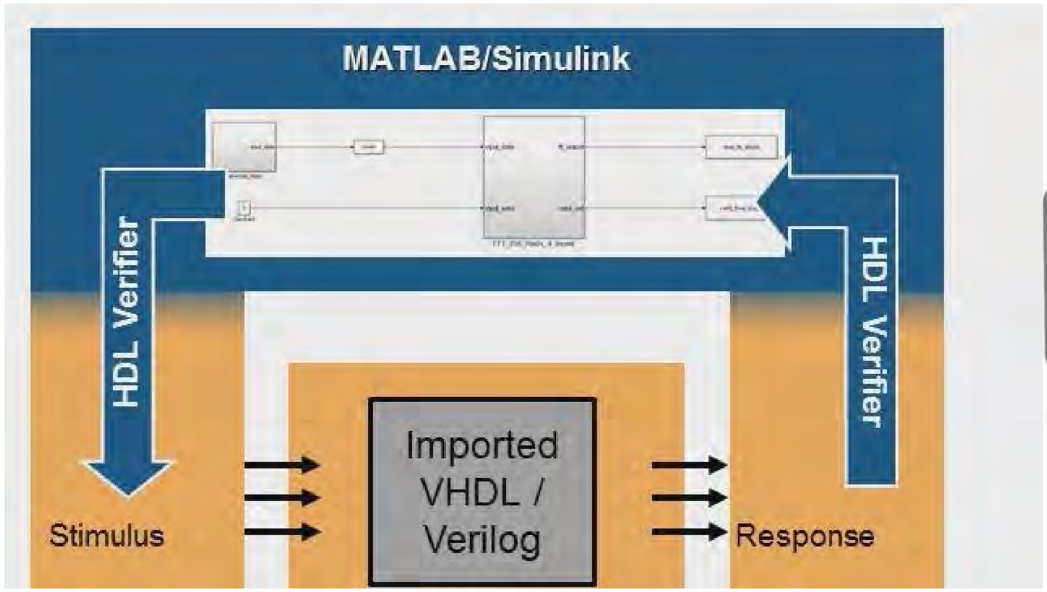


Legend

▭	Not Run
✓	Passed
✗	Failed
⚠	Warning
^	Running this check triggers an Update Diagram.

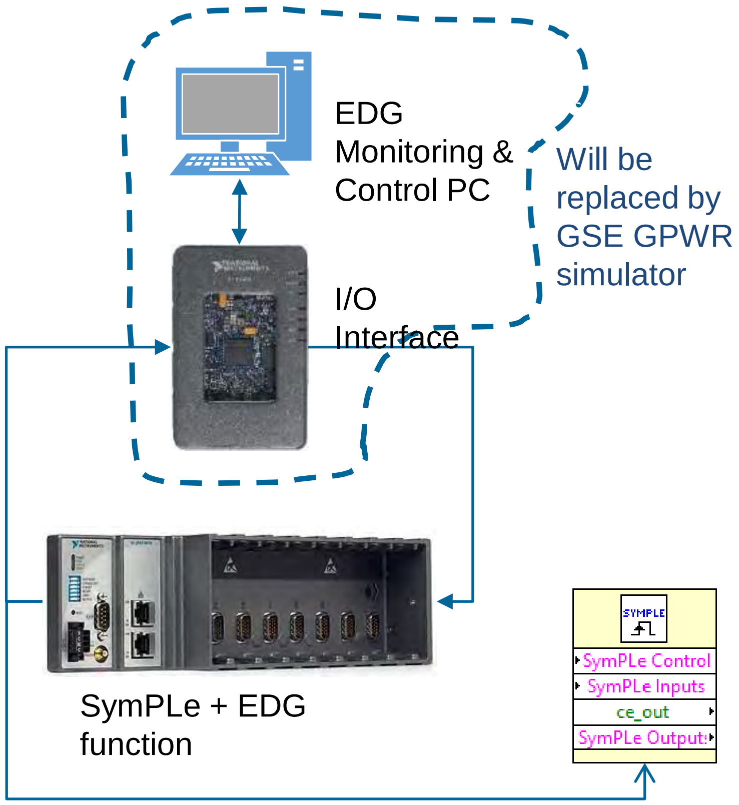
Observation 9: Co-simulation is nice

- Test cases used in models help us demonstrate equivalency between model and downstream components: HDL, netlists, and hardware implementation.
- Model confirms implementations



Observation 9 Continued

- HIL was setup for the Emergency Diesel Generator (EDG) application.
- A simple user interface was built using National Instruments LabVIEW software to manipulate EDG inputs and monitor the outputs.
- SymPLe architecture, implementing EDG logic, was downloaded to National Instruments cRIO controller FPGA.
- Using modular design, a SymPLe subsystem block was developed in LabVIEW, where it could be reused in other test programs.

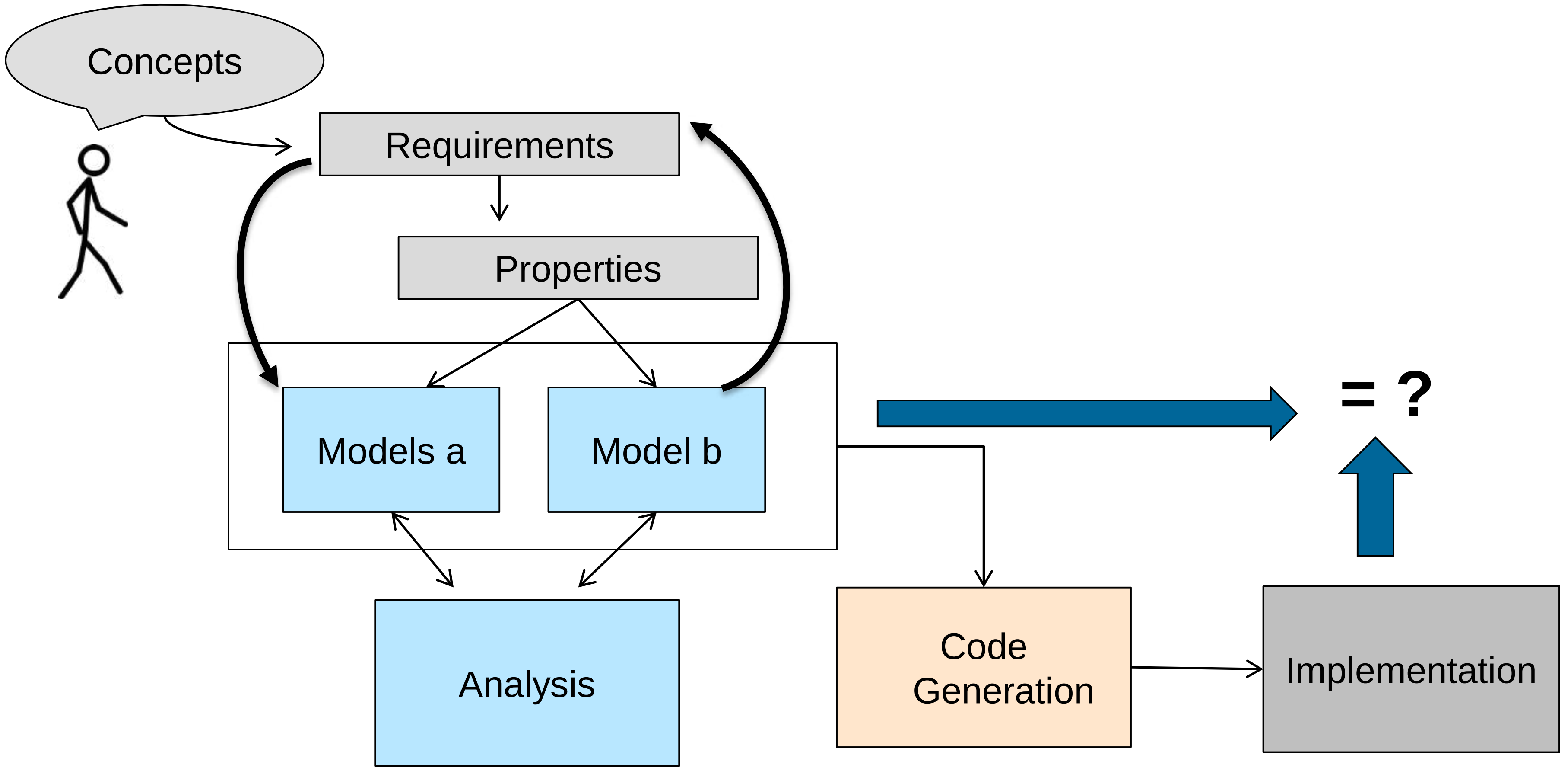


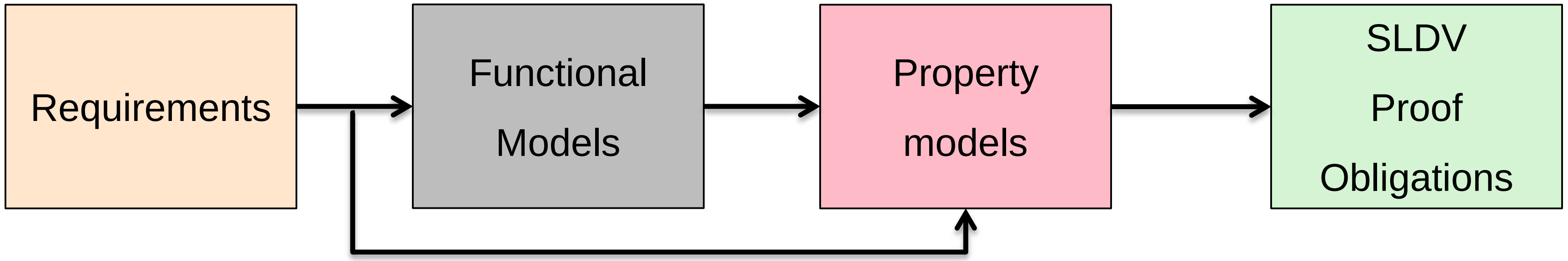
Overall Experience Summary

- Generally very pleased with the outcomes so far.
- Investment is necessary to reap the full benefits
  - Does not eliminate HW synthesis tools, but allows reasoning to be “moved” up to where more stakeholders are inclusive
  - Much more formal in it’s approach. Which is a good thing…
- It’s a paradigm shift from traditional “platform” based design
- Great for making the “evidence” accessible and transparent to reviewers, stakeholders.
  - Paragon Inc is reviewing the models, evidences, and results as part of a “mock” CGD.
- We are finding difficult bugs, which is a metric in itself..
- Would have been very difficult to achieve project goals in traditional HW synthesis tools
- Not perfect, room for improvement, but the benefits (at least to us) are very compelling.

Model-Based Development Examples(circa 2010)

Company	Product	Tools	Specified & Autocoded	Benefits Claimed
Airbus	A340	SCADE With Code Generator	• 70% Fly-by-wire Controls • 70% Automatic Flight Controls • 50% Display Computer • 40% Warning & Maint Computer	• 20X Reduction in Errors • Reduced Time to Market
Eurocopter	EC-155/135 Autopilot	SCADE With Code Generator	• 90 % of Autopilot	• 50% Reduction in Cycle Time
GE & Lockheed Martin	FADEDC Engine Controls	ADI Beacon	• Not Stated	• Reduction in Errors • 50% Reduction in Cycle Time • Decreased Cost
Schneider Electric	Nuclear Power Plant Safety Control	SCADE With Code Generator	• 200,000 SLOC Auto Generated from 1,200 Design Views	• 8X Reduction in Errors while Complexity Increased 4x
US Spaceware	DCX Rocket	MATRIXx	• Not Stated	• 50-75% Reduction in Cost • Reduced Schedule & Risk
PSA	Electrical Management System	SCADE With Code Generator	• 50% SLOC Auto Generated	• 60% Reduction in Cycle Time • 5X Reduction in Errors
CSEE Transport	Subway Signaling System	SCADE With Code Generator	• 80,000 C SLOC Auto Generated	• Improved Productivity from 20 to 300 SLOC/day
Honeywell Commercial Aviation Systems	Primus Epic Flight Control System	MATLAB Simulink	• 60% Automatic Flight Controls	• 5X Increase in Productivity • No Coding Errors • Received FAA Certification


Concepts
Requirements
Properties
Models a
Model b
Analysis
Code Generation
Implementation
= ?


Requirements
Functional Models
Property models
SLDV Proof Obligations

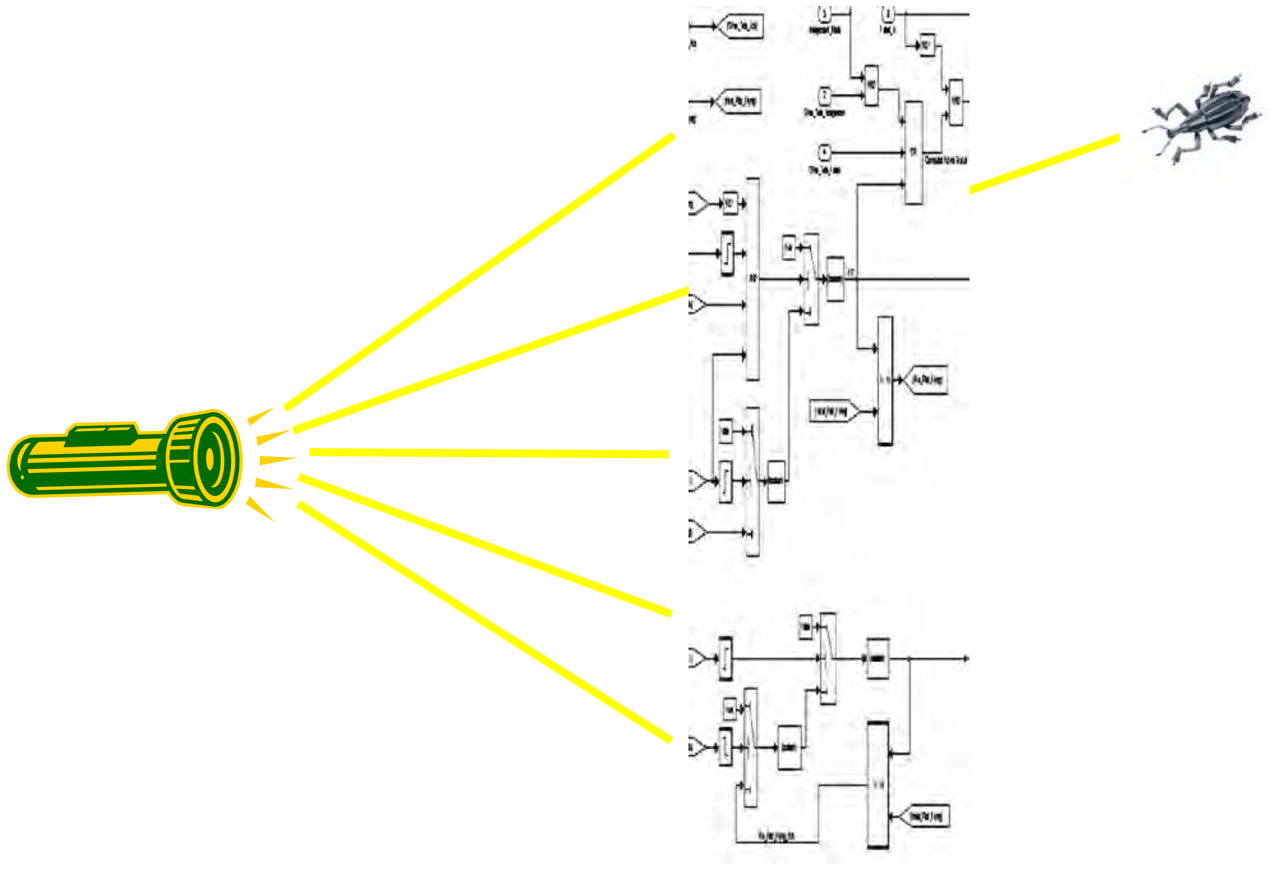
Team

Team

Advantage of Model Checking

Testing Checks Only the Values We Select

Even Small Systems Have Trillions (of Trillions) of Possible Tests!



Advantage of Model Checking

Model Checker Tries Every Possible Input and State!