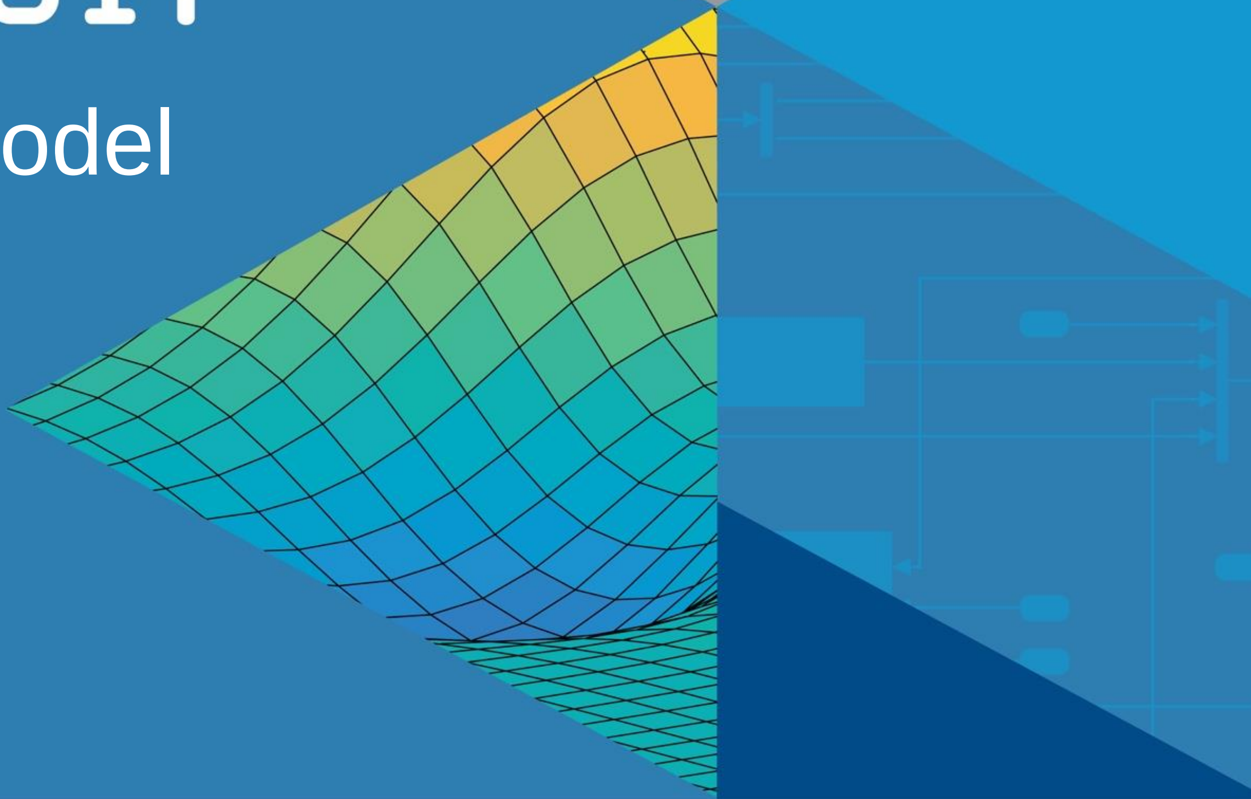
MATLAB EXPO 2017

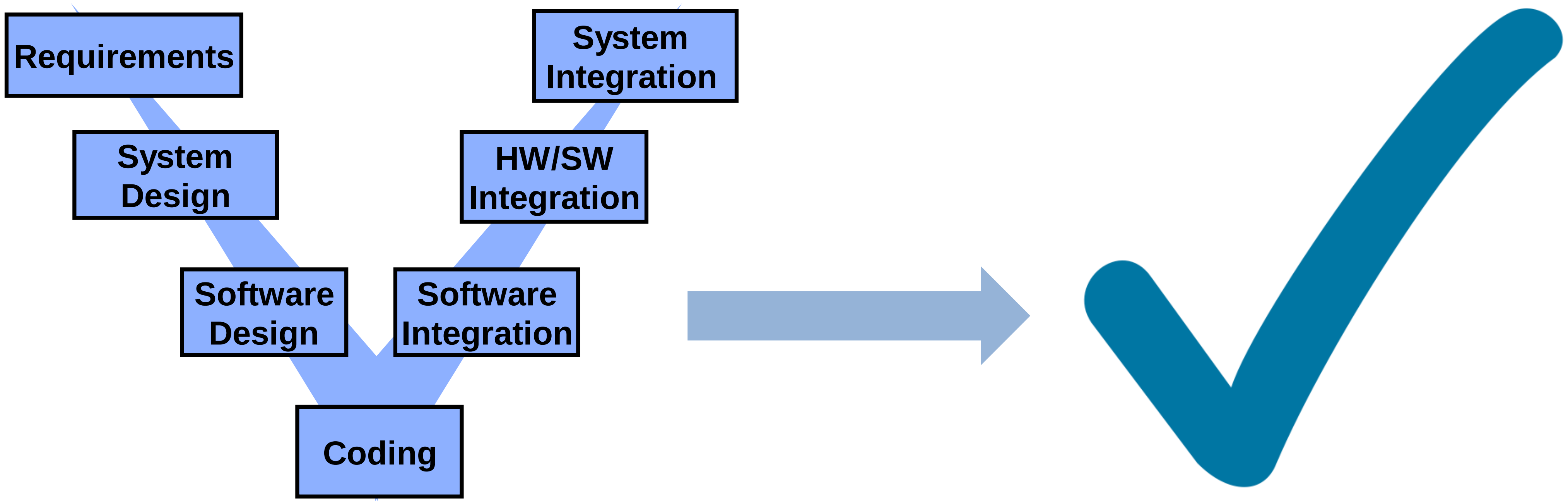
Verification Techniques for Model and Code

Paul Lambrechts



Key Takeaway

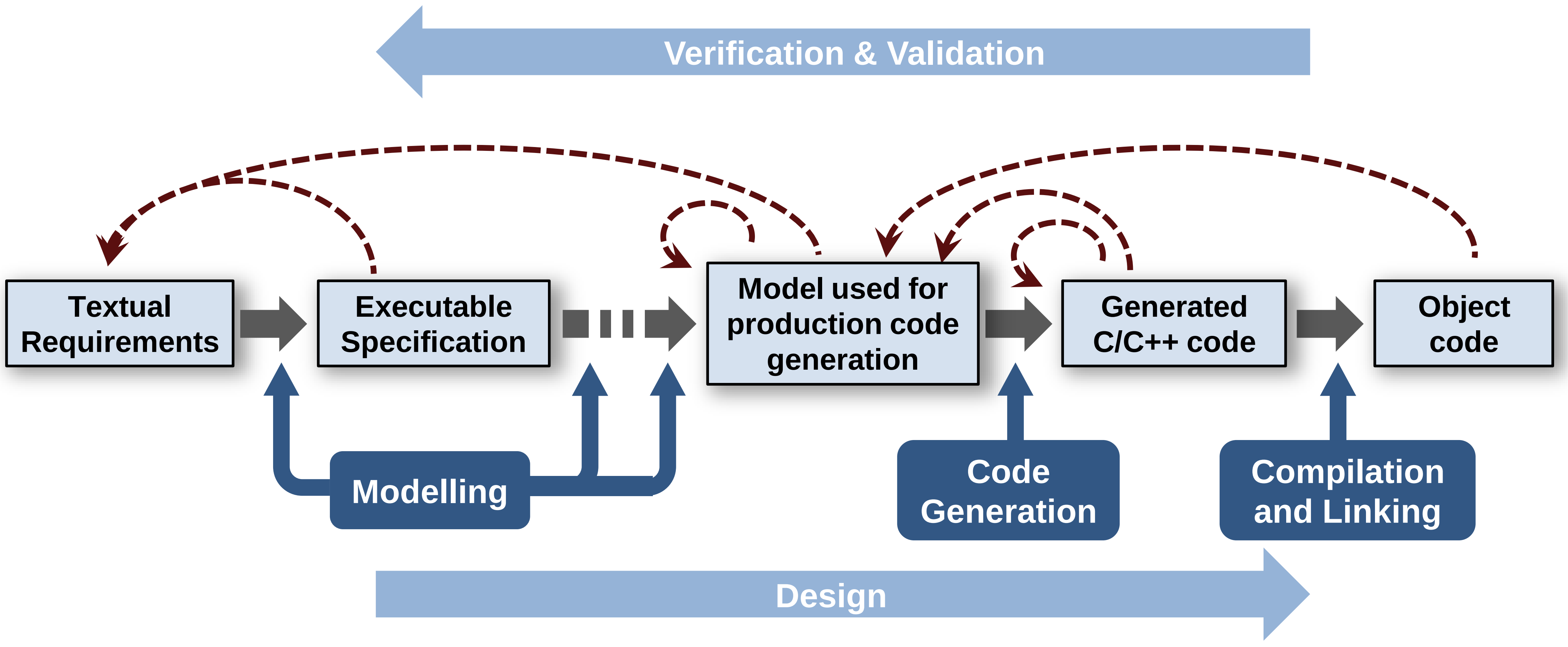
A good design workflow leads to a good design, but verification ***proves*** it!



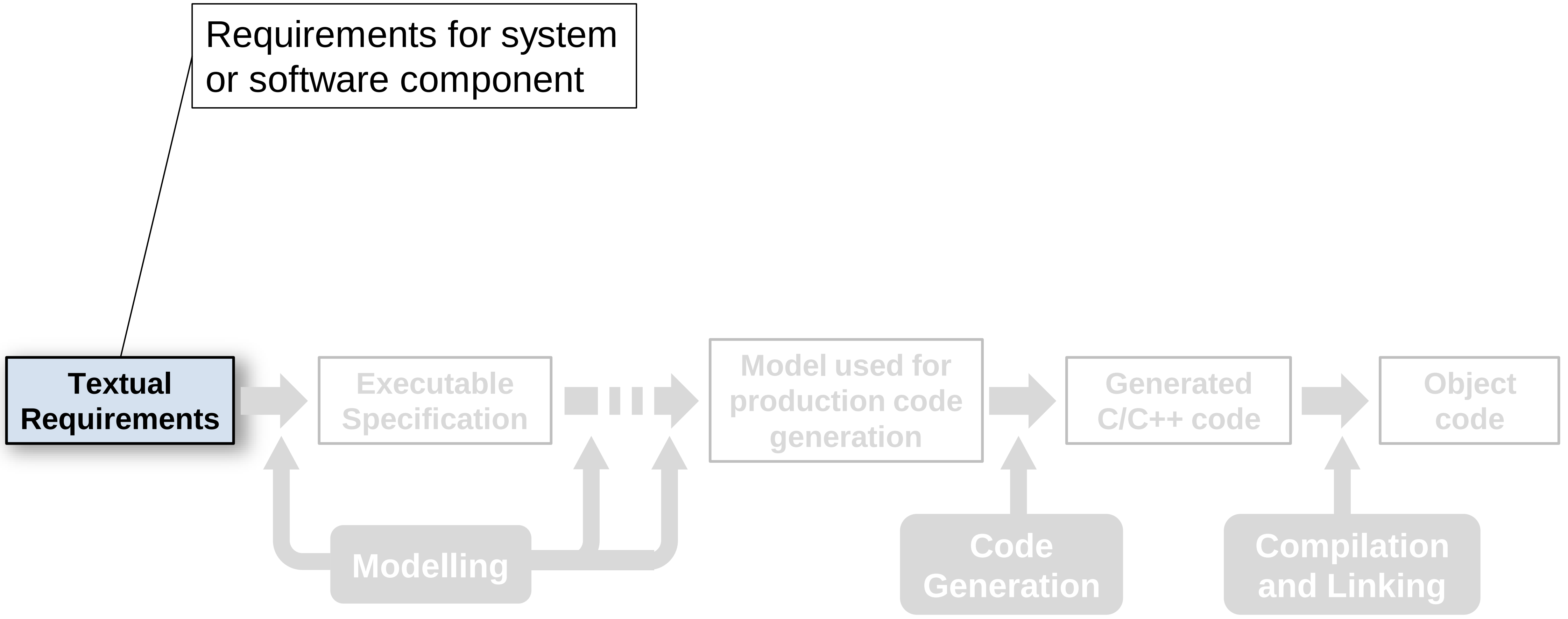
LEAR CORPORATION

The 100-day design cycle with MATLAB and Simulink

Model-Based Design and a Testing and Proving Workflow

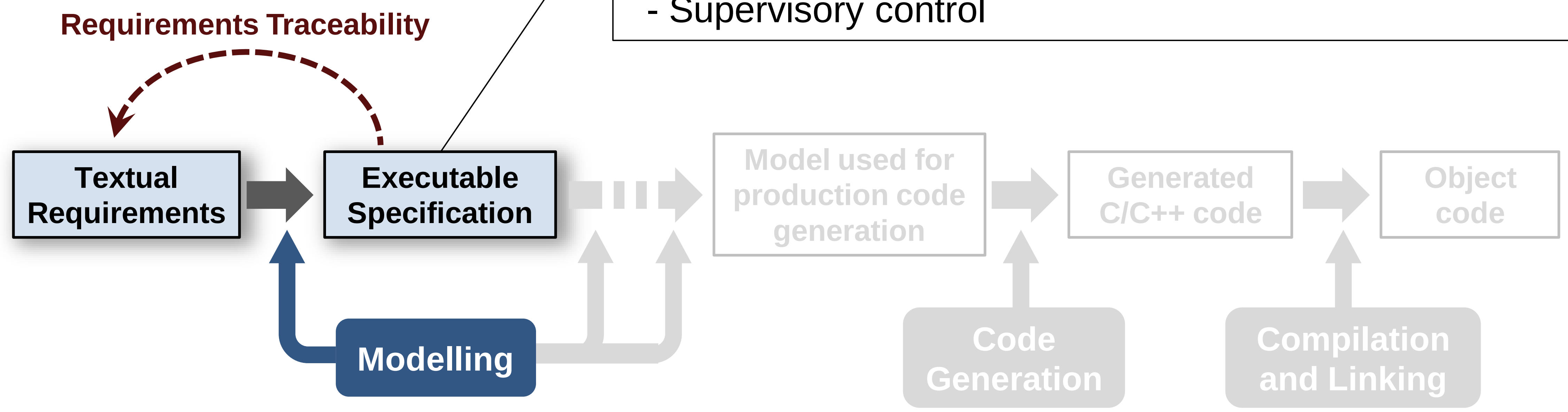


Start with Requirements



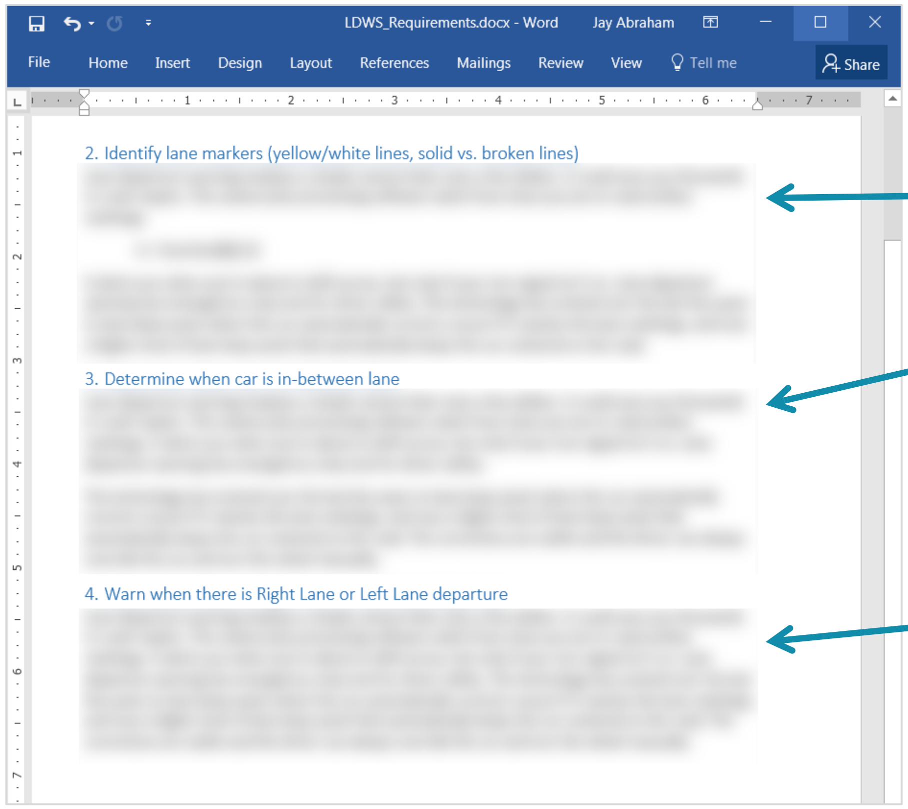
Transform Requirements into Executable Specifications

- Simulink models for continuous or discrete time behavior
  - Signal processing filters
  - Control algorithms
- Stateflow for logic and discrete events control
  - Start-up behavior, health checking
  - Supervisory control



Bi-directionally Trace Requirements

Textual Requirements



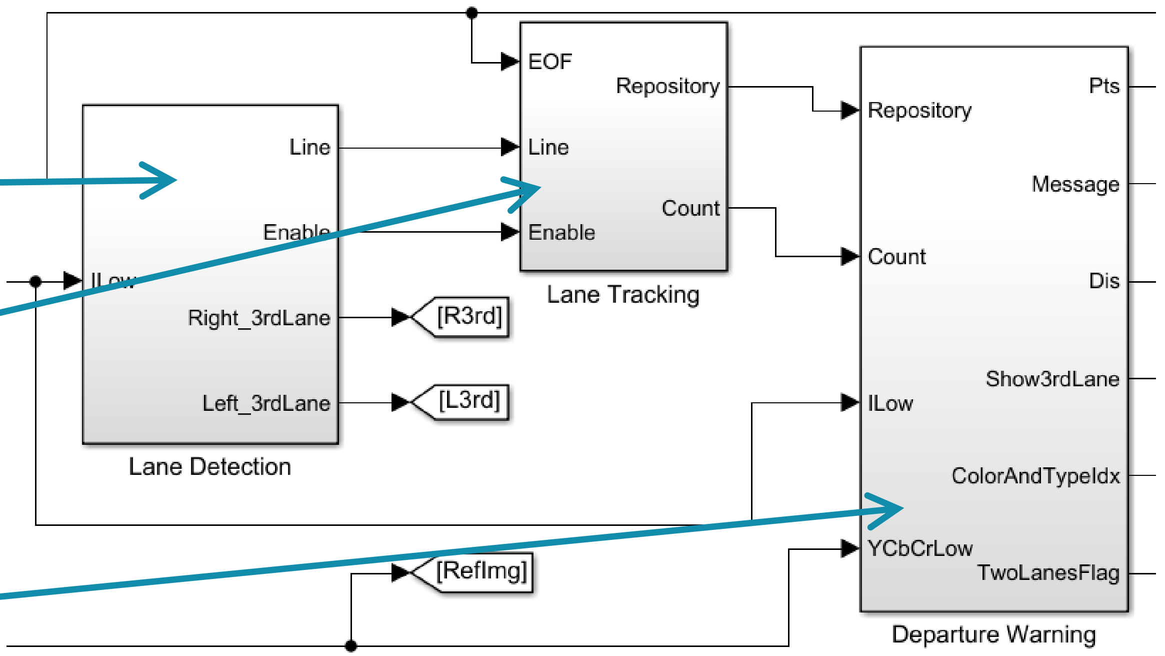
LDWS_Requirements.docx - Word

2. Identify lane markers (yellow/white lines, solid vs. broken lines)

3. Determine when car is in-between lane

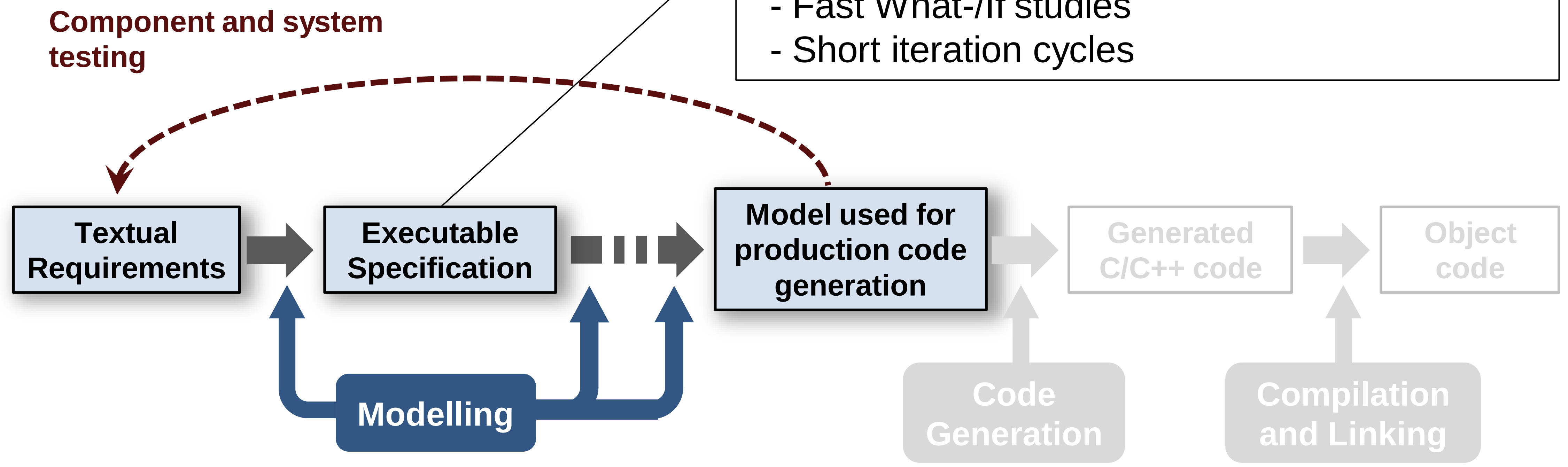
4. Warn when there is Right Lane or Left Lane departure

Design Model in Simulink



Test Early in Simulation

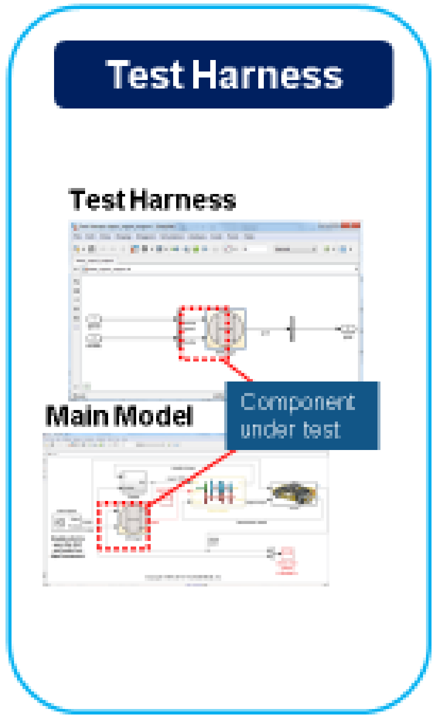
- Predict dynamic system behavior by simulation
  - System & environment models
  - Precision with floating point
- Use of simulation results for system design
  - Fast What-/If studies
  - Short iteration cycles

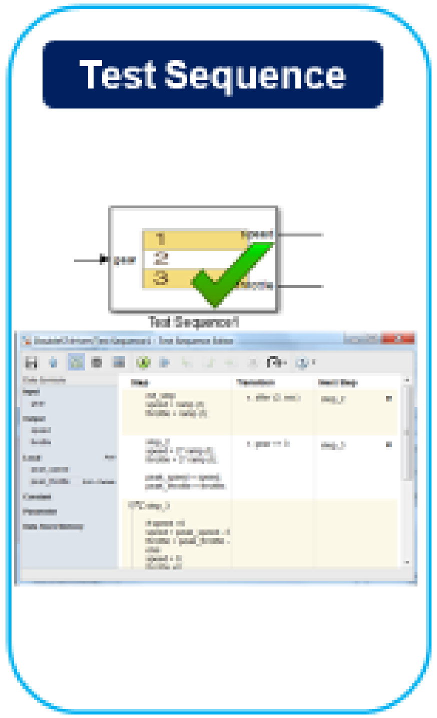


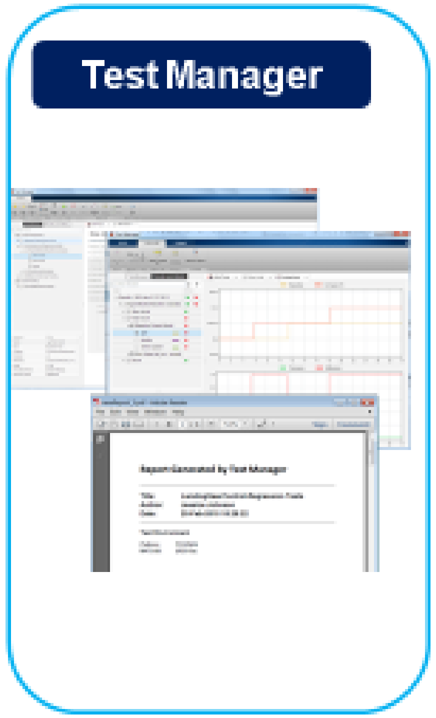
Functional Testing

- Author test-cases that are derived from requirements
  - Use test harness to isolate component under test
  - Test Sequence to create complex test scenarios

- Manage tests, execution, results
  - Re-use tests for regression
  - Automate in Continuous Integration systems such as Jenkins

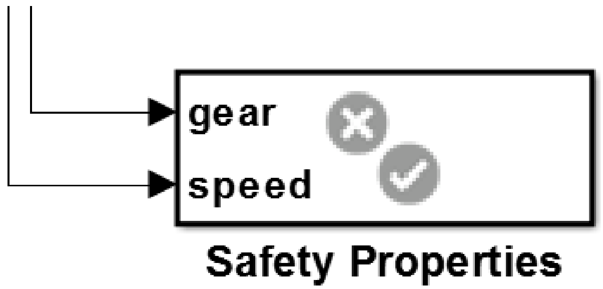






Simulink Test

Formal Verification: Proving Requirements



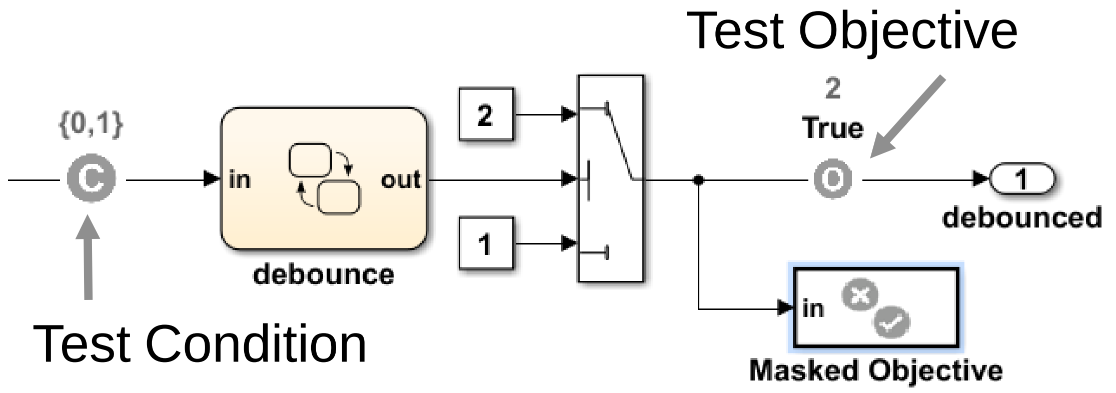
Checks that design meets requirements

- Condition 1: Gear 2 *always* engages
- Condition 2: Gear 2 *never* engages

Formal Verification: Test Case Generation

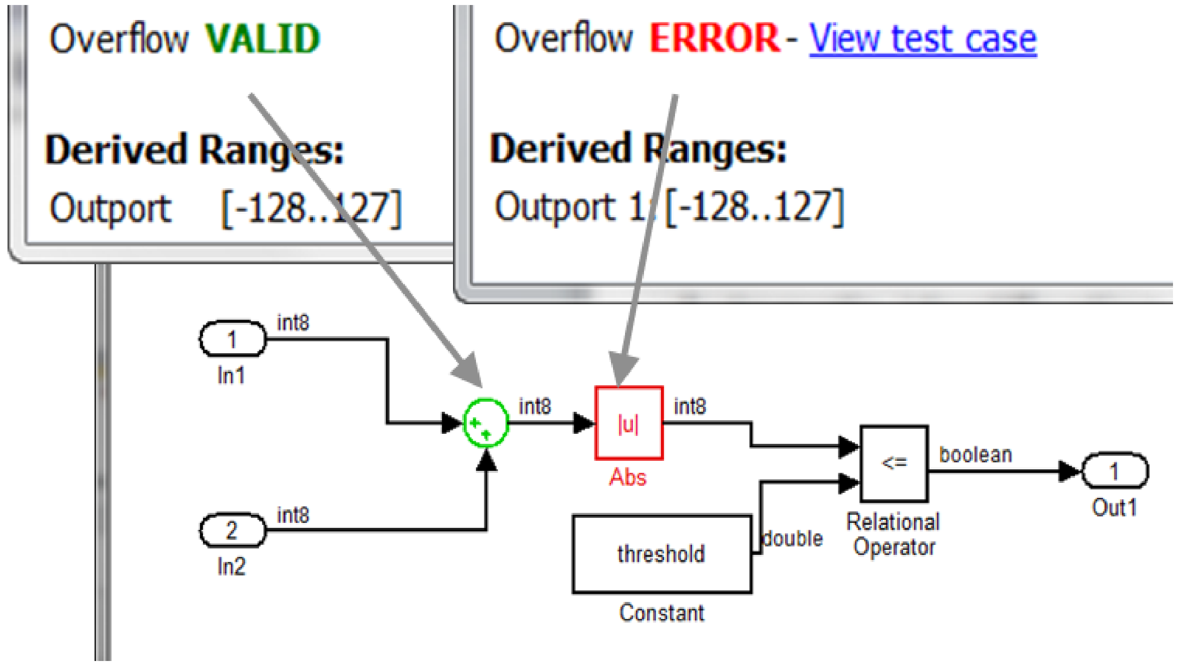
Automatically generate test cases for:

- Functional Requirements Testing
- Model Coverage Analysis



•The Test Objective block defines the values of a signal that a test case must satisfy.
•The Test Condition block constrains the values of a signal during analysis.

Formal Verification: Proving Robustness


Overflow VALID

Derived Ranges:
Outport [-128..127]

Overflow ERROR - View test case

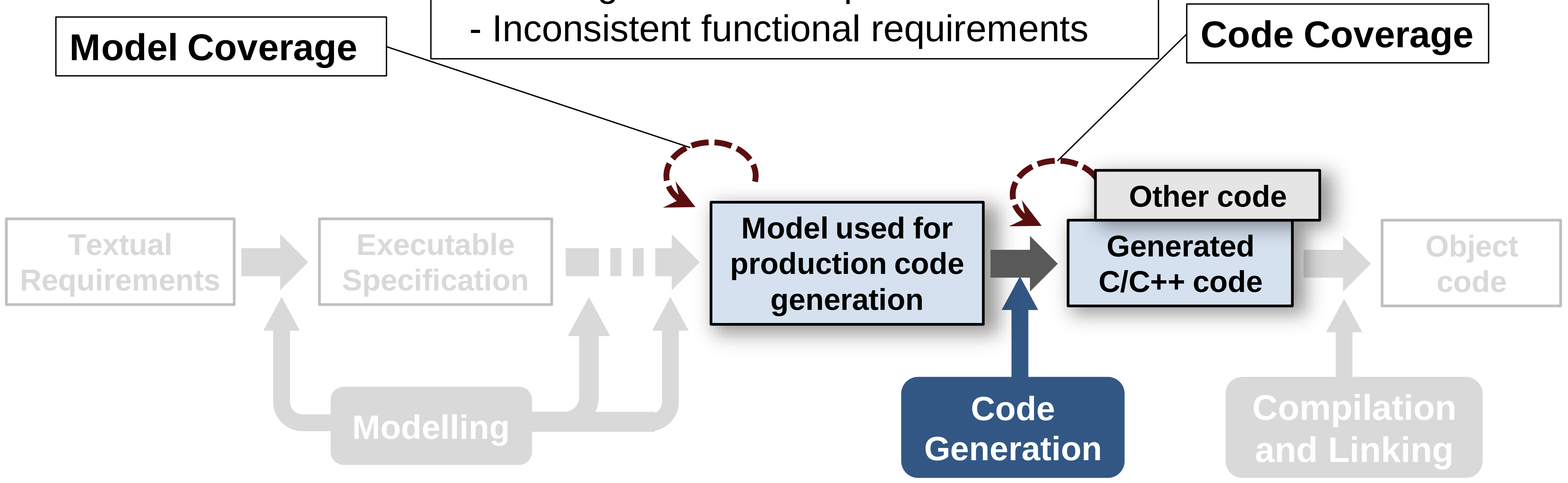
Derived Ranges:
Outport 1: [-128..127]

Detect overflows, divide by zero, and other robustness errors

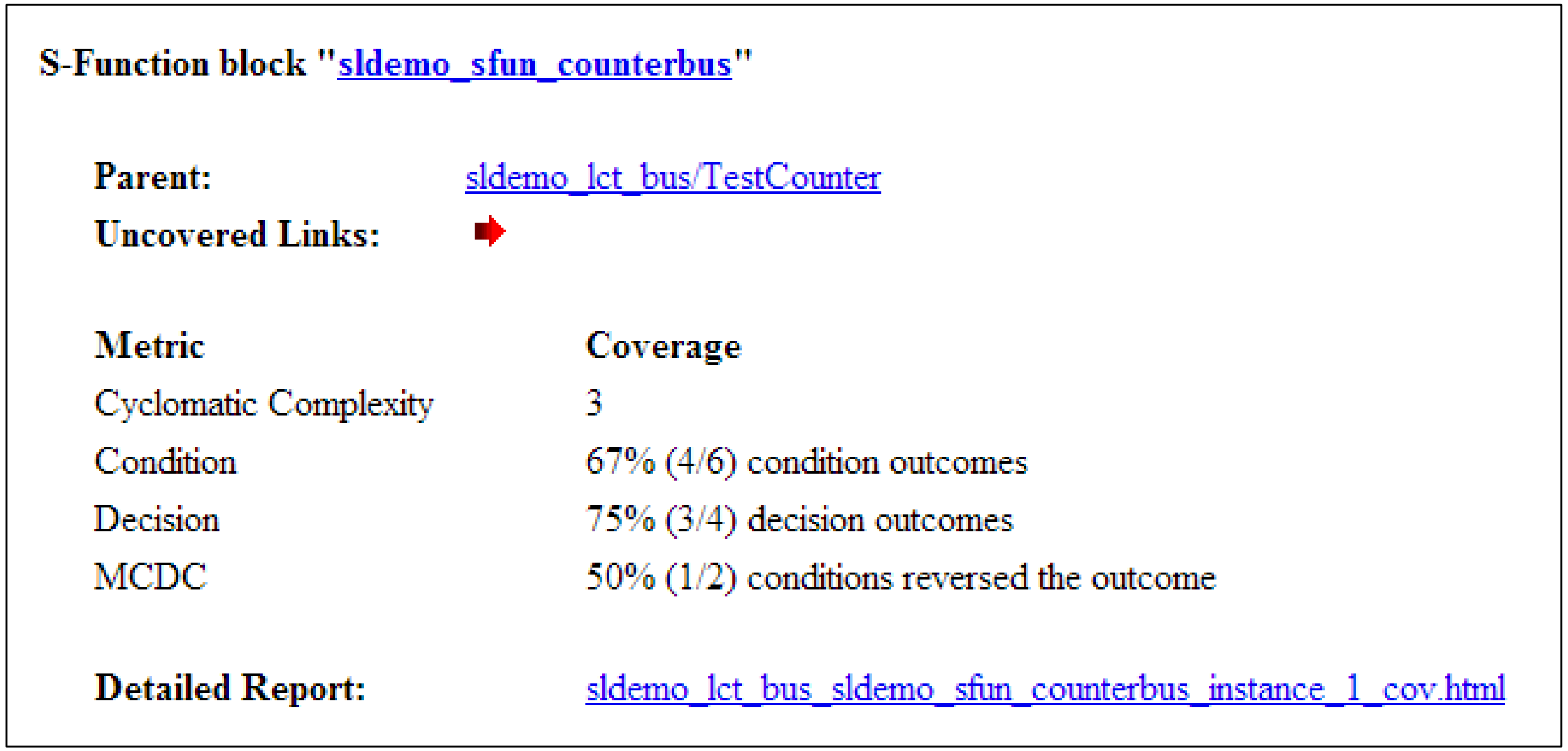
- Proven that overflow does NOT occur
- Proven that overflow DOES occur

Coverage Analysis

- Measure how much has been tested
  - Find untested design elements
  - Find dead logic and unreachable states
- Identify requirement issues early
  - Missing functional requirements
  - Inconsistent functional requirements



Coverage Analysis: also for self-written C/C++ in S-functions

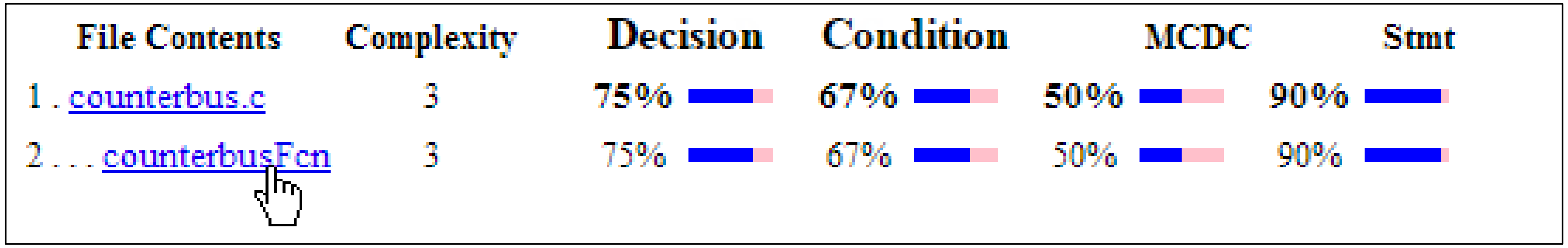


S-Function block "sldemo_sfun_counterbus"

Parent:	sldemo_lct_bus/TestCounter
Uncovered Links:	➡

Metric	**Coverage**
Cyclomatic Complexity	3
Condition	67% (4/6) condition outcomes
Decision	75% (3/4) decision outcomes
MCDC	50% (1/2) conditions reversed the outcome

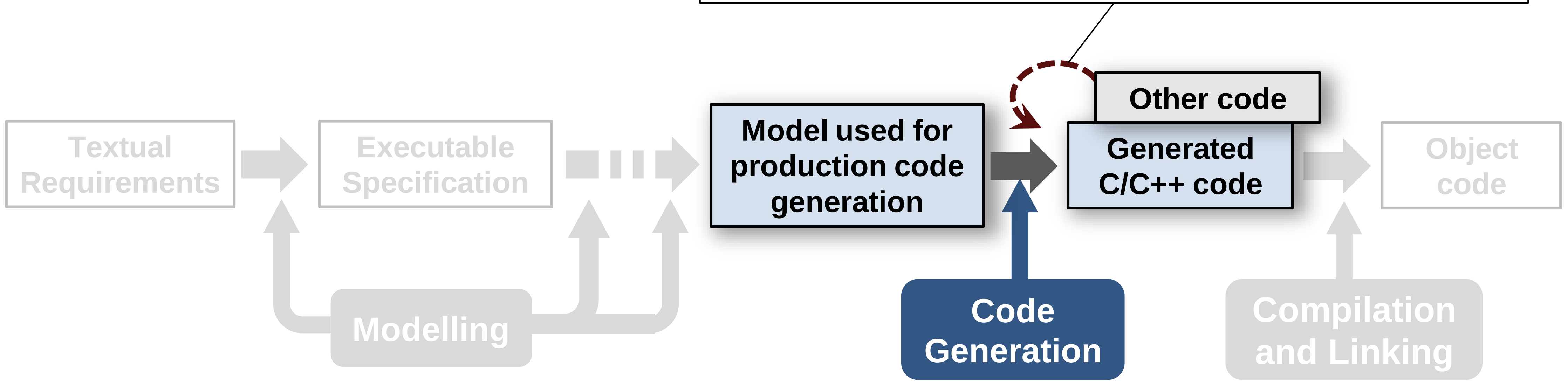
Detailed Report: sldemo_lct_bus_sldemo_sfun_counterbus_instance_1_cov.html



File Contents	Complexity	Decision	Condition	MCDC	Stmt
1 . counterbus.c	3	**75%**	**67%**	**50%**	**90%**
2 . . . counterbusFcn	3	75%	67%	50%	90%

Static Code Analysis

- Code metrics and standards
  - Comment density, cyclomatic complexity,…
  - MISRA and security standards compliance
  - Custom check authoring
- Bug Finding
  - Data and control flow
  - CERT C check for security vulnerabilities
- Code Proving
  - Formal Methods / Abstract Interpretation
  - No false negatives



Static Code Analysis: Proving vs. Bug Finding

Green implies absence of the most important classes of run-time errors:

Formally Proven

Green: reliable
safe pointer access

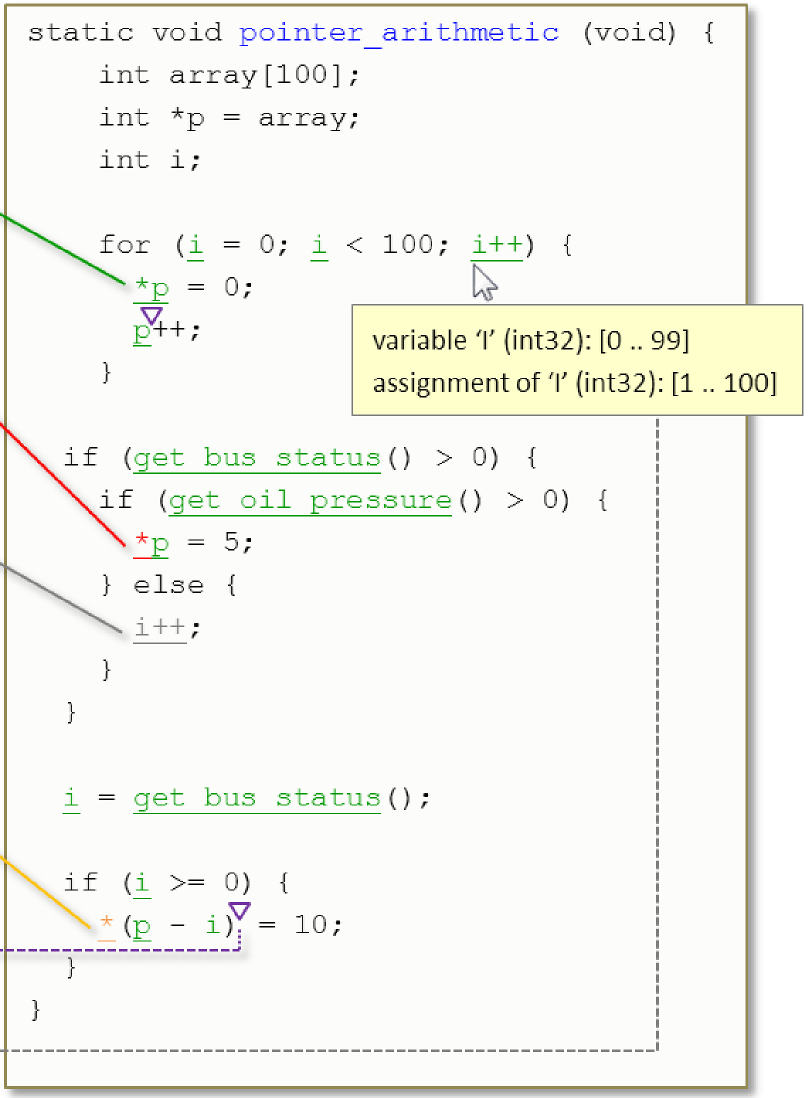
Red: faulty
out of bounds error

Gray: dead
unreachable code

Orange: unproven
may be unsafe for some conditions

Purple: violation
MISRA-C/C++ or JSF++ code rules

Range data
tool tip

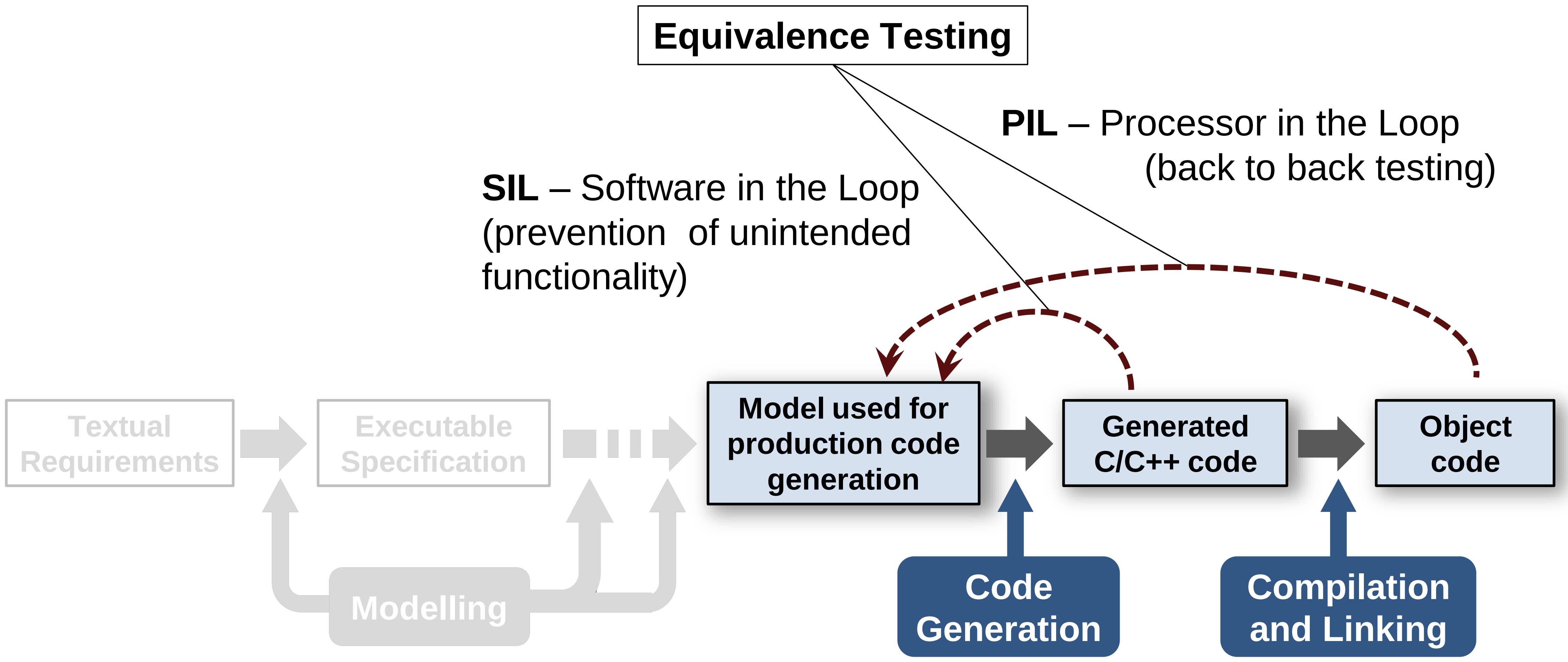


```
static void pointer_arithmetic (void) {
    int array[100];
    int *p = array;
    int i;

    for (i = 0; i < 100; i++) {
      *p = 0;
      p++;
    }

  if (get_bus_status() > 0) {
    if (get_oil_pressure() > 0) {
      *p = 5;
    } else {
      i++;
    }
  }

  i = get_bus_status();

  if (i >= 0) {
    *(p - i) = 10;
  }
}
```

variable ‘I’ (int32): [0 .. 99]
assignment of ‘I’ (int32): [1 .. 100]

Polyspace Code Prover

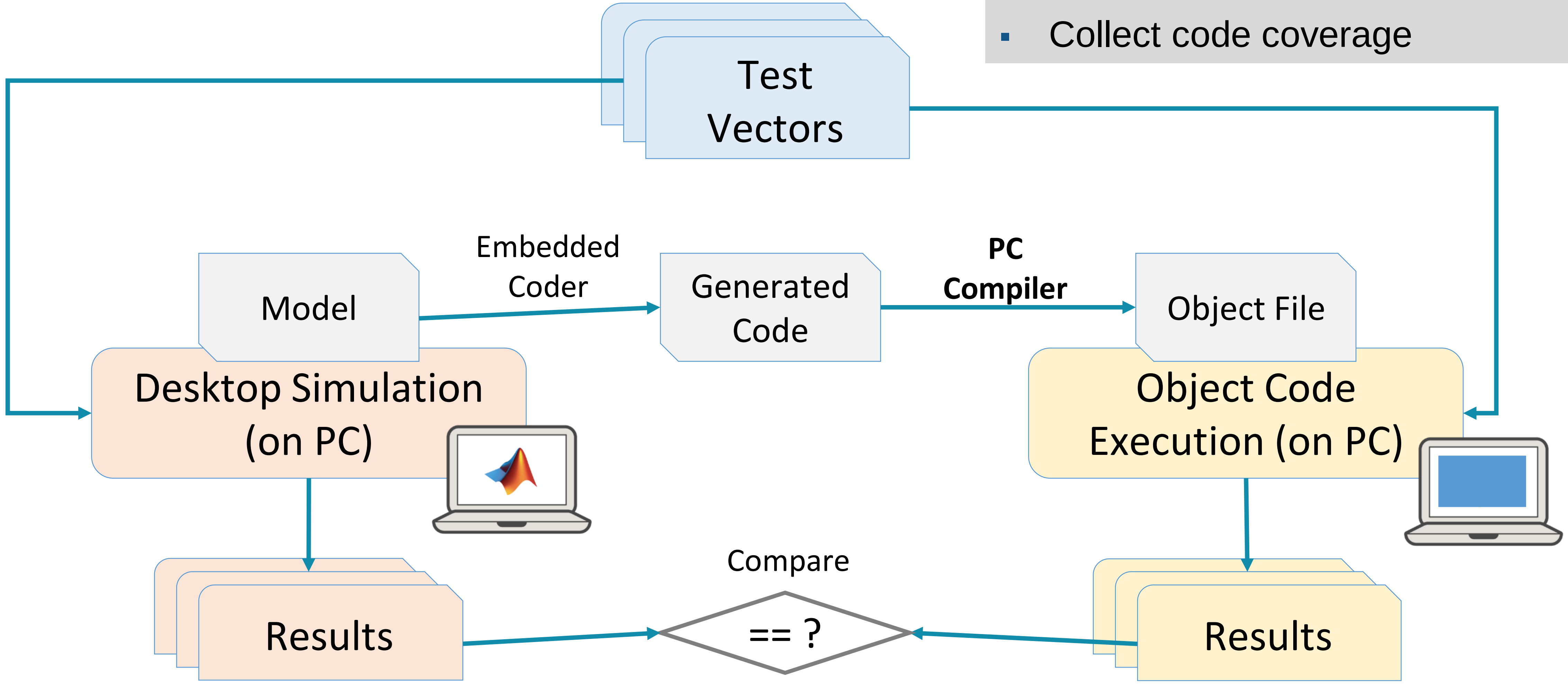
Equivalence Testing (Back to Back Testing)



Equivalence Testing
PIL – Processor in the Loop (back to back testing)
SIL – Software in the Loop (prevention of unintended functionality)
Textual Requirements
Executable Specification
Modelling
Model used for production code generation
Generated C/C++ code
Object code
Code Generation
Compilation and Linking

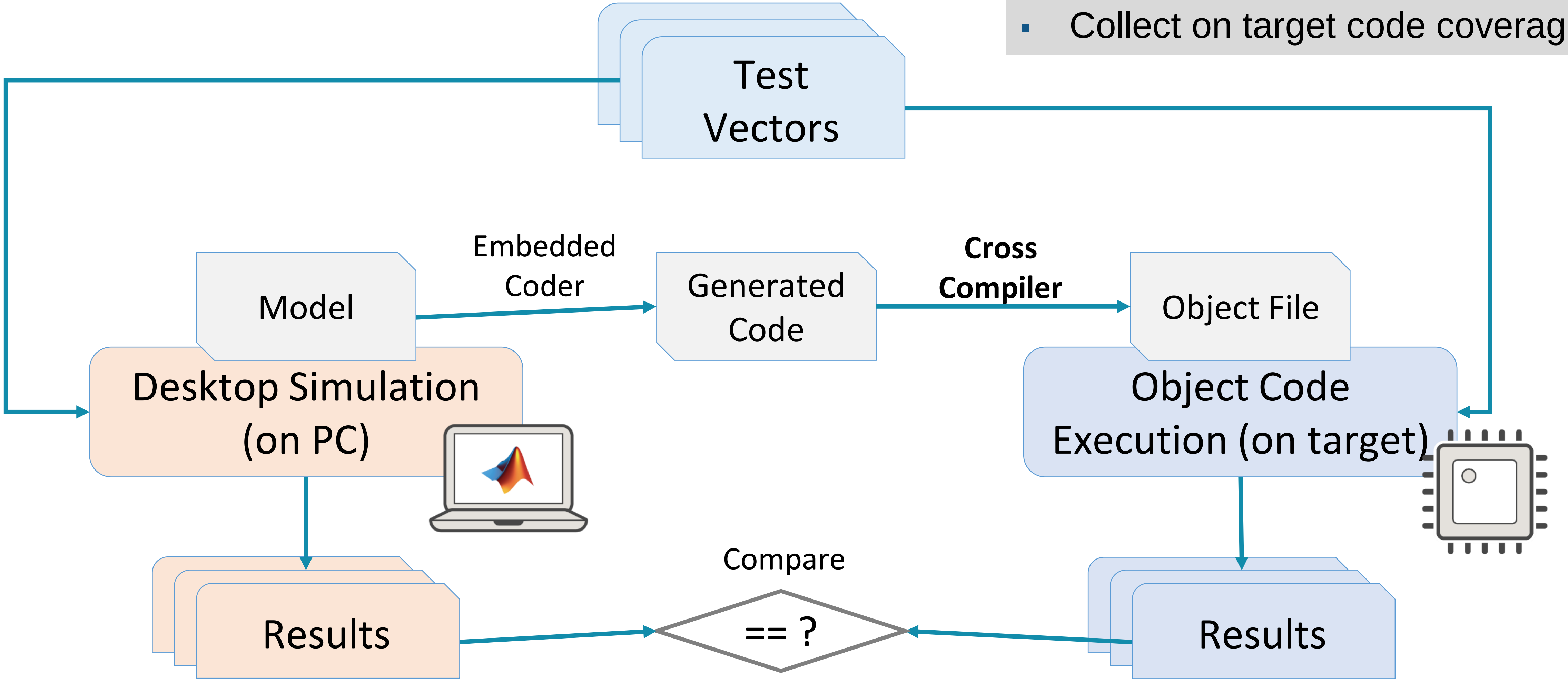
Software In the Loop (SIL) Testing

- Show equivalence, model to code
- Assess code execution time
- Collect code coverage

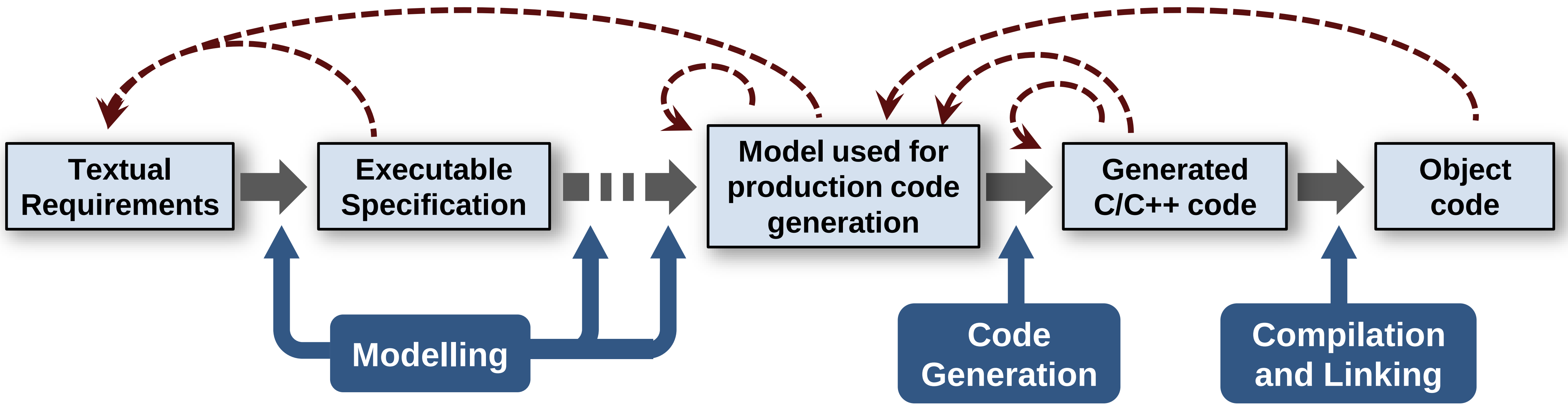


Processor In the Loop (PIL) Testing

- Verify numerical equivalence
- Assess target execution time
- Collect on target code coverage



Model-Based Design Reference Workflow (IEC 61508-3)



Automotive (ISO 26262)

Medical (IEC 62304)

Aerospace (DO-178)

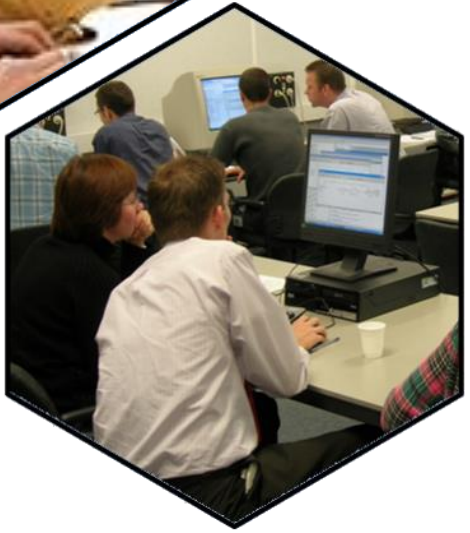
Rail (EN 50128)

Industrial (IEC 61508)

Training

Public

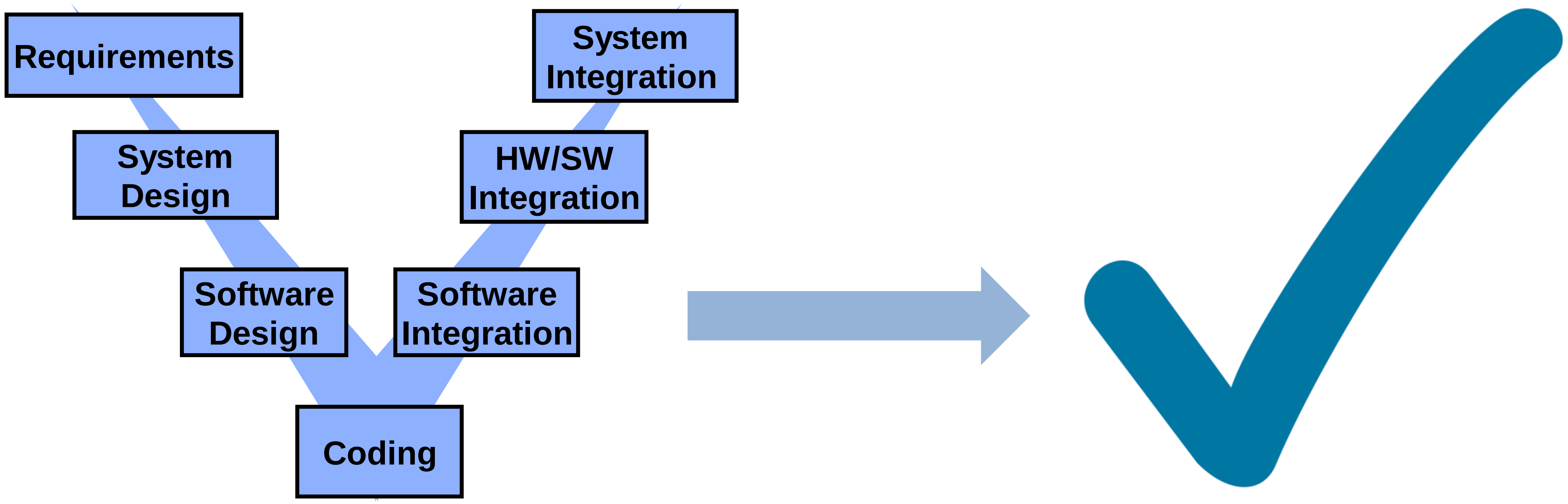
On-Site



Verification and Validation of Simulink Models
Testing Generated Code in Simulink
Polyspace for C/C++ Code Verification
Polyspace Bug Finder for C/C++ Code Analysis

Key Takeaway

A good design workflow leads to a good design, but verification ***proves*** it!


Requirements
System Design
Software Design
Coding
Software Integration
HW/SW Integration
System Integration