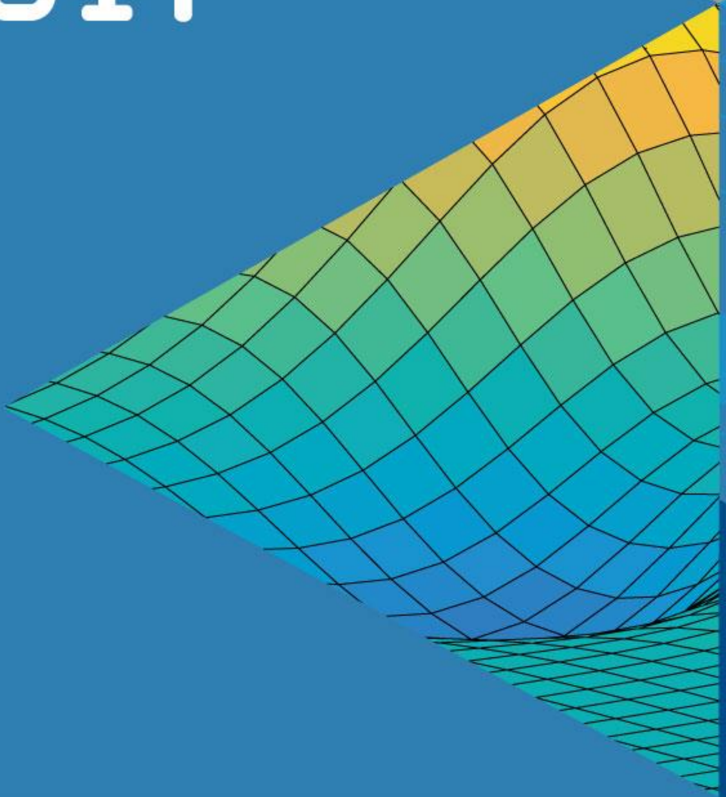
MATLAB EXPO 2017
KOREA

4월 27일, 서울

등록 하기 matlabexpo.co.kr

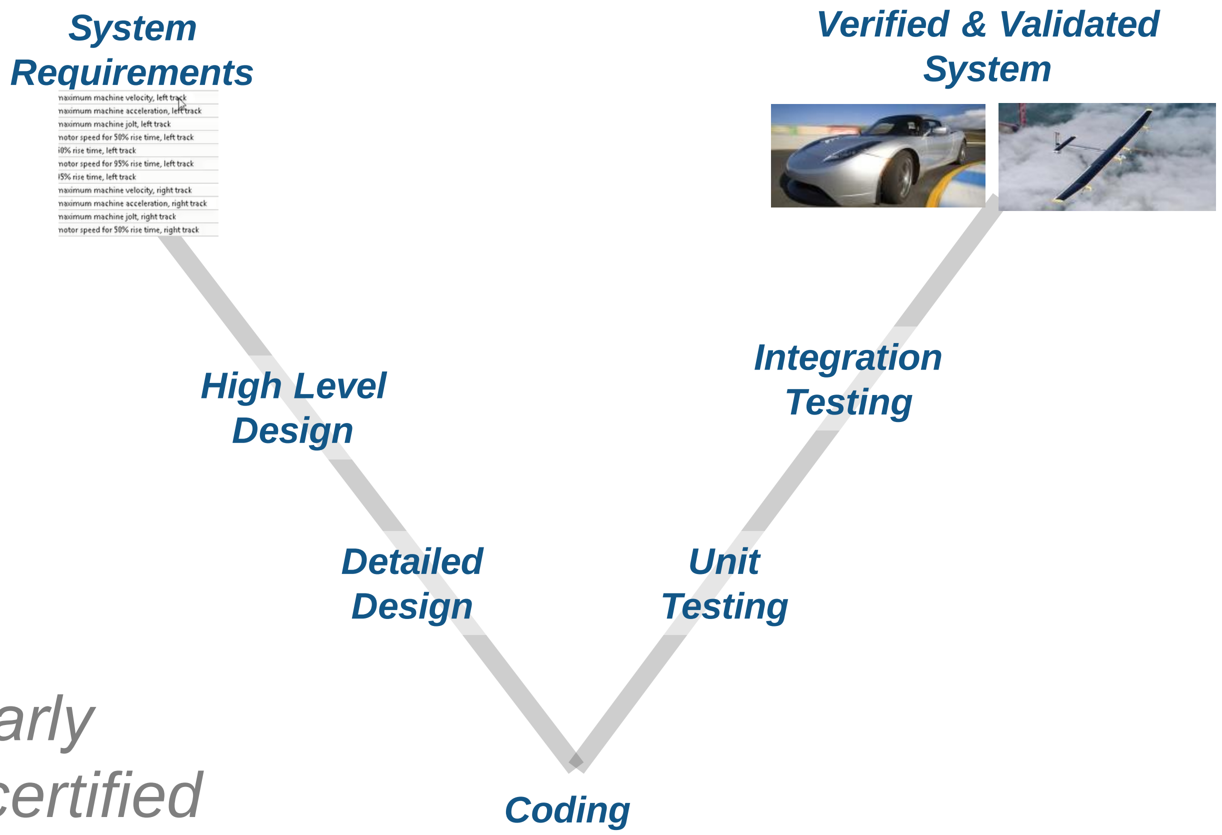


Verification Techniques in Model-Based Design for High Integrity System

Young Joon Lee
Principal Application Engineer

Key Takeaways

1. Find bugs early, develop high quality software
2. Replace manual verification tasks with workflow automation
3. Learn about reference workflow that conforms to safety standards



naximum machine velocity, left track
naximum machine acceleration, left track
naximum machine jolt, left track
notor speed for 50% rise time, left track
0% rise time, left track
notor speed for 95% rise time, left track
5% rise time, left track
naximum machine velocity, right track
naximum machine acceleration, right track
naximum machine jolt, right track
notor speed for 50% rise time, right track

"Reduce costs and project risk through early verification, shorten time to market on a certified system, and deliver high-quality production code that was first-time right"   Michael Schwarz, ITK Engineering

Safety of Electronic Systems

- Critical functionality in industries such as Automotive, Aerospace, Medical, Industrial Automation
- Real-time operation
  - Compute time lag cannot be tolerated
- Predictable behavior
  - No unintended functionality
- Must be robust
  - Program crash or reboot not allowed

Role of Certification Standards

- ISO 26262 (Automotive)
  - Defines functional safety for automotive electronic systems
  - Automotive Safety Integrity Level ASIL QM, A to D (least to most; derived from severity, controllability, probability)
  - ISO 26262-6 pertains to software development, verification, and validation

- DO-178 (Avionics)
  - Guidelines for the safety of software in certain airborne systems
  - Level A to E (most critical to least)
  - Verification activities include review of requirements and code, testing of software, code coverage

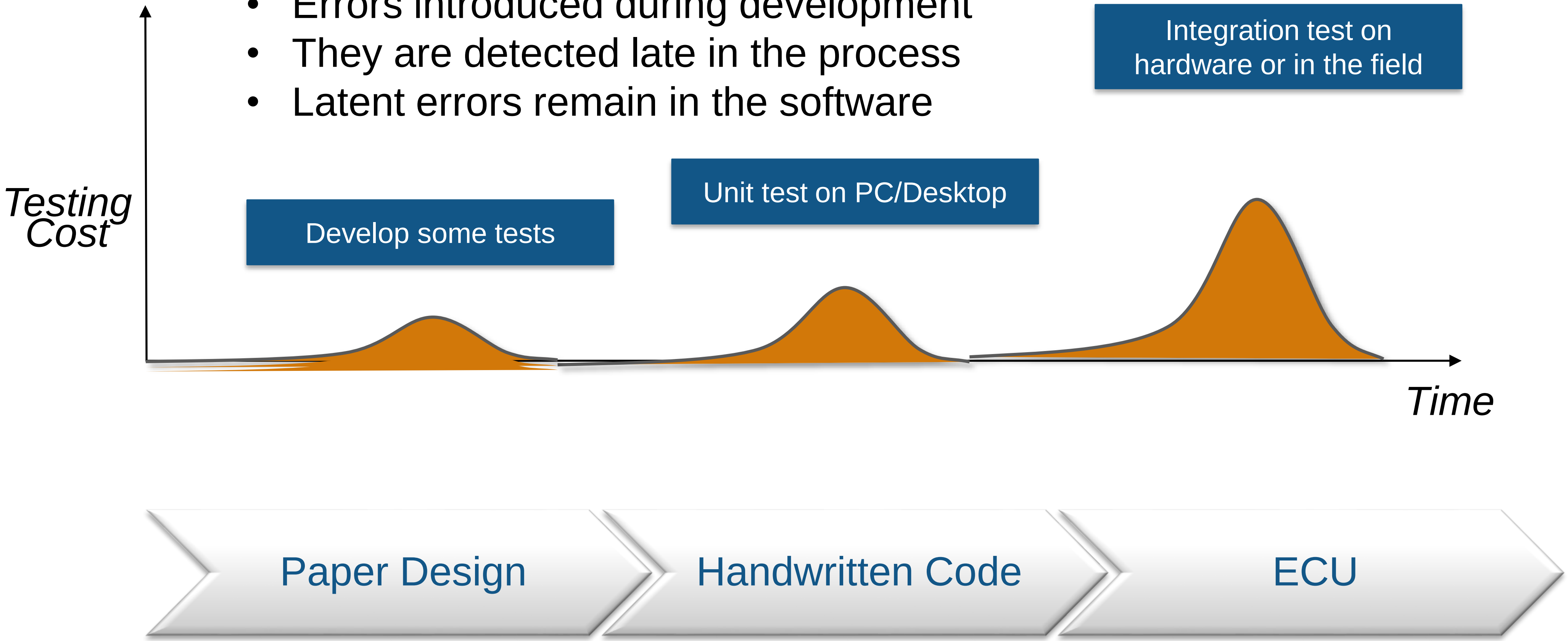
- IEC 62304 (Medical Device)
  - Describes software development and maintenance processes for medical device software
  - Safety levels Class A to C (least critical to most)
  - Identifies various verification and testing activities

Traditional Development Process

- Start with a paper design
- Manually determine system architecture
- Identify algorithms for the application
- Start writing code for the algorithms
- Develop testing platform to unit test algorithms
- Manually unit test the code with the testing platform
- Test the design with the real hardware and code
- Find bugs, fix bugs, repeat … ***very painful !!!***

Problems with Traditional Development Process

- Errors introduced during development
- They are detected late in the process
- Latent errors remain in the software



Addressing Design and Development Challenges

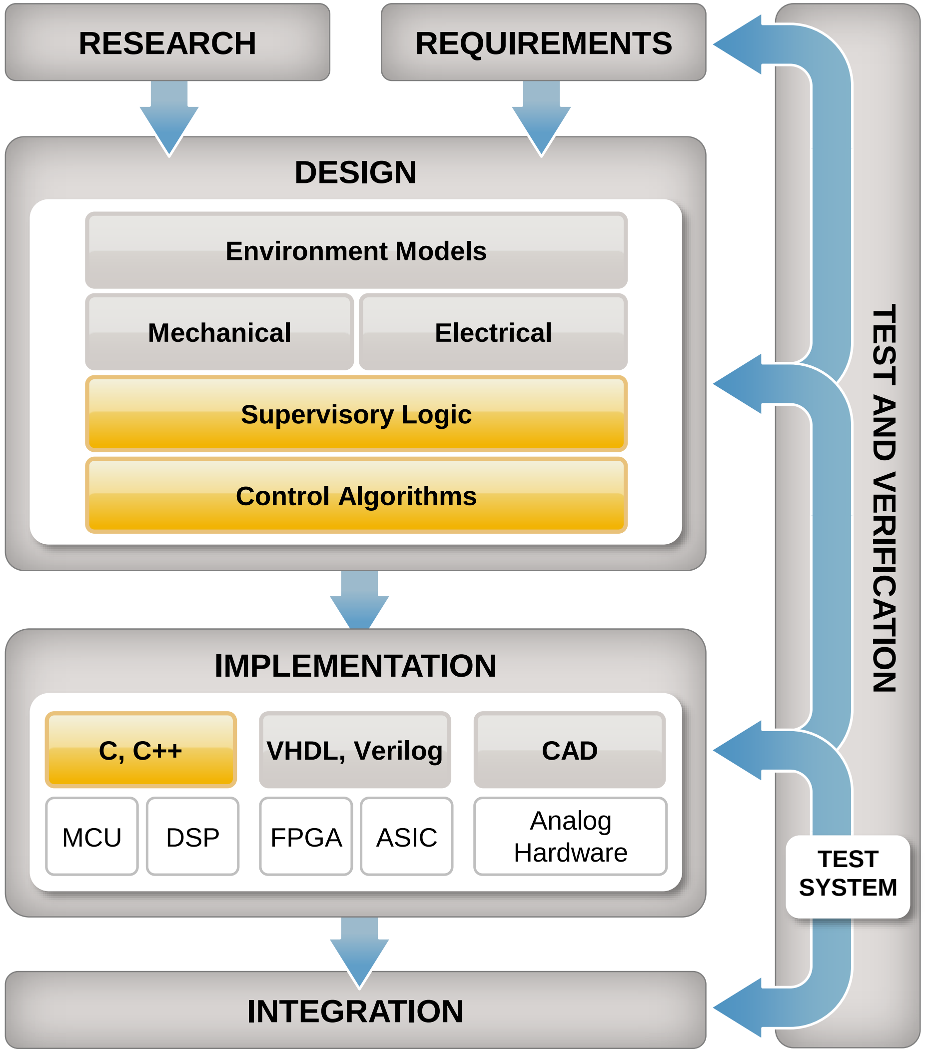
> ***It is easier and less expensive to fix design errors early in the process when they happen.***

Model-Based Design enables:

1. *Early testing to increase confidence in your design*
2. *Delivery of higher quality software for production use*
3. *Credits and artifacts for certification to satisfy safety standards*

Model Based Design

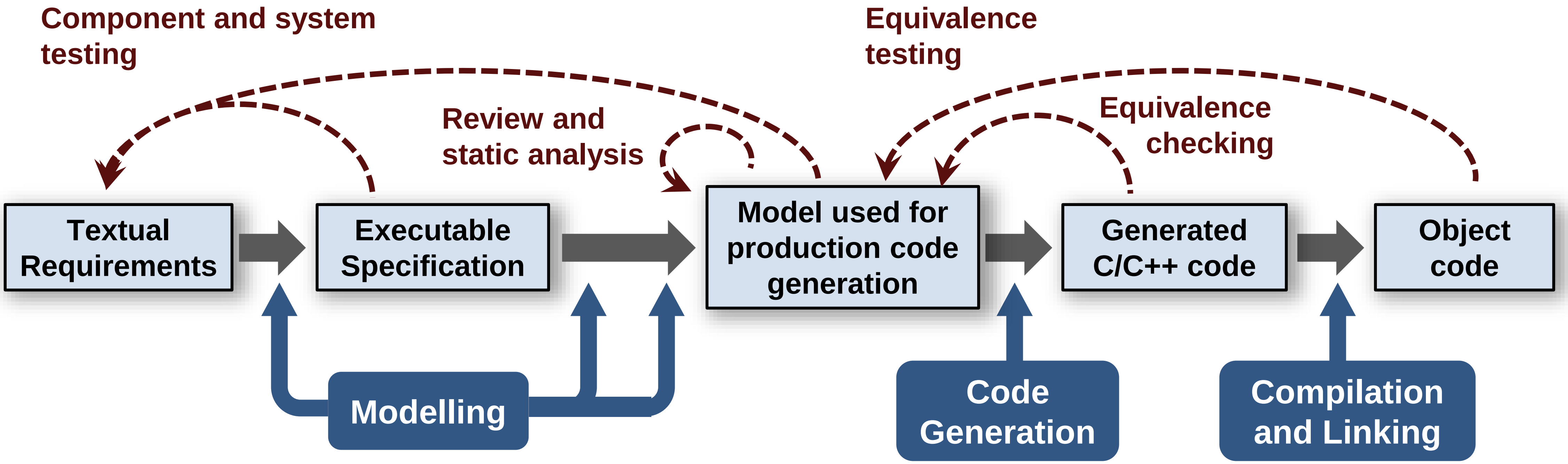
- Modeling
  - Model algorithms and environment
  - Explore design alternatives and options
- Simulation
  - Design exploration with simulation
  - Find issues early, on your desktop PC
- Production code
  - Code generated automatically from model
  - Early verification for high quality code



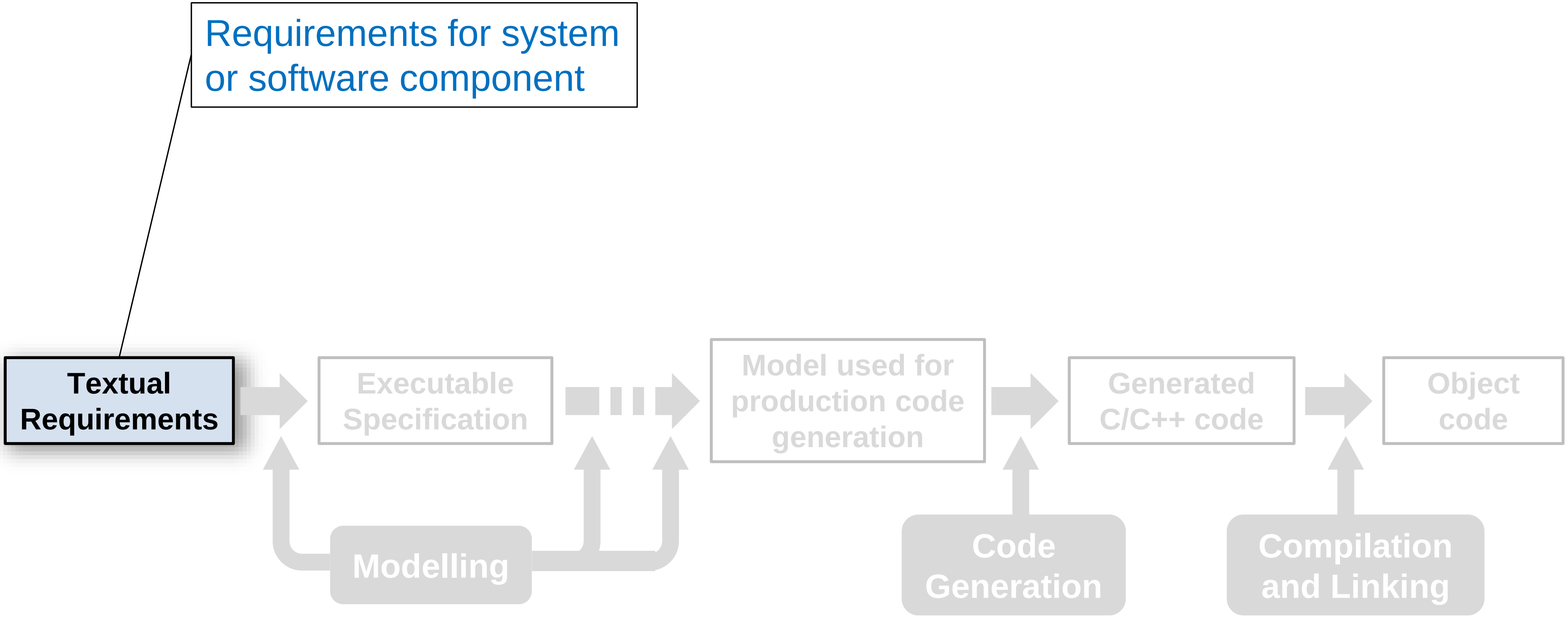
Reference Verification and Validation Workflow

Reference Verification and Validation Workflow

- Certifiable Model-Based Design Workflow to develop critical embedded software
- Reviewed and approved by TÜV SÜD certification authority



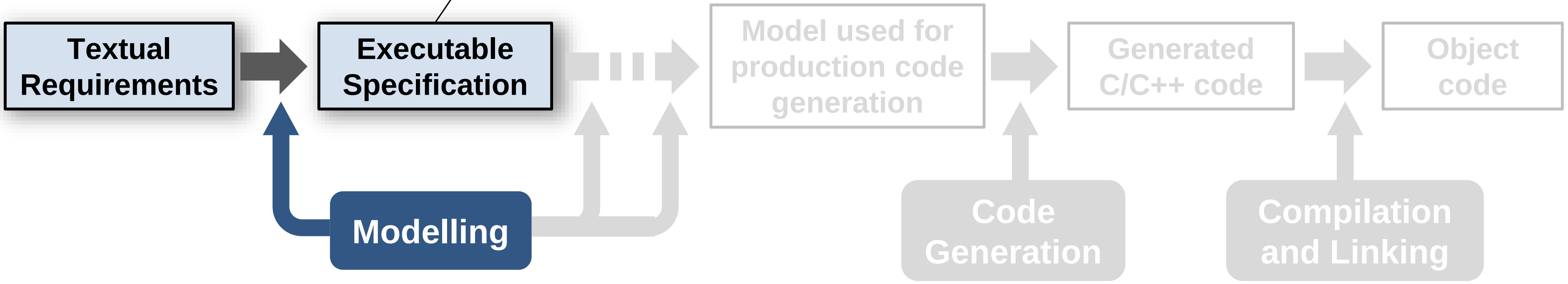
Reference Verification and Validation Workflow



Requirements for system or software component

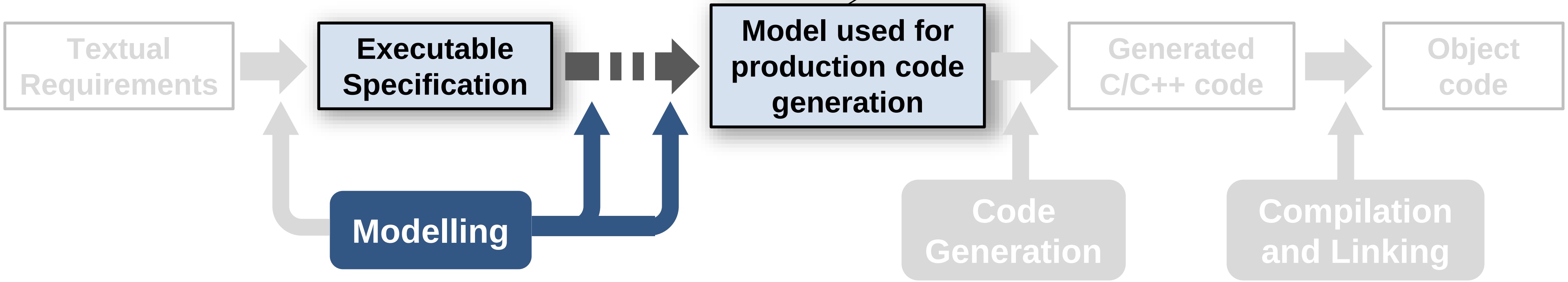
Reference Verification and Validation Workflow

- 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



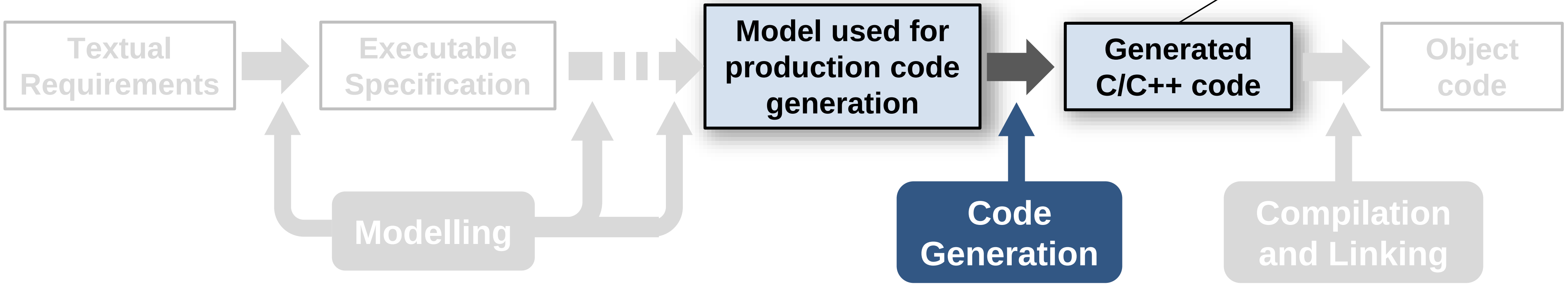
Reference Verification and Validation Workflow

- Model tuned for target processor
  - Fixed point mathematics, real-time behavior
- Configure for production use
  - Support for standards (AUTOSAR, ASAP2)

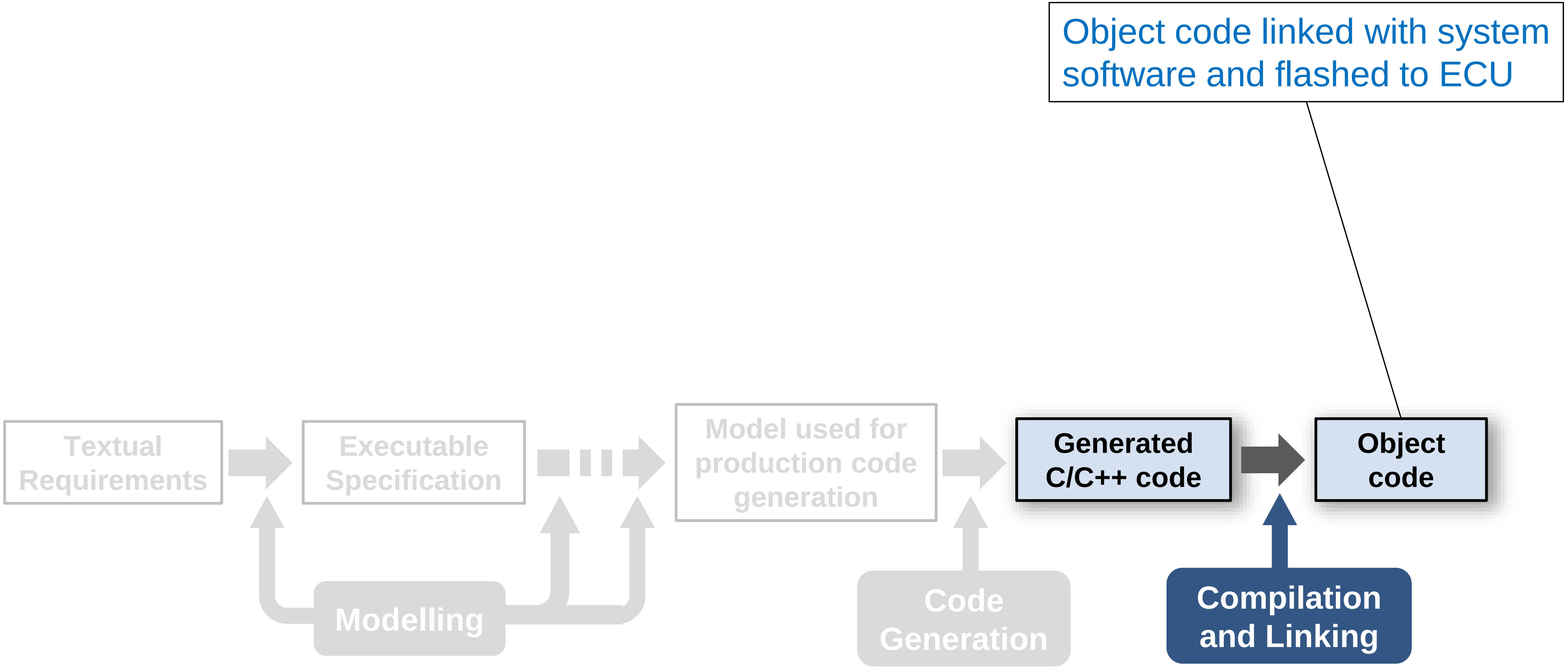


Reference Verification and Validation Workflow

- Automatically generated code for target processor
  - Optimized, efficient C/C++ code
- Fine grain control of generated code
  - Files, functions, data



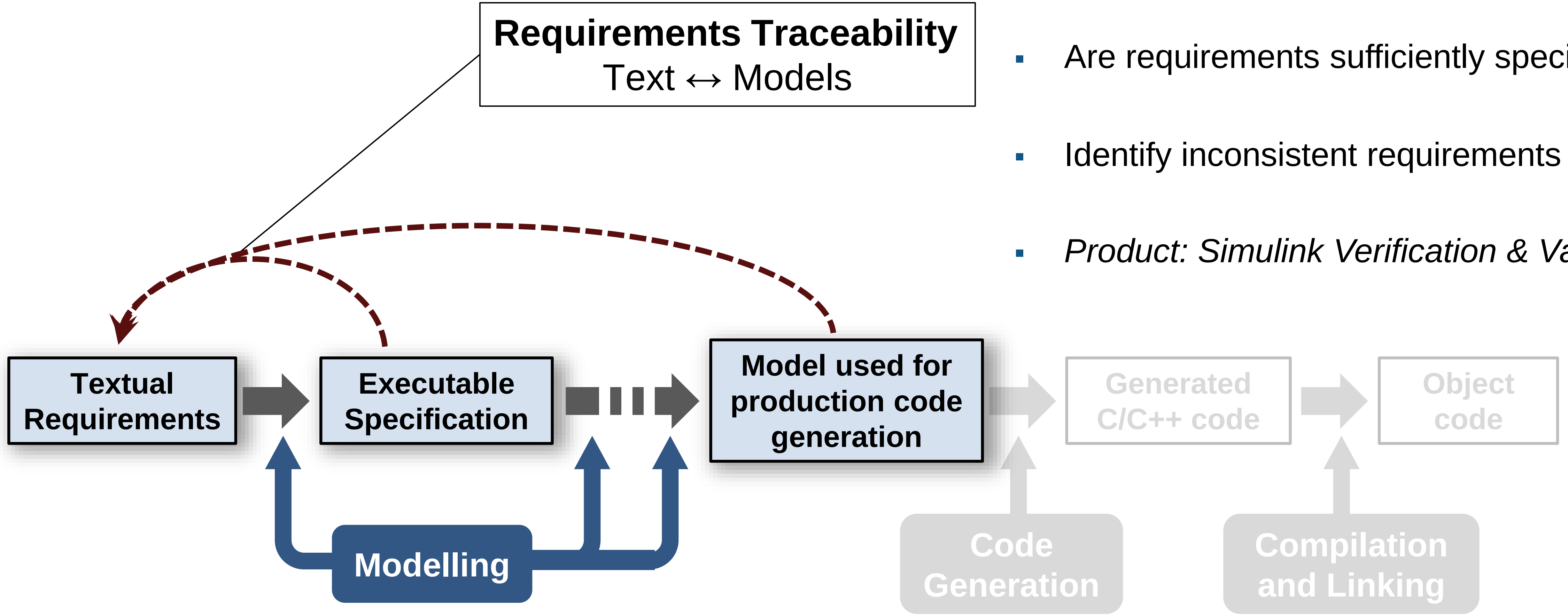
Reference Verification and Validation Workflow



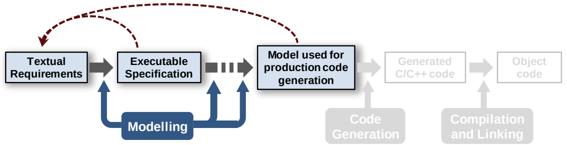
Verification and Validation Tasks and Activities

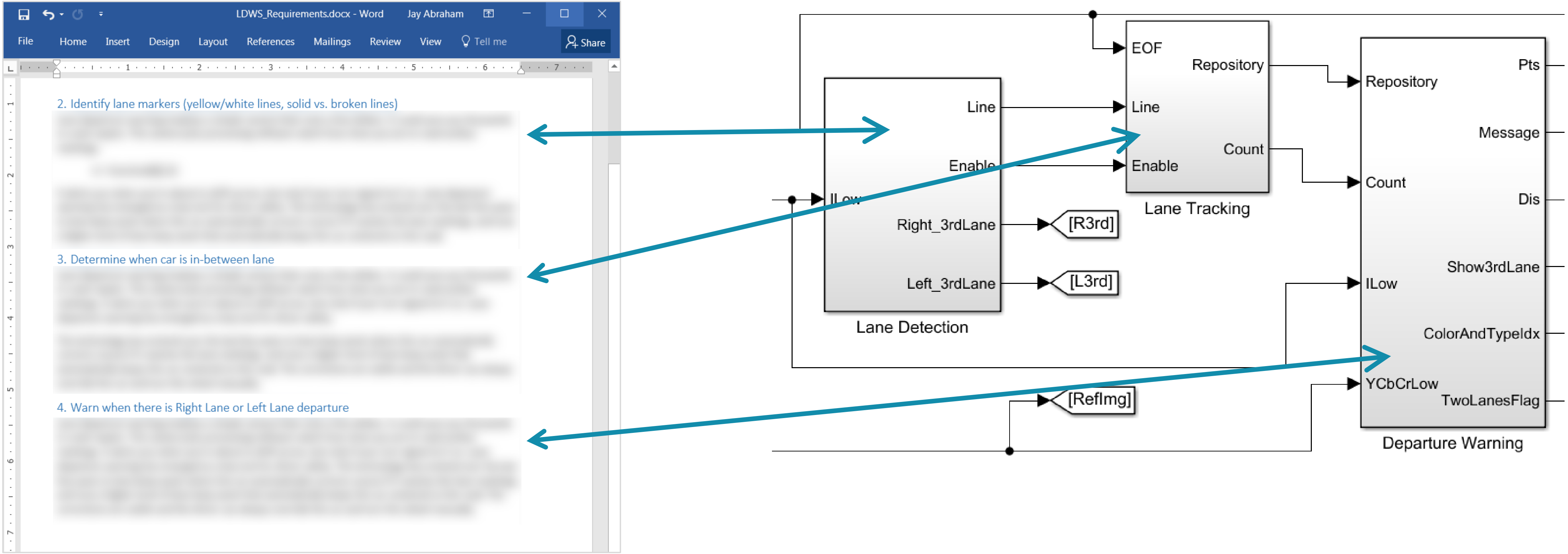
Verification and Validation Tasks and Activities

- Find missing or incomplete requirements
- Are requirements sufficiently specified
- Identify inconsistent requirements
- *Product: Simulink Verification & Validation*



Bi-directionally Trace Requirements





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

3. Determine when car is in-between lane

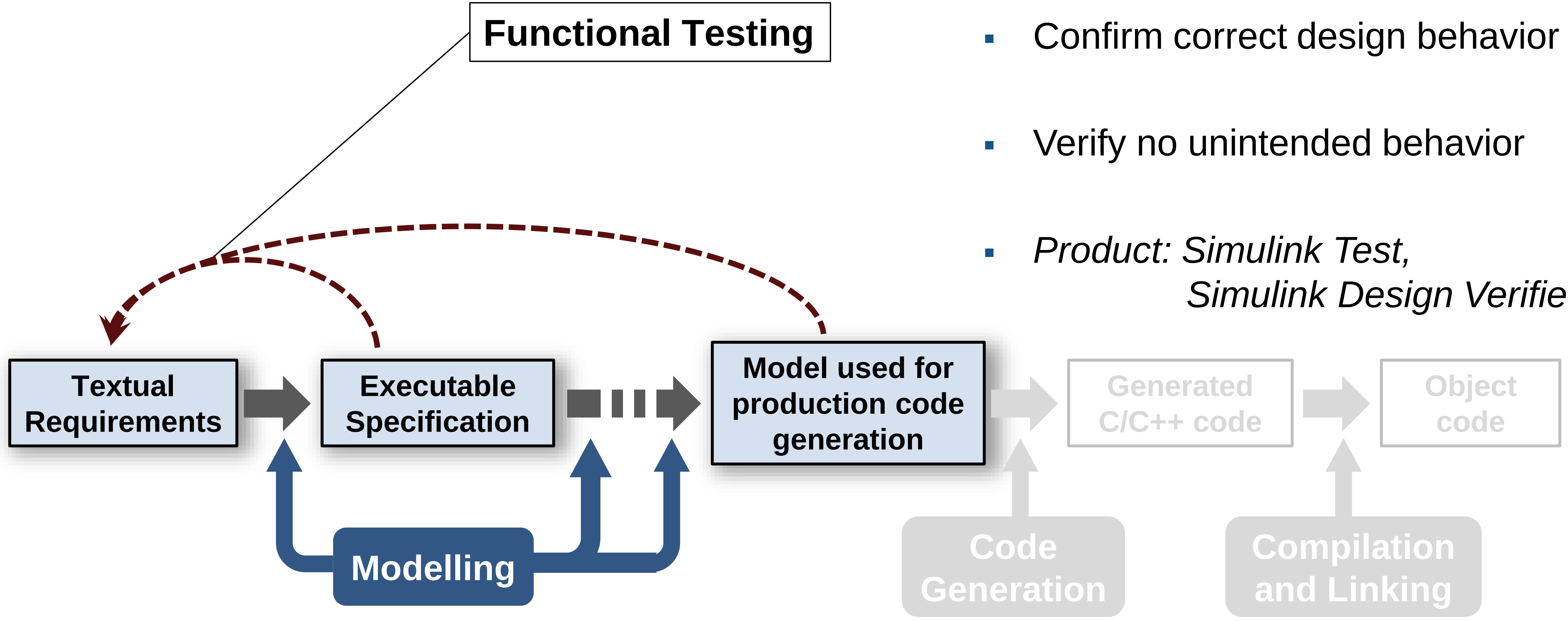
4. Warn when there is Right Lane or Left Lane departure

Textual Requirements

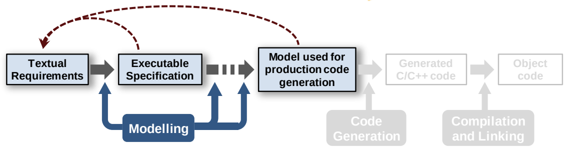
Design Model

Verification and Validation Tasks and Activities

- Does design meet requirements
- Confirm correct design behavior
- Verify no unintended behavior
- *Product: Simulink Test,*
  Simulink Design Verifier



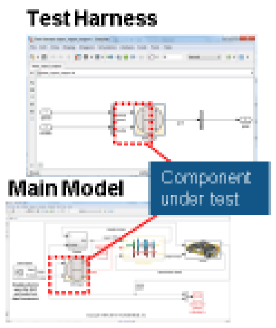
Functional Testing



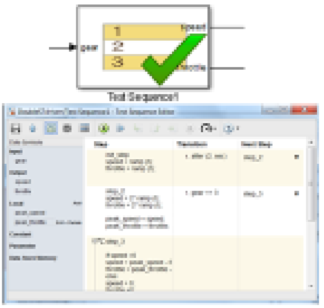
- Functional testing process
  - Author test-cases (derived from requirements)
  - Use formal verification to auto generate tests (more on this next)
  - Execute tests across design environments (with test iterations)
  - Monitor test verdicts (pass/fail)

- Product: Simulink Test
  - Test harness to isolate component under test
  - Author complex test scenarios with Test Sequer
  - Manage tests, execution, and results

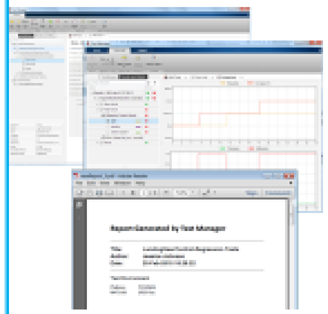
Test Harness



Test Sequence

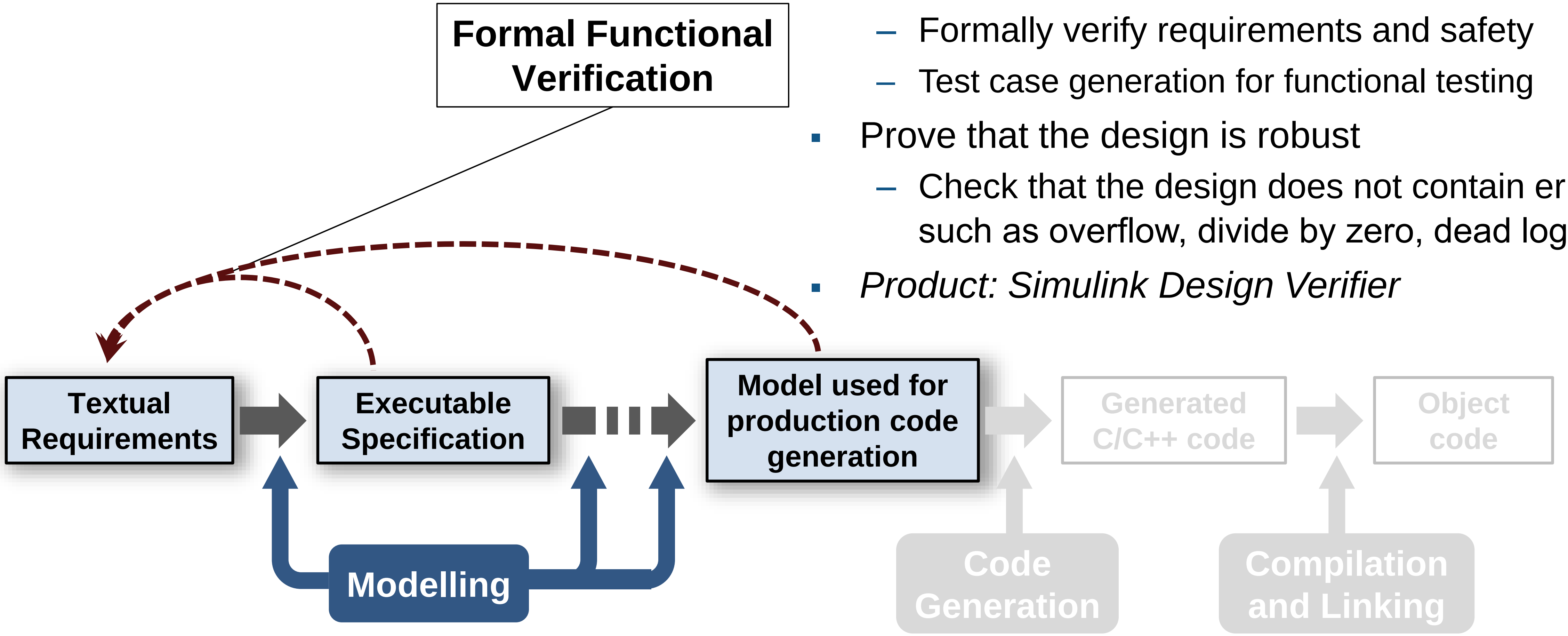


Test Manager



Verification and Validation Tasks and Activities

- Prove design meets requirements
  - Formally verify requirements and safety
  - Test case generation for functional testing
- Prove that the design is robust
  - Check that the design does not contain errors such as overflow, divide by zero, dead logic, …
- *Product: Simulink Design Verifier*



Formal Functional Verification

Textual Requirements

Executable Specification

Model used for production code generation

Generated C/C++ code

Object code

Modelling

Code Generation

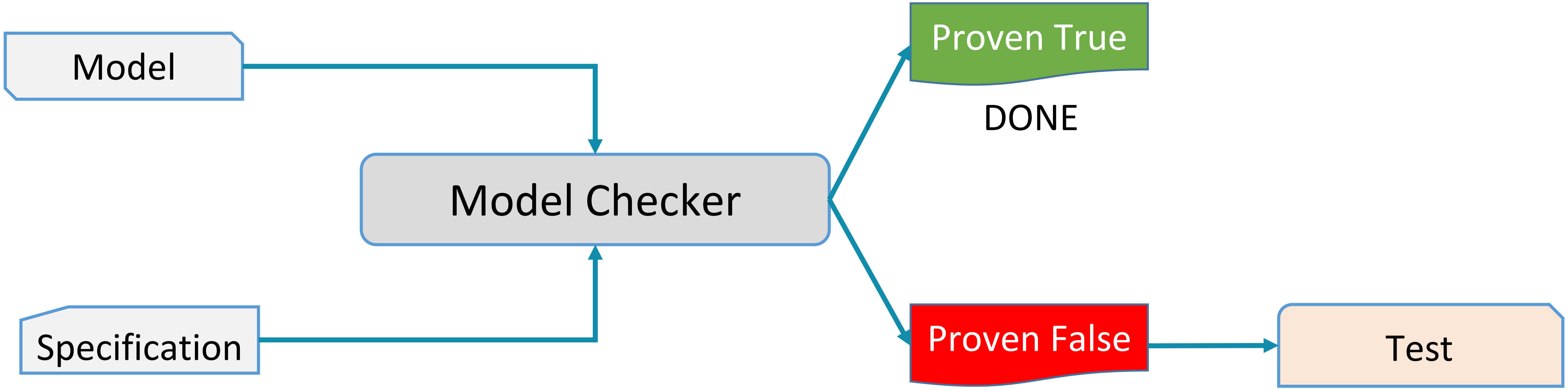
Compilation and Linking

Motivation for Formal Verification (Formal Methods)

- “Program testing can be used to show the presence of bugs, but never to show their absence” (Dijkstra)
- “Given that we cannot really show there are no more errors in the program, when do we stop testing?” (Hailpern)

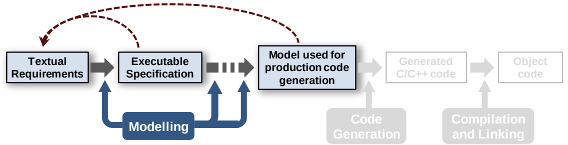
Dijkstra, “Notes On Structured Programming”, 1972
Hailpern, Santhanam, “Software Debugging, Testing, and Verification”, IBM Systems Journal, 2002

Formal Methods Technique – Model Checking



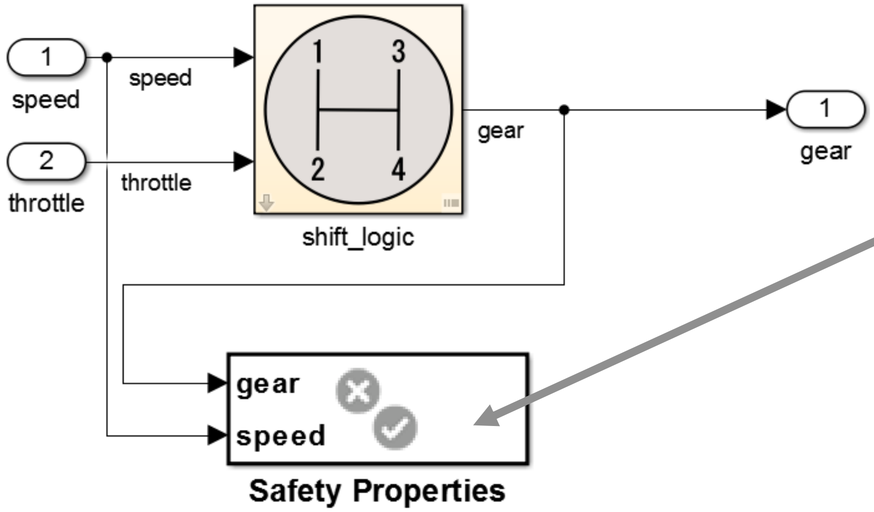
- Given
  - Design model
  - Requirement specification

- Prove that
  - Design meets the requirement specification, or
  - Does not meet the requirement and automatically generate test-case proving requirement not met



Prove That Design Meets Requirements

With Model Checking

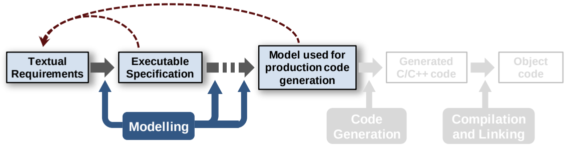


Checks that design meets requirements

- Gear 2 *always* engages when speed > 50
- Gear 2 *never* engages when speed < 5

Expected behavior of design

Behavior that design should not exhibit

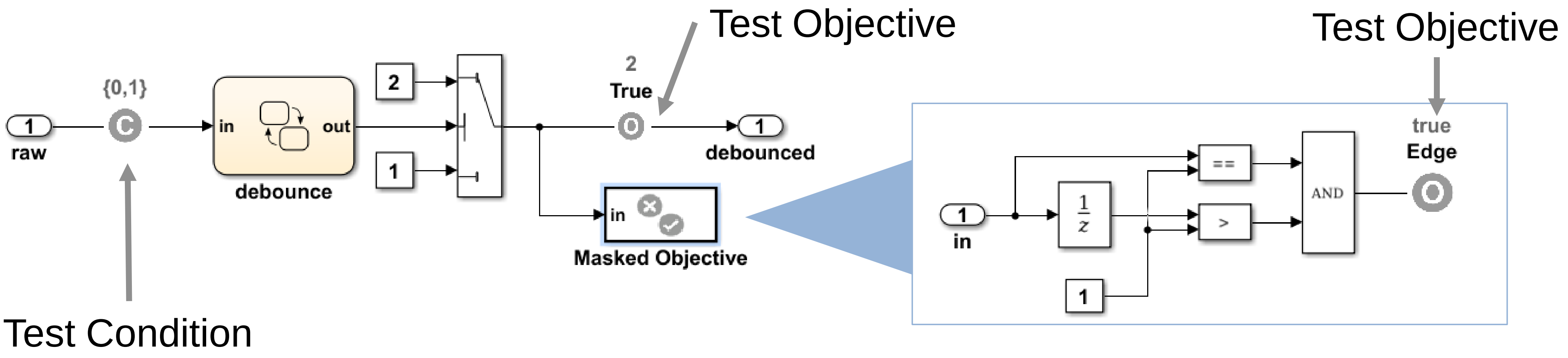


Test Case Generation for Functional Testing

With Model Checking

- Specify functional test objectives
  - Define custom objectives that signals must satisfy in test cases

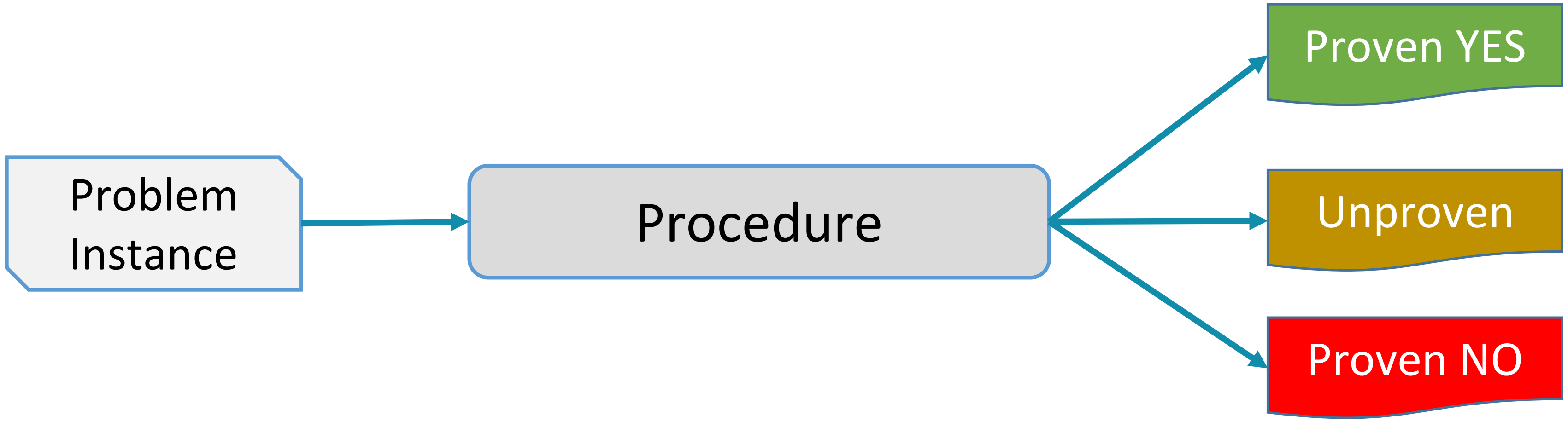
- Specify functional test conditions
  - Define constraints on signal values to constrain test generator


Test Objective

Test Objective

Test Condition

Formal Methods Technique – Abstract Interpretation



- Consider multiplication of three integers

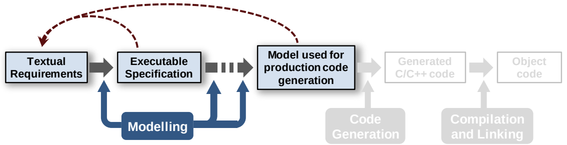
  –4586 × 34985 × 2389 = ?

- Quickly compute the final value by hand
  - What is the final answer?
  - What about the sign?

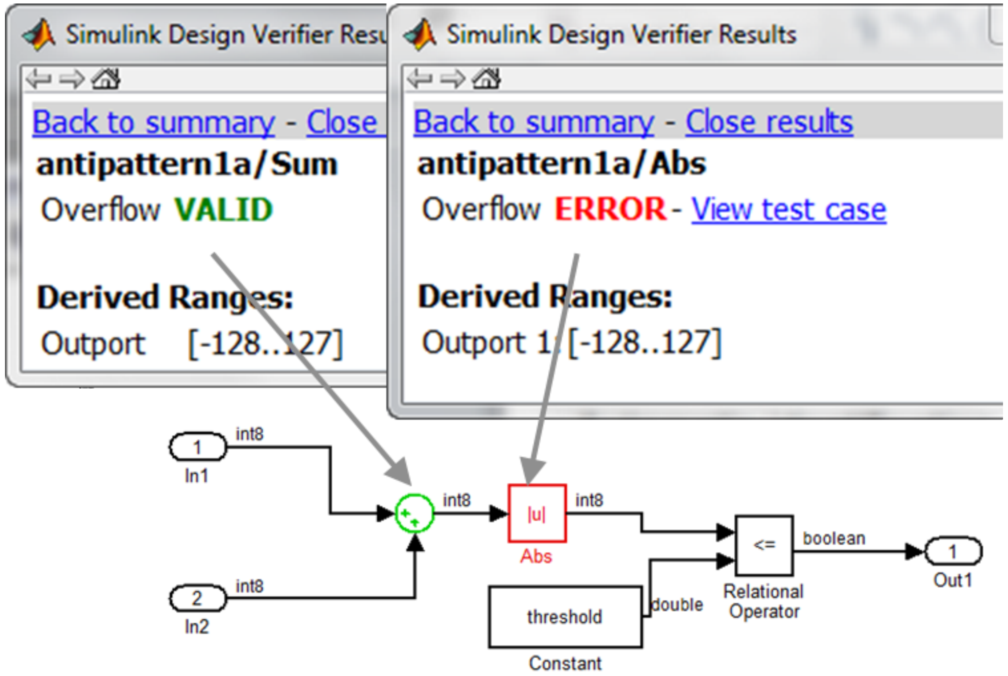
- The sign result
  - Could be positive, negative (or zero)
  - Per math rules, we know it is negative

- We abstracted complex details
  - Provably know precisely the sign

Prove That Design is Robust



With Abstract Interpretation



Simulink Design Verifier Resu

Back to summary - Close

antipattern1a/Sum

Overflow **VALID**

Derived Ranges:

Outport [-128..127]

Simulink Design Verifier Results

Back to summary - Close results

antipattern1a/Abs

Overflow **ERROR** - View test case

Derived Ranges:

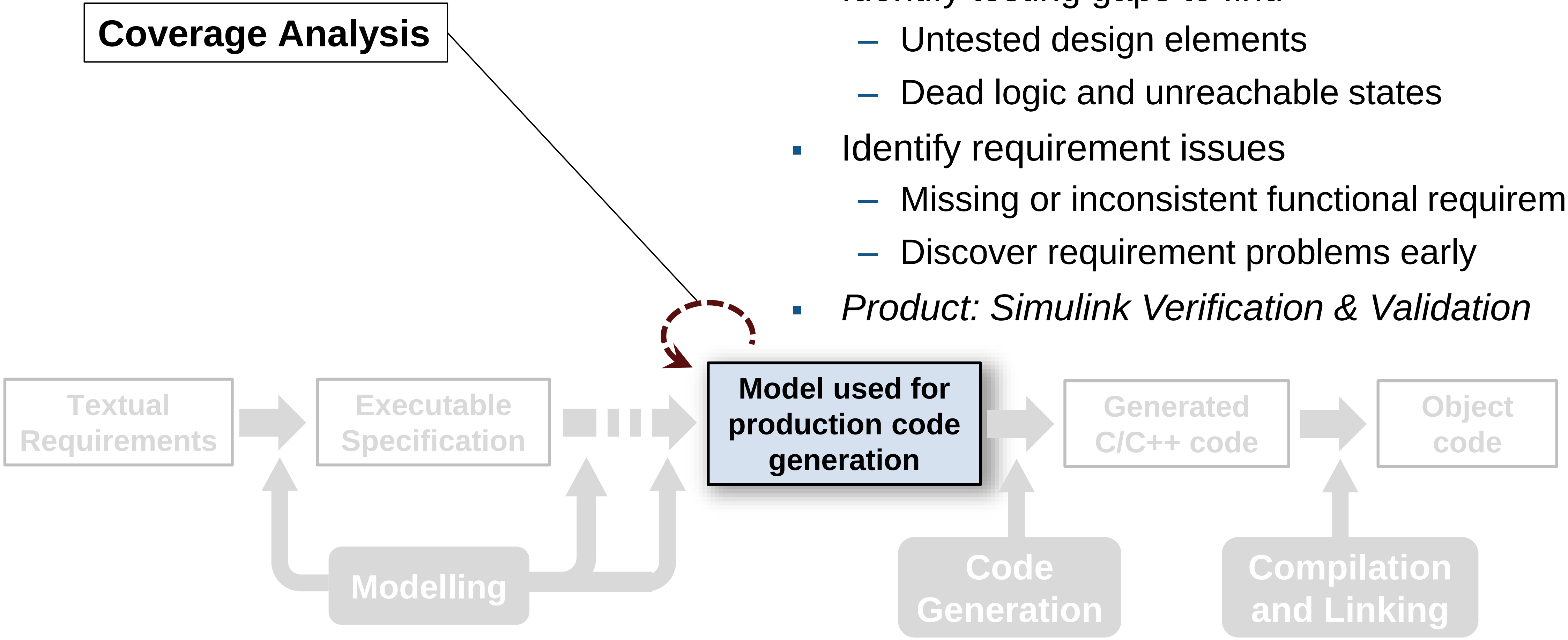
Outport 1 [-128..127]

Design can suffer from overflows, divide by zero, and other robustness errors

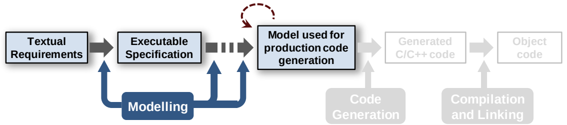
- Proven that overflow does NOT occur
- Proven that overflow DOES occur

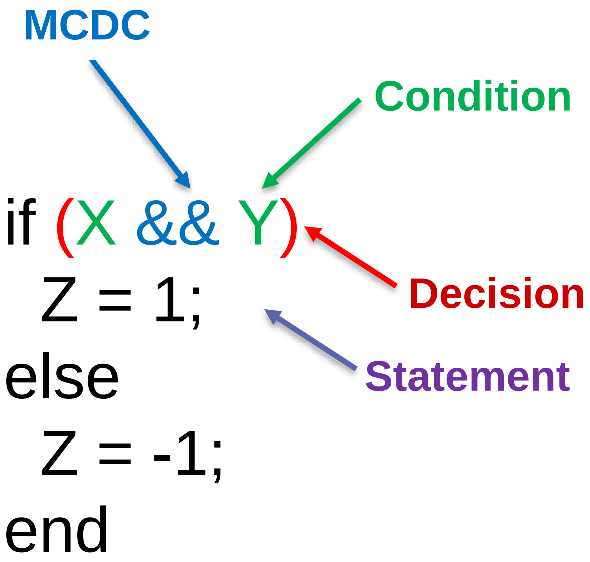
Verification Task

- Coverage metric
  - Measure of how much software has been tested
- Identify testing gaps to find
  - Untested design elements
  - Dead logic and unreachable states
- Identify requirement issues
  - Missing or inconsistent functional requirements
  - Discover requirement problems early
- *Product: Simulink Verification & Validation*



Coverage Concepts





```
if (X && Y)
  Z = 1;
else
  Z = -1;
end
```

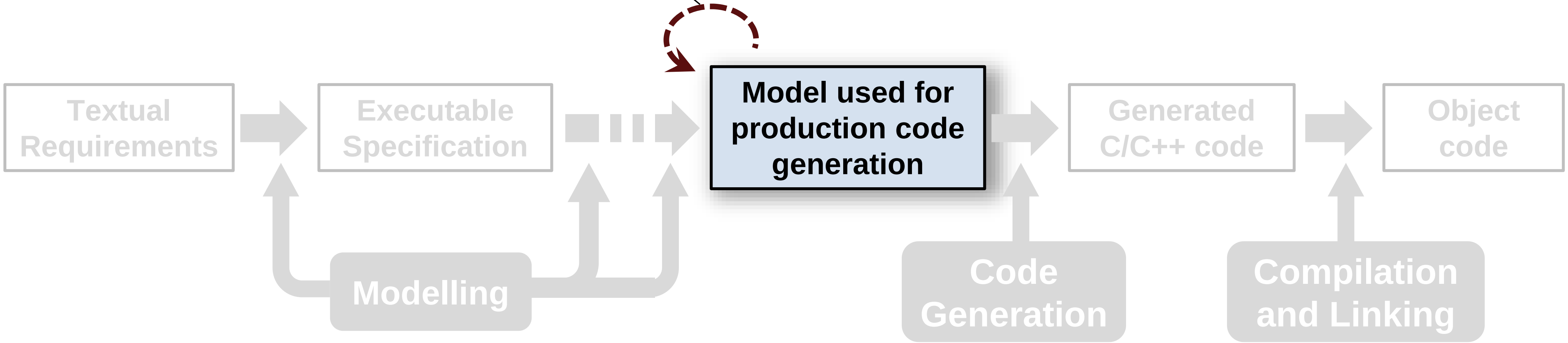
- Types of coverage
  - Statement: each statement in the code executed
  - Decision: has every branch of control statements executed
  - Condition: Boolean sub-expression evaluated for both true and false
  - Modified Condition Decision Coverage (MCDC)

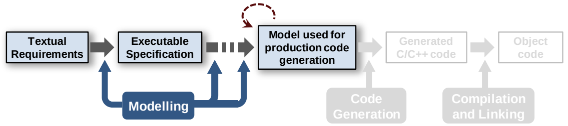
- MCDC explained
  - All entry/exit points invoked
  - Condition in decisions and conditions taken all possible outcomes
  - Each condition in a decision independently affects decision outcome

Verification and Validation Tasks and Activities

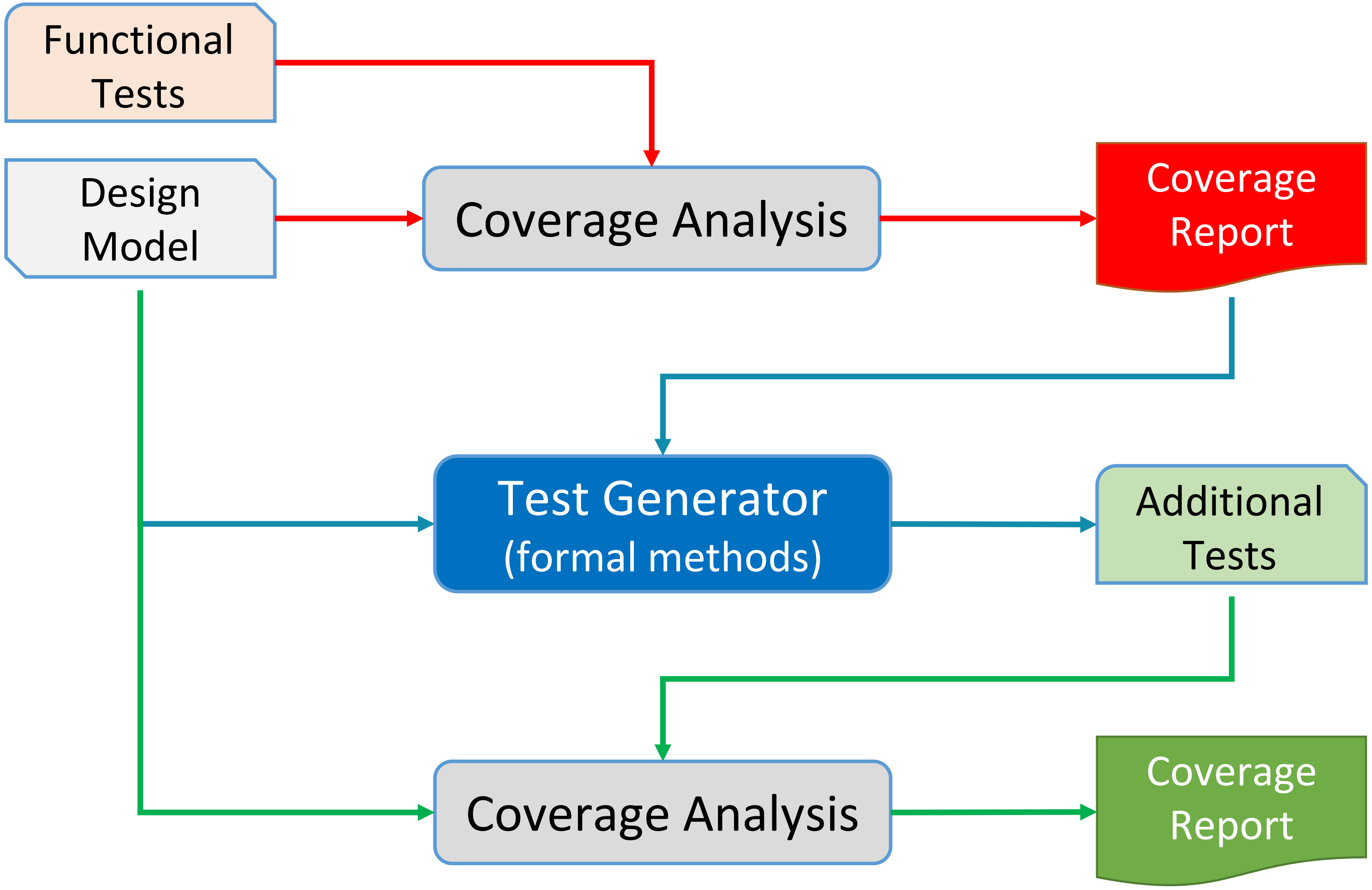
Test Generation for Coverage

- Automate manual task of writing test-cases and test inputs
  - Intelligent determination of input combinations for high coverage
- Formal methods based test generation
  - Analyze design, states, logic paths in the design model
- *Product: Simulink Design Verifier*





Addressing Missing Coverage



Step 1

Functional Tests

Design Model

Coverage Analysis

Coverage Report

Partial Coverage
(less than 100%)

Step 2

Test Generator
(formal methods)

Additional Tests

Step 3

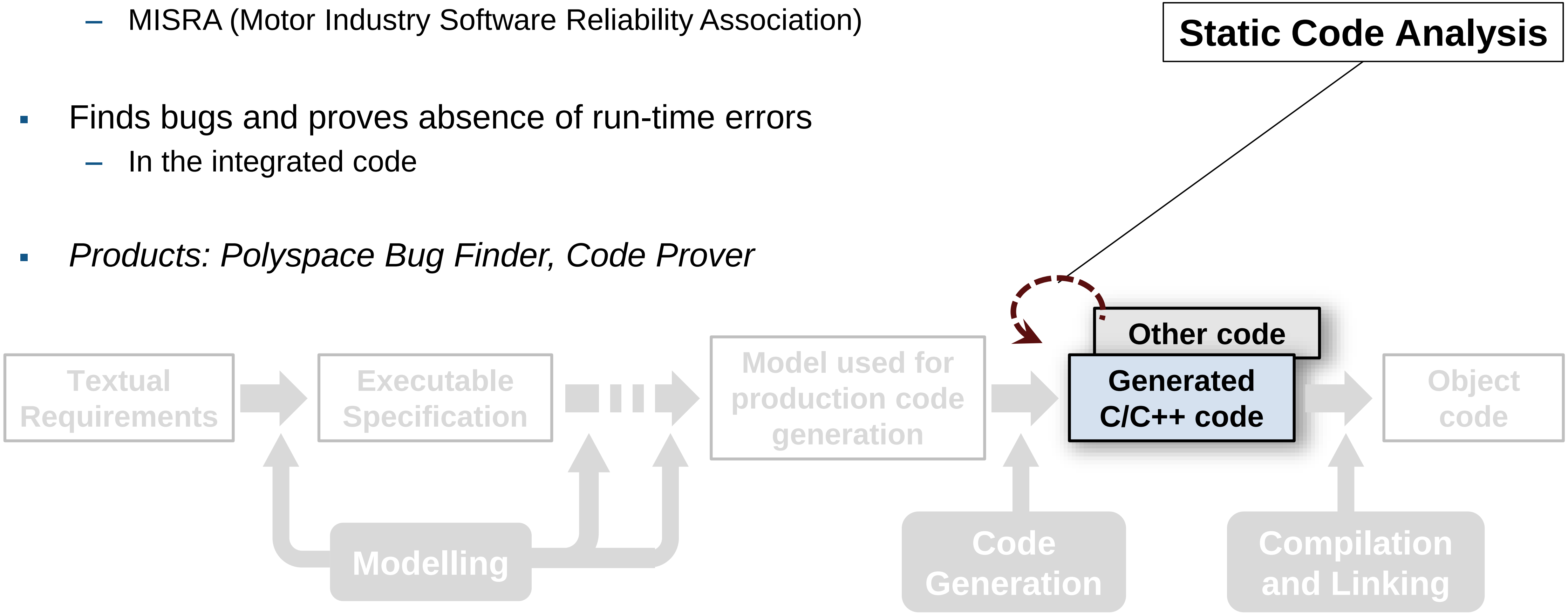
Coverage Analysis

Coverage Report

Full Coverage
(100%)

Verification and Validation Tasks and Activities

- Checks conformance to coding standards
  - MISRA (Motor Industry Software Reliability Association)
- Finds bugs and proves absence of run-time errors
  - In the integrated code
- *Products: Polyspace Bug Finder, Code Prover*

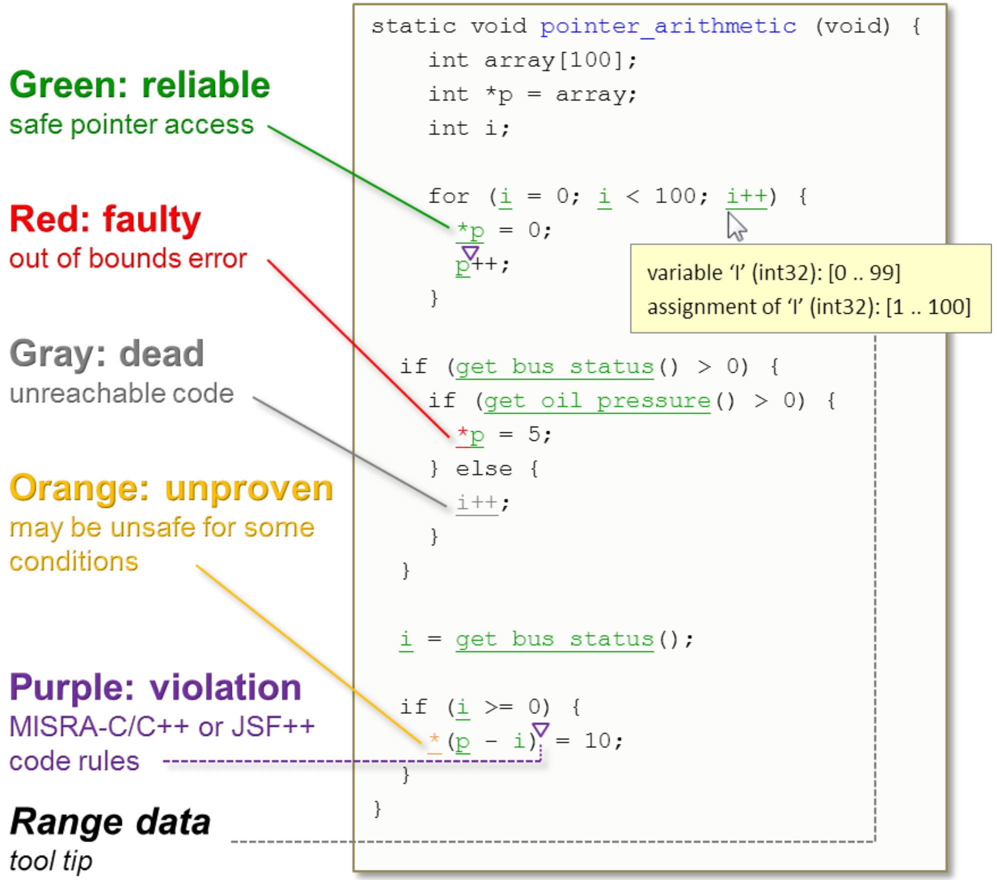


Motivation for Static Code Analysis

- The Generated Code is integrated with other Handwritten Code
- Impossible to exhaustively test the integrated code for bugs
- Certification standards require checking code for coding standards
- Critical run-time errors can cause un-intended behavior

Static Code Analysis Techniques

- Compiler warnings
  - Incompatible type detection, etc.
- Code metrics and standards
  - Comment density, cyclomatic complexity, MISRA C/C++
- Bug finding
  - Pattern matching, heuristics, data/control flow
- Code proving
  - Formal methods with abstract interpretation
  - No false negatives



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

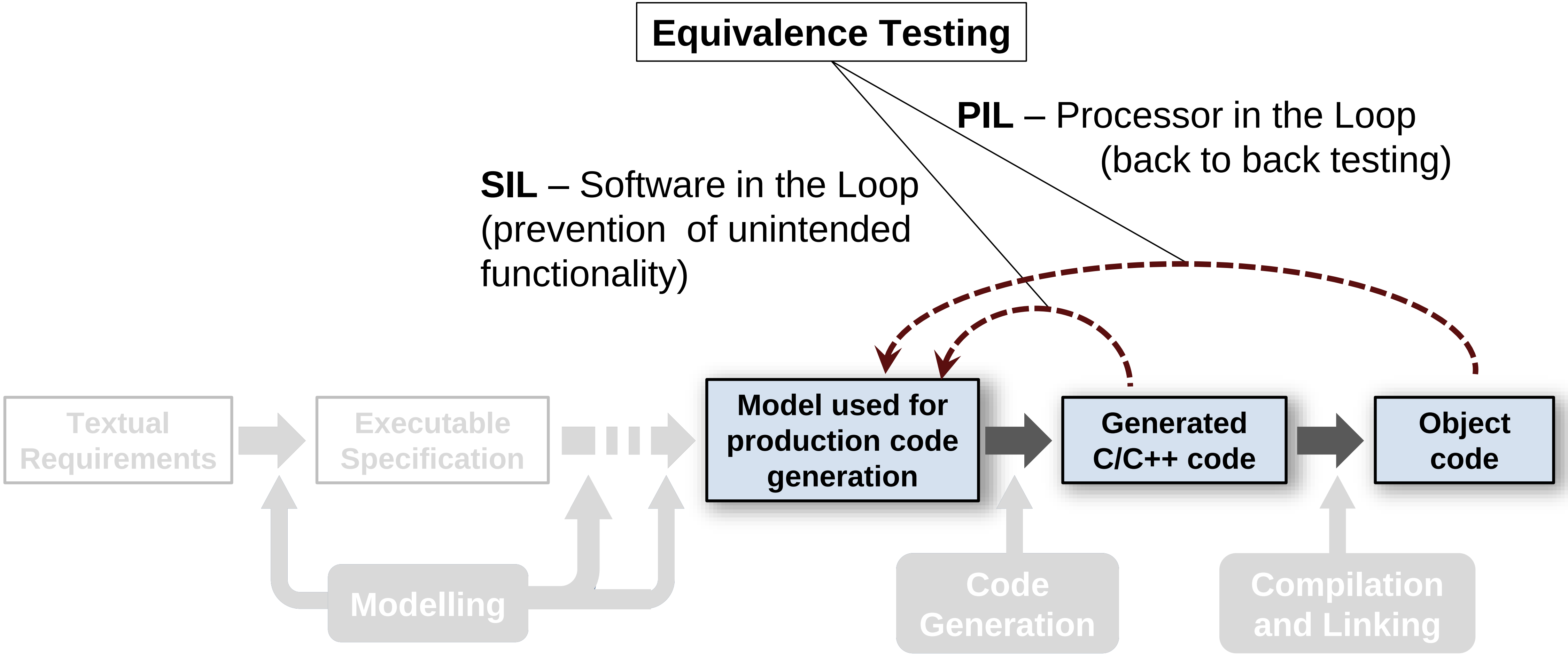
Range data
tool tip

```
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]

Results from Polyspace Code Prover

Verification and Validation Tasks and Activities



Equivalence Testing

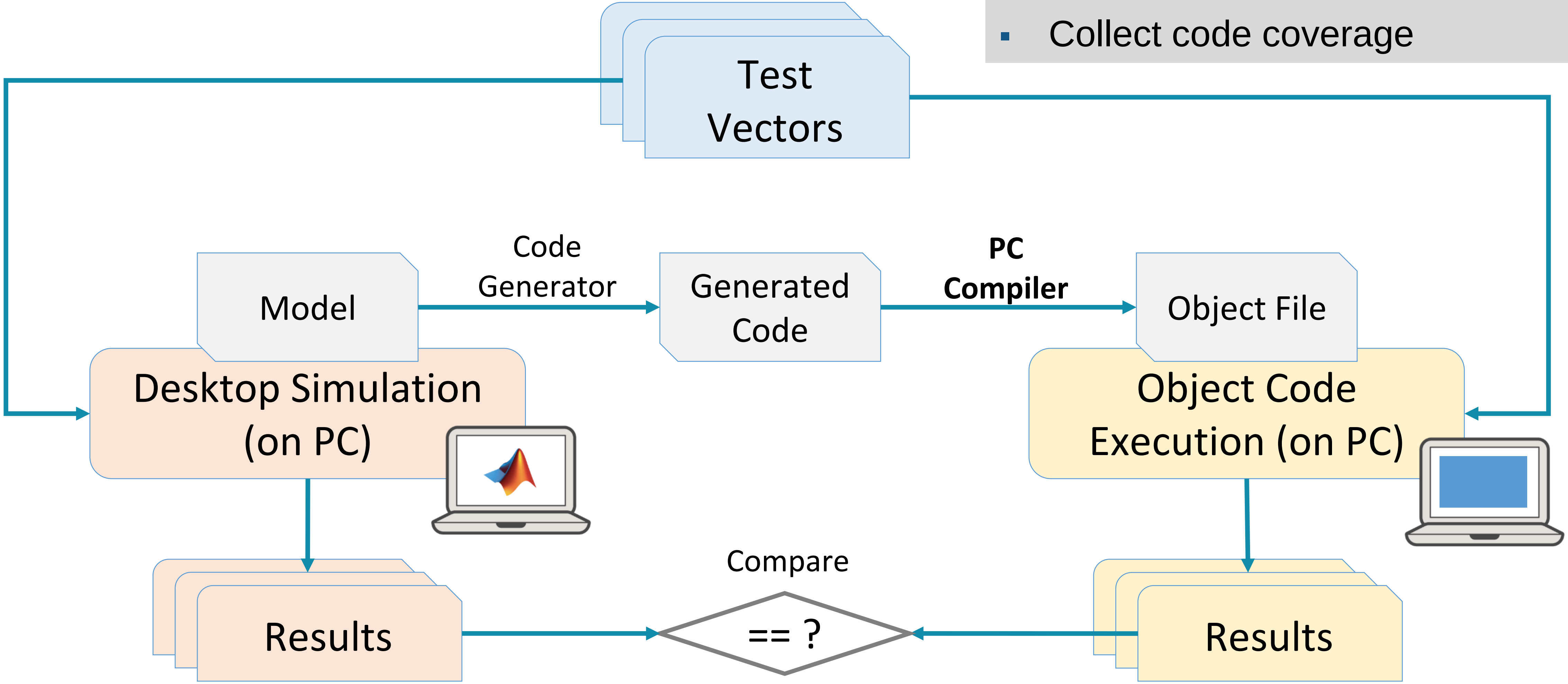
PIL – Processor in the Loop (back to back testing)

SIL – Software in the Loop (prevention of unintended functionality)

* Reference: ISO 26262 www.iso.org

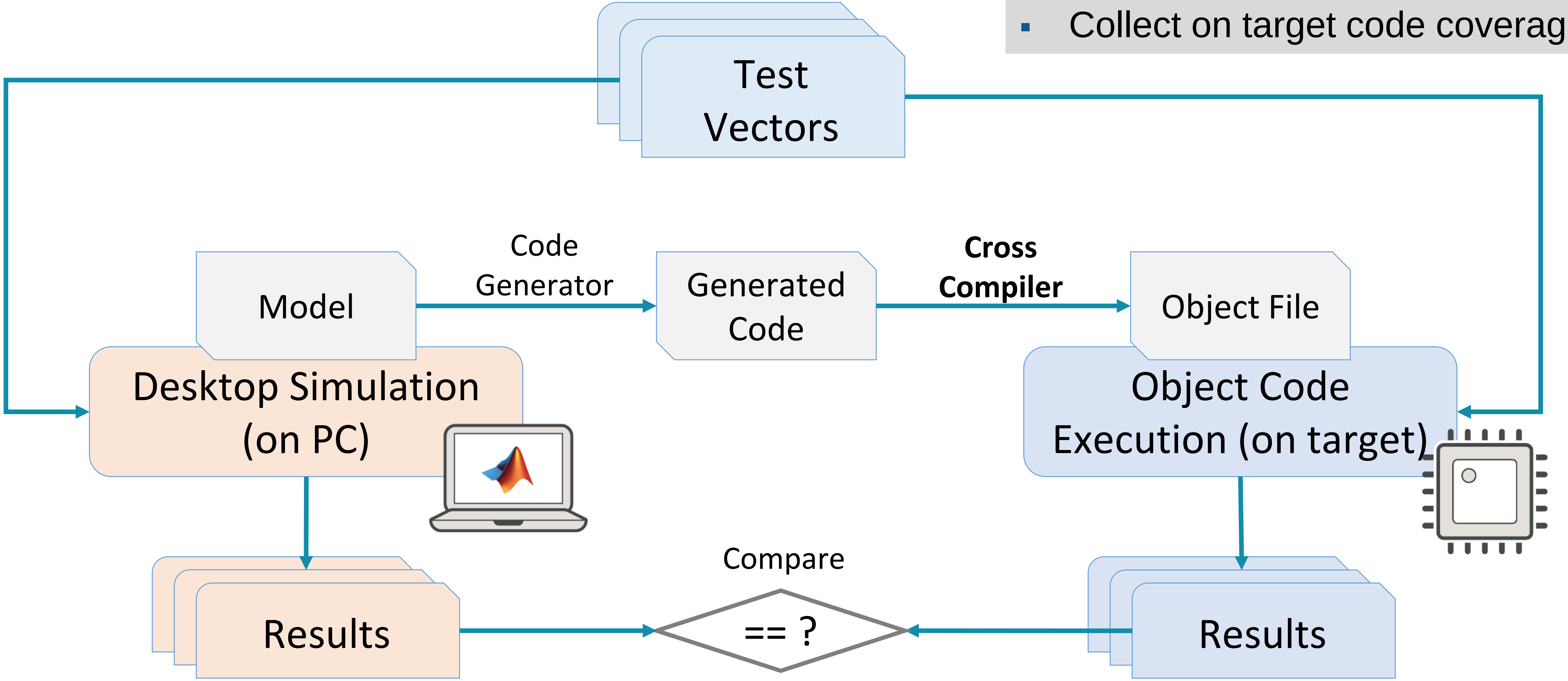
Software In the Loop (SIL) Testing

- 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


Test Vectors

Model

Code Generator

Generated Code

Cross Compiler

Object File

Desktop Simulation (on PC)

Object Code Execution (on target)

Results

Compare

== ?

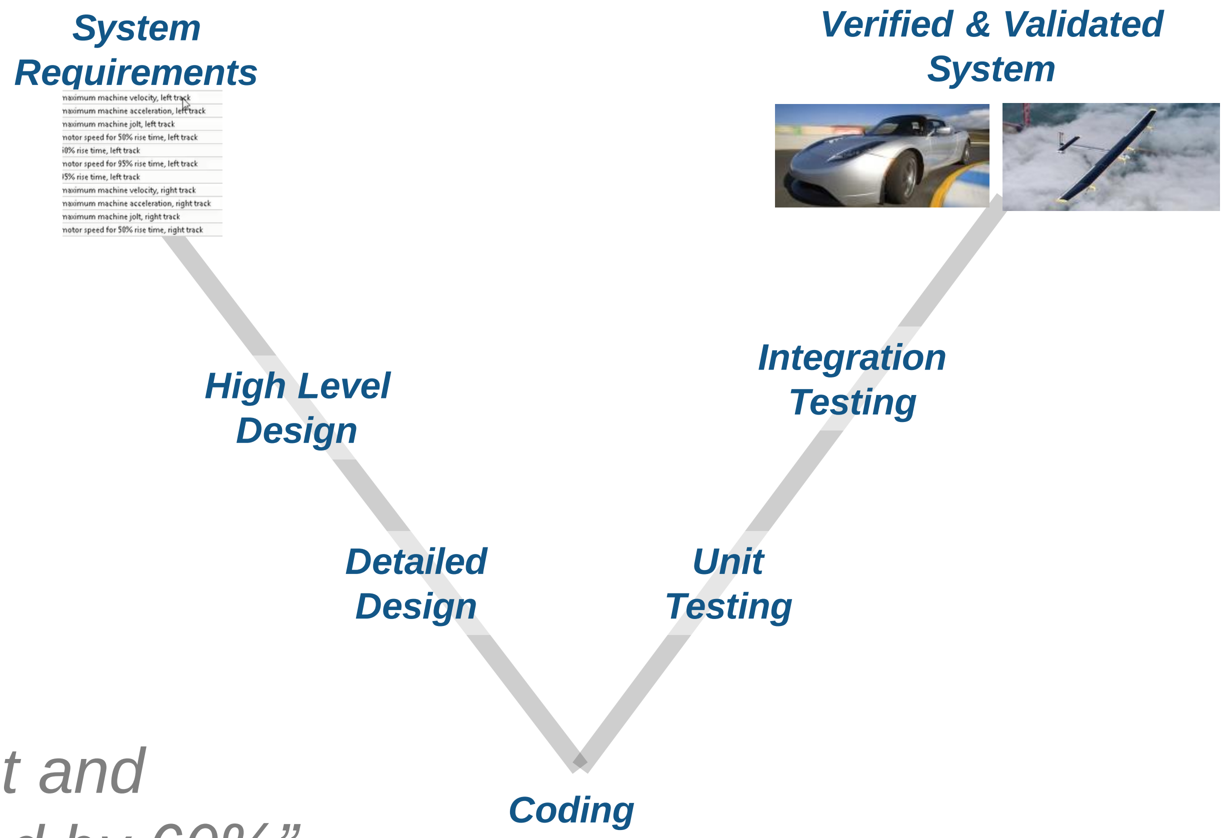
Results

MathWorks Solution Summary

Requirements Traceability	Simulink Verification and Validation
Testing	Simulink Test, Simulink Design Verifier
Formal Verification	Simulink Design Verifier, Polyspace Code Prover
Coverage Analysis	Simulink Verification and Validation
Static Code Analysis	Polyspace Bug Finder, Polyspace Code Prover
SIL, PIL	Simulink Test

Key Takeaways

1. Find bugs early, develop high quality software
2. Replace manual verification tasks with workflow automation
3. Learn about reference workflow that conforms to safety standards



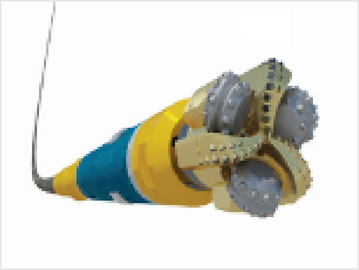
maximum machine velocity, left track
maximum machine acceleration, left track
maximum machine jolt, left track
motor speed for 50% rise time, left track
0% rise time, left track
motor speed for 95% rise time, left track
5% rise time, left track
maximum machine velocity, right track
maximum machine acceleration, right track
maximum machine jolt, right track
motor speed for 50% rise time, right track

UAV Flight Control Software Development and Verification … "development effort reduced by 60%"
Jungho Moon, KAL

Additional Customer References and Applications

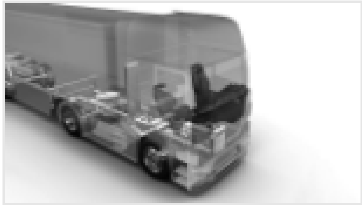
Airbus Helicopters Accelerates Development of DO-178B Certified Software with Model-Based Design

Airbus Helicopters Accelerates Development of DO-178B Certified Software with Model-Based Design



Baker Hughes Improves Precision of Oil and Gas Drilling Equipment

Baker Hughes Improves Precision of Oil and Gas Drilling Equipment



Continental Develops Electronically Controlled Air Suspension for Heavy-Duty Trucks

Continental AG used MathWorks software to design a level and roll control system for heavy-duty, 40-ton trucks.

Lear Delivers Quality Body Control Electronics Faster Using Model-Based Design

Lear Delivers Quality Body Control Electronics Faster Using Model-Based Design