Simulink 모델과 C/C++ 코드에 대한 매스웍스의 정형 검증 툴 소개
Agenda

- Formal Verification
  - Key concept
  - Applications
- Verification of designs against (functional) requirements
- Design error detection
- Test generation
- Test coverage analysis
- Run-time error detection
- Proving the absence of run-time errors
- Formal methods in certifications
Formal Verification Example

Example application of formal verification:
- Error detection in model and code

Model:
- **Simulink Design Verifier**

Code:
- **Polyspace Code Verifier**

```
static void pointer_arithmetic(void) {
    int array[100];
    int fp = array;
    int i;
}
```
Formal Verification

- Formal verification is…
  - “…mathematically rigorous procedures to search through the possible execution paths of your model for design errors, test cases and counterexamples.” *(Simulink Design Verifier homepage)*
  - “…act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.” *(Wikipedia)*
What are the key formal verification concepts?
Testing vs. Formal Verification

- Illustration

**TESTING**
- Point coverage through simulation

**FORMAL VERIFICATION** (Ideal case)
- Complete coverage of design space (formal proof)

**FORMAL VERIFICATION*** (Real-world)
- Real-world application of formal verification

* Source: Erik Seligman, “Formal Verification – An Overview”
Formal Verification Process

1. Algorithm (System)
   - Informal Functional and Non-Functional Requirement

2. Formal Property
   - Formal model (System)
   - Formal Methods Engine

3. Improve algorithm, or requirements
What are applications of formal verification today?
Applications Today

- Verification of designs against functional requirements
- Design error detection
- Test generation
- Test coverage analysis
- Run-time error detection
- Proving the absence of run-time errors
Verification of Designs Against Requirements
Verification of Designs Against Requirements

Algorithm (System)

Verification Model

Properties (Requirements)

Simulink Design Verifier

V1.0
V2.0
V3.0
Examples of Formalized Requirements

Explicit descriptions of required behavior – functional or safety requirement

- **Primitive:**
  - Objectives, proofs

- **Invariant:**
  - \( A > B \implies C=0 \)

- **Temporal:**
  - \( A > B \) for 10 time steps \( \implies C=0 \) within 5 time steps

- **Complex, state-based**
  - Stateflow, MATLAB functions

Example invariant:
- Always identical, for every time step, including initialization, all modes of operation
Design Error Detection
Design Error Detection

- Automated detection of hard-to-find errors:
  - Integer overflow
  - Division by zero
  - Dead logic
  - Assertion violation
  - Range violation

- Results displayed on the model including data values

- For each error* a test vector is generated for root-cause analysis in simulation

*Where applicable
Test Generation
Generating Tests for Functional Requirements

Must Do (Test Case)

Must demonstrate opening and closing the valve when [Condition]
Test Coverage Analysis
Model Coverage Testing

Generate Tests

Measure Coverage in Simulation

Simulink Design Verifier

Simulink Verification and Validation
Model Coverage Analysis
Measure of Test Completeness

- Execution analysis
  - Based on the model structure
  - Dynamic – data collected during simulation

- Coverage results
  - Displayed directly in the model
  - Available in a separate html report linked with the model objects

- Supports:
  - Simulink
  - Stateflow
  - Embedded MATLAB

Supported coverage types:
- Decision coverage
- Condition coverage
- MC/DC
- Lookup table coverage
- Signal range coverage
Run-Time Error Detection
Augmenting Static Code Analysis with Formal Methods

- Static code analysis – scan source code to automate verification
- Range from *unsound* methods to *sound* techniques

**Formal methods**
- Sound proof based techniques, applied to source code
- Can show SW is robust for wide operating conditions

**Bug finding**
- Pattern matching, heuristics, data/control flow

**Compiler warnings**
- Incompatible type detection, etc.
Polyspace – Formal Methods based Static Code Analysis

- Exhaustively verify code
  - Detect and prove absence of runtime errors
  - Precisely determines and propagates variable ranges

- Languages supported
  - C, C++, and Ada

- Verify SW robustness
  - Analyze for full range operating conditions
  - Specified ranges of parameters and inputs
Example: Finding a Run-Time Error

```c
int array[100];
int i, *p = array;

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

if(get_bus_status() > 0)
{
    if(get_oil_pressure() > 0)
    {
        *p = 5; /* Out of bounds */
    }
    else
    {
      /* Dereference of variable 'p' (pointer to int 32, size: 32 bits): 
         pointer is not null 
         points to 4 bytes at offset 400 in buffer of 400 bytes, so is outside bounds 
         may point to variable or field of variable in: {Pointer_Arithmetic:array} */
    }
}  
```
Proving Absence of Runtime Errors
Formal Methods based Static Code Analysis

You Can Prove That:

\[ c = a + b; \quad \text{will never overflow} \]

\[ j = \text{arr}[i]; \quad i \text{ will always be within array bounds} \]

\[ *\text{ptr} = 12; \quad \text{will never be an illegal dereference} \]

\[ w = x / (y + z); \quad y \text{ never equal to } -z \text{ or visa versa (divide by zero)} \]

And many more …
Example: Proving Absence of Run-time Errors

```c
int new_position(int sensor_pos1, int sensor_pos2)
{
    int actuator_position;
    int x, y, tmp_pos, magnitude;

    actuator_position = 2; /* default */
    tmp_pos = 0;        /* values */
    magnitude = sensor_pos1 / 100;
    y = magnitude + 5;
    x = actuator_position;

    while (actuator_position <= 10)
    {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
    }

    if ((3*magnitude + 100) >= 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
    }

    return actuator_position + tmp_pos; /* new value */
}
```
Example: Proving Absence of Run-time Errors

```c
x -= 3;
}
if ((3*magnitude + 100) > 43)
{
    magnitude++;
    x = actuator_position;
    actuator_position = x √ (x - y);
}
return actuator_position + tmp_pos; /* new value */
```
Example: Proving Absence of Run-time Errors

```c
16         x = 0;
17         }
18         if ((3\times magnitude + 100) > 43) {
19         
magnitude++;
20         x = actuator_position;
21         actuator_position = x \div (x - y);
22         }
24         return actuator_position + tmp_pos + x;
25         }
```

variable ‘x’ (int 32): 10
Example: Proving Absence of Run-time Errors

```c
16       x := y;
17     }
18     if ((3*\texttt{magnitude} + 100) > 43)
19         
20             \texttt{magnitude}++;
21             x = \texttt{actuator\_position};
22       \texttt{actuator\_position} = x \div (x - y);
23         }
24     return \texttt{actuator\_position} + /
25     }
```

variable \texttt{y} (int 32): 11 .. 21474865
Example: Proving Absence of Run-time Errors

```c
int new_position(int sensor_pos1, int sensor_pos2)
{
    int actuator_position; // parameter sensor_pos1 (int 32): full-range [-2^{31} .. 2^{31} - 1]
    int x, y, tmp_pos, magnitude;

    actuator_position = 2; /* default */
    tmp_pos = 0; /* values */
    magnitude = sensor_pos1 / 100;
    y = magnitude + 5;
    x = actuator_position;

    while (actuator_position <= 10)
    {
        actuator_position++;
        tmp_pos = sensor_pos2 / 100;
        y += 3;
    }

    if ((3*magnitude + 100) >= 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
    }

    return actuator_position + tmp_pos; /* new value */
}
```
Example: Proving Absence of Run-time Errors

```c
int new_position(int sensor_pos1, int sensor_pos2)
{
    int actuator_position;
    int x, y, tmp_pos, magnitude;

    actuator_position = 2; /* default */
    tmp_pos = 0; /* values */
    magnitude = sensor_pos1 / 100;
    y = magnitude + 5;
    x = actuator_position;

    while (actuator_position <= 10)
    {
        actuator_position++;
        tmp_pos += sensor_pos2 / 100;
        y += 3;
    }

    if ((3*magnitude + 100) >= 43)
    {
        magnitude++;
        x = actuator_position;
        actuator_position = x / (x - y);
    }

    return actuator_position + tmp_pos; /* new value */
}
```

parameter sensor_pos2 (int 32): full-range [-2^31 .. 2^31 - 1]
Can I get certification credit if I use formal verification techniques in my process?
DO-178B Certification Credit

Table A-5 (ref section: 6.3.4b, 6.3.4c, 6.3.4d, 6.3.4f)
- Dataflow, control flow
- Dead code
- Compliance with standards (MISRA, JSF++)
- Arithmetic overflow, resource contention, uninitialized variables, ...

Table A-6 (ref section: 6.4.2.1, 6.4.2.2, 6.4.3)
- Uninitialized variables, parameter passing errors, data corruption, ...
- Incorrect loop/logic decisions, arithmetic faults, violations of array limits, ...
- Diagnostics of runtime failure, operation under full range conditions, non terminating loops, ...
Formal Methods Supplement to DO-178C and DO-278A

- Allows the use of Formal Methods Analysis for verification of:
  - Software requirements
  - Software design
  - Source code
  - Executable object code

- Adds an objective that the formal analysis method must be shown to be sound

- What does this mean
  - Polyspace is a formal methods tool for analysis of source code
  - As a Criteria 2 tool, partial credit may be taken for analysis of executable object code and this is included in the DO Qualification Kit
  - DO Qualification Kit already includes a Theoretical Foundation document to justify the soundness of Abstract Interpretation
IEC 61508 Certification of MathWorks Products

- TÜV SÜD certified:
  - Real-Time Workshop Embedded Coder
  - Simulink Design Verifier
  - Simulink Verification and Validation
  - Polyspace products for C/C++

- For use in development processes which need to comply with IEC 61508, ISO 26262, or EN 50128

Note: The products listed above were not developed using certified processes