*Simulink Models – Assurance Through Comprehensive Formal Verification**

John C. Knight
Dependable Computing LLC
&
University of Virginia

*Funded by Toyota ITC, Mountain View, CA

DEPENDABLE COMPUTING

The Problem

Traditiona

Natural Language ***Informal Specification***

Traditional Development

Existing Techniques Problematic

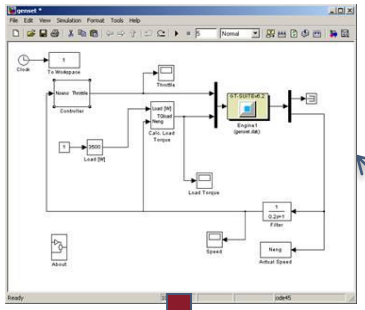
High Level Language Implementation

Binary Implementation

The world has changed ...

Model-Based Development

Consistency Checking
Functional Correctness
Safety Properties



Formal Specification

Increasing And Diverse Use

Automatic Synthesis

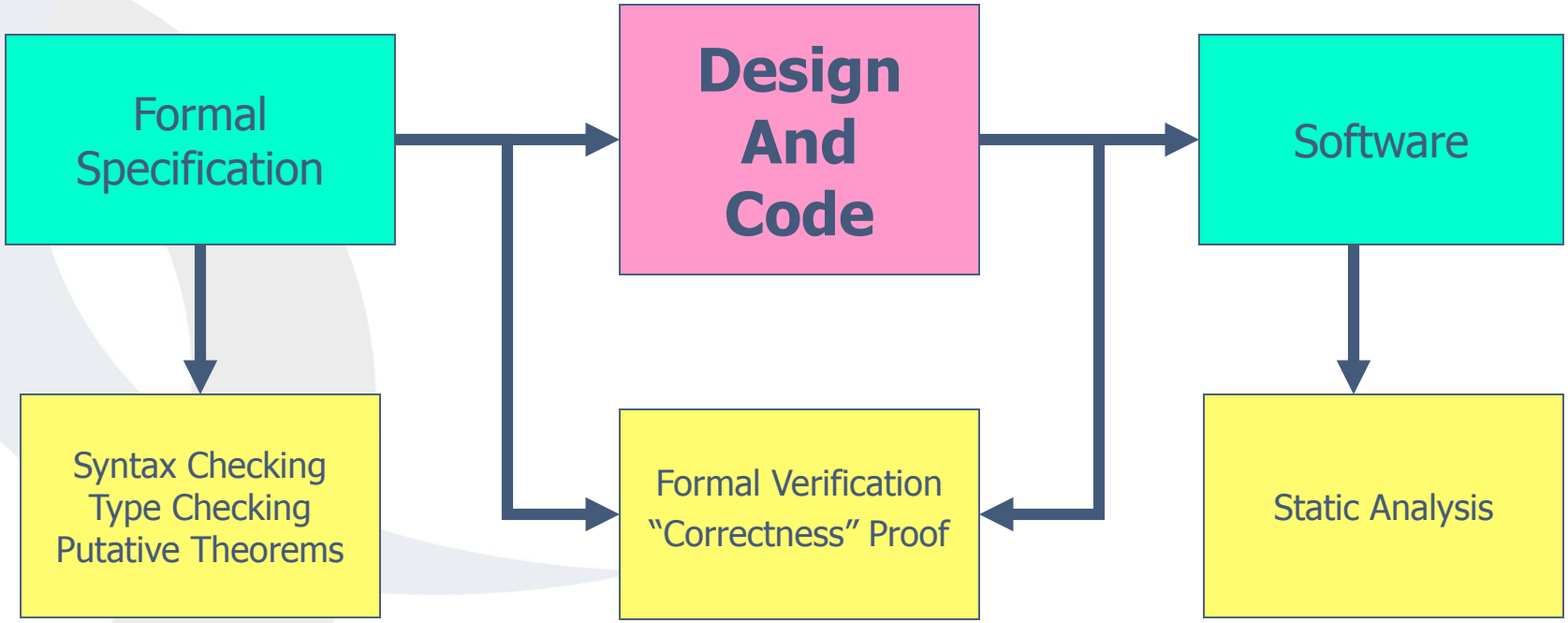
High Level Language Implementation

Binary Implementation

DEPENDABLE COMPUTING

Formal Languages & Proofs

- Proof in *traditional* development:



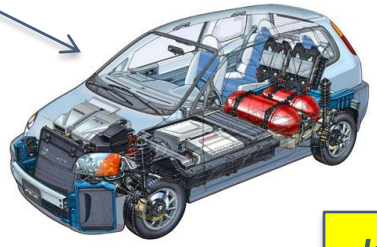
Proof is equivalent to executing *all* test cases

In principle, displaced

DEPENDABLE COMPUTING

Derivation of Software Spec'n

System of Interest



Derivation Of Specification

Implication

Derivation Of Crucial Properties

Software Based Component

System *Design*

Software *Specification*

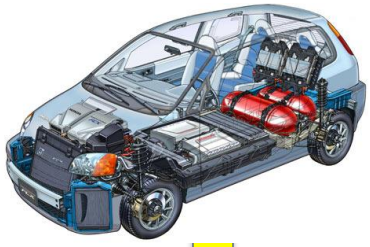
Software **specification** derives from *system* design

Crucial *software* **properties** derive from *system* design

DEPENDABLE COMPUTING

Model-Based Development

New Innovative Concept



Model-Based Development Simulink, SCADE, Etc.

What About Proof?

Design Environment

Software Specification

Development Environment

PRODUCTION

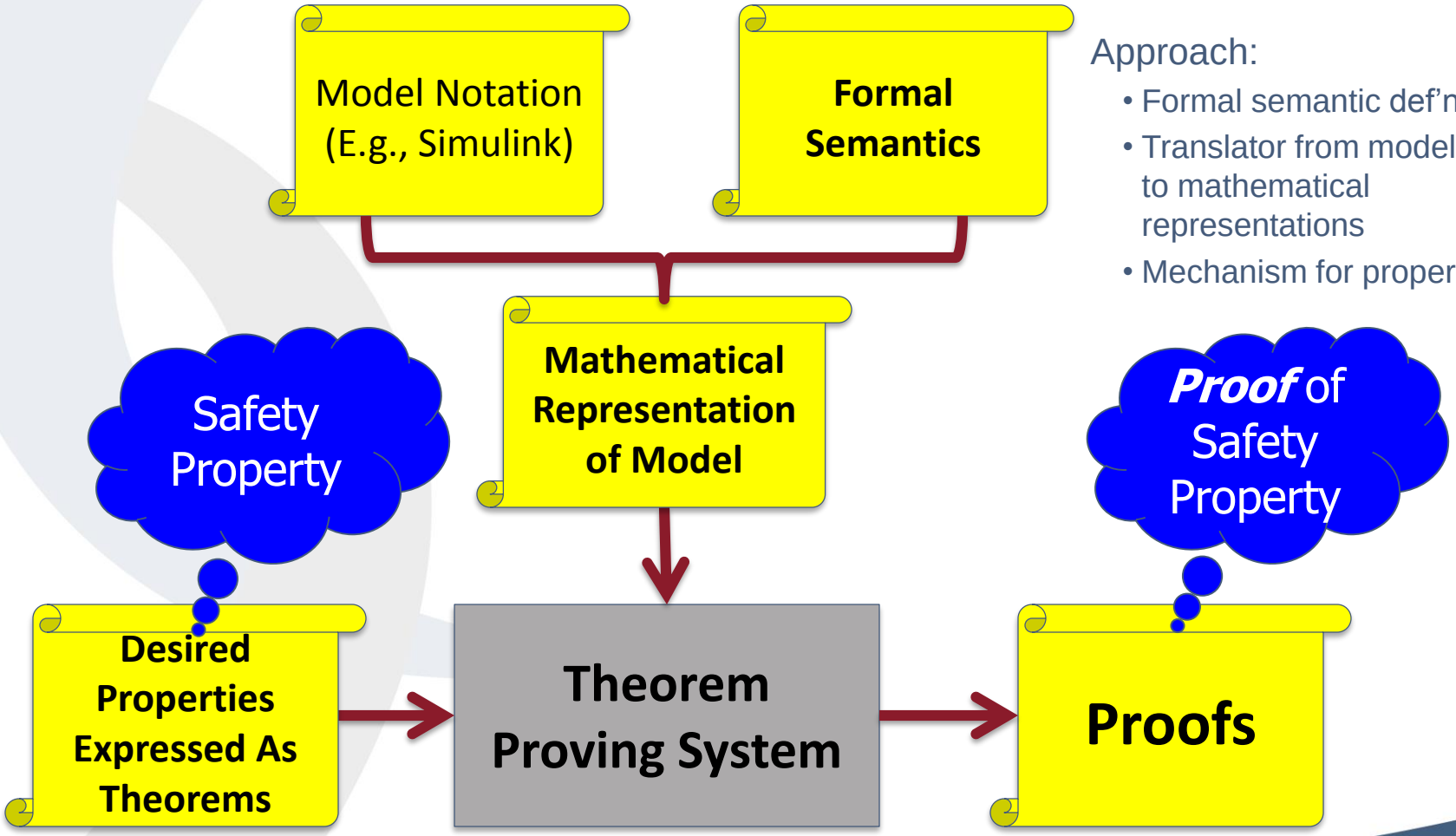
System

Software Implementation

Sub System

DEPENDABLE COMPUTING

Verification Concept

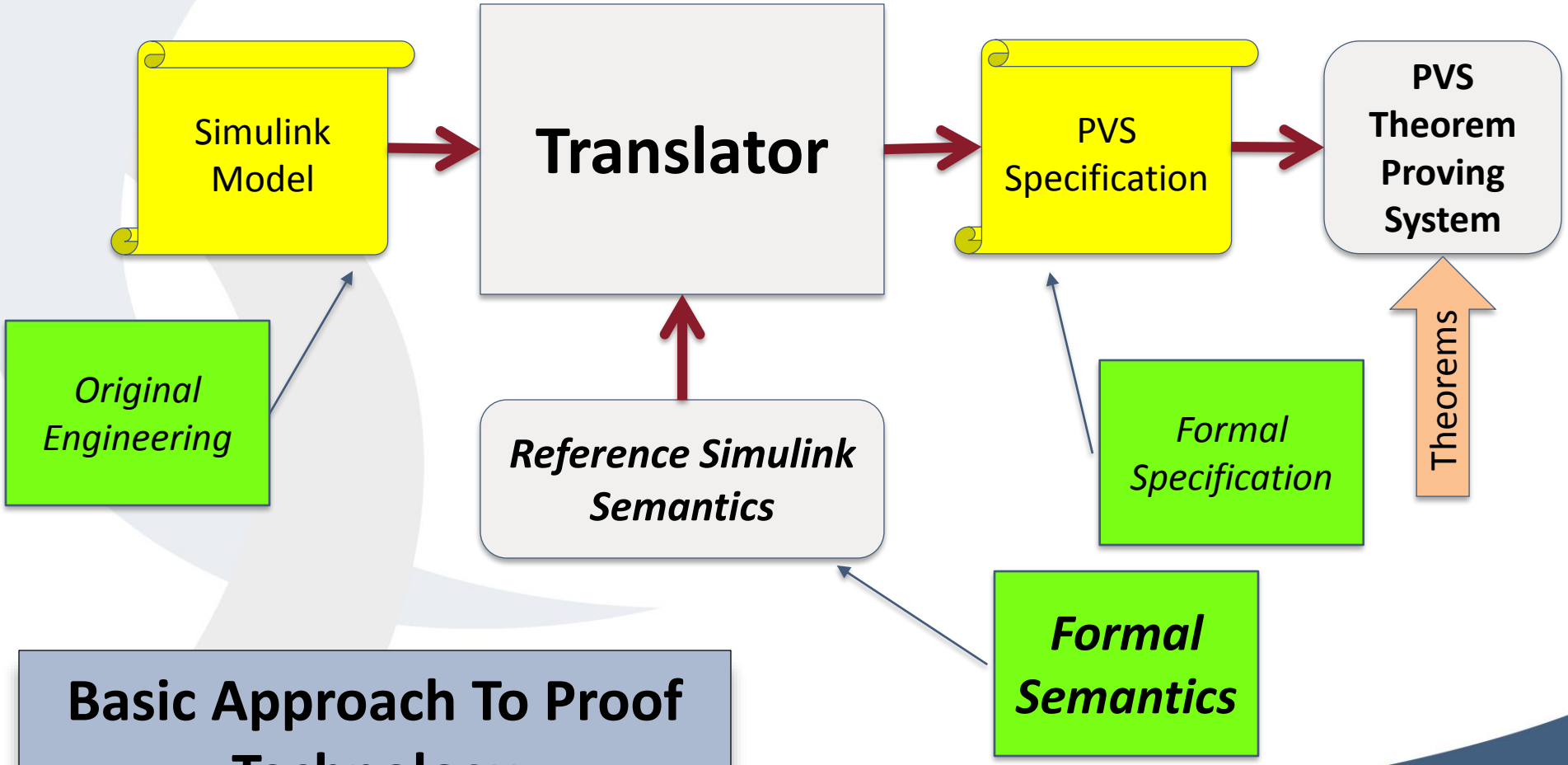


Approach:

- Formal semantic def'n
- Translator from model notation to mathematical representations
- Mechanism for property def'ns

DEPENDABLE COMPUTING

Translation To PVS



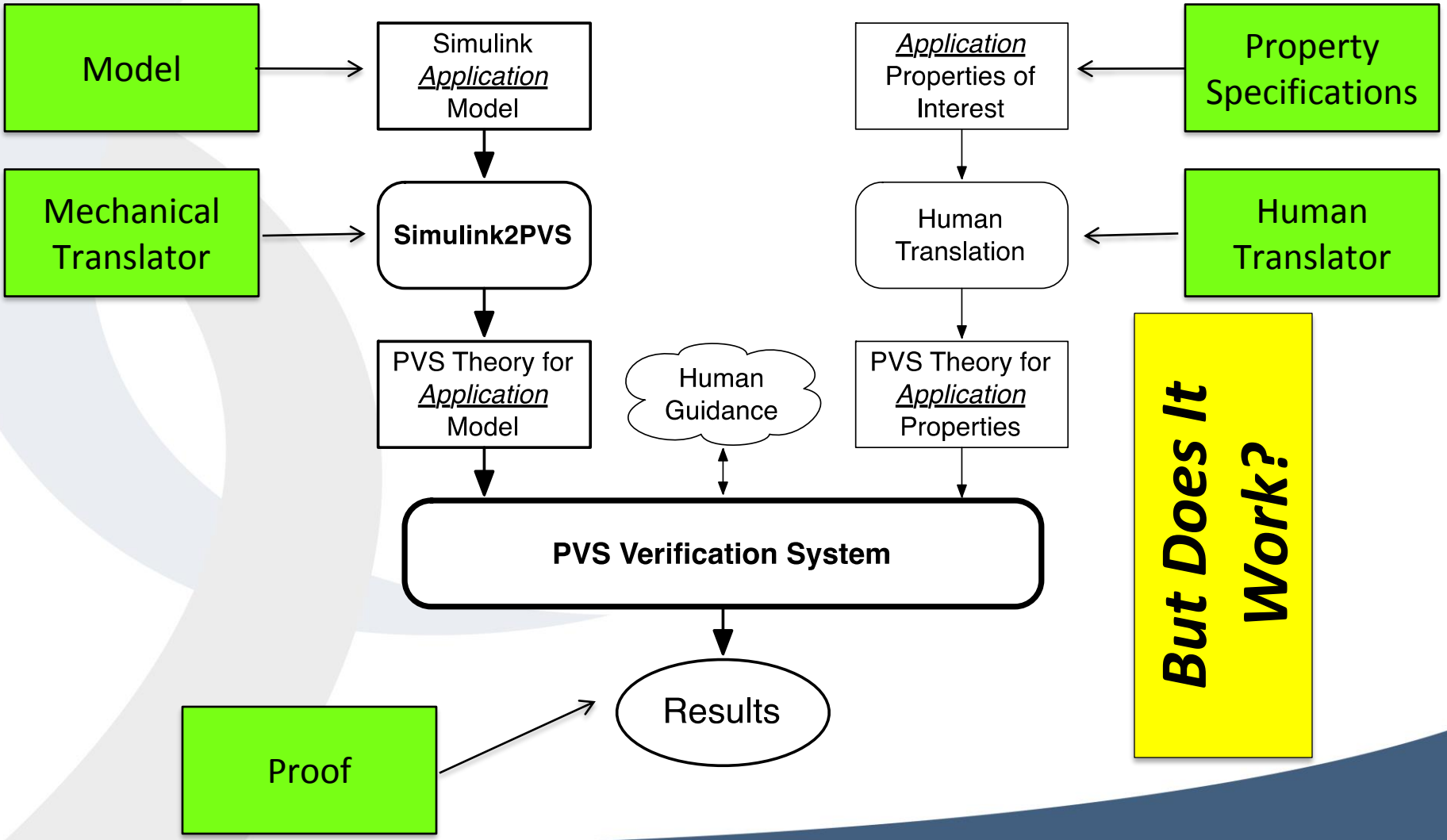
DEPENDABLE COMPUTING

Simulink Blocks For Which Semantics Defined

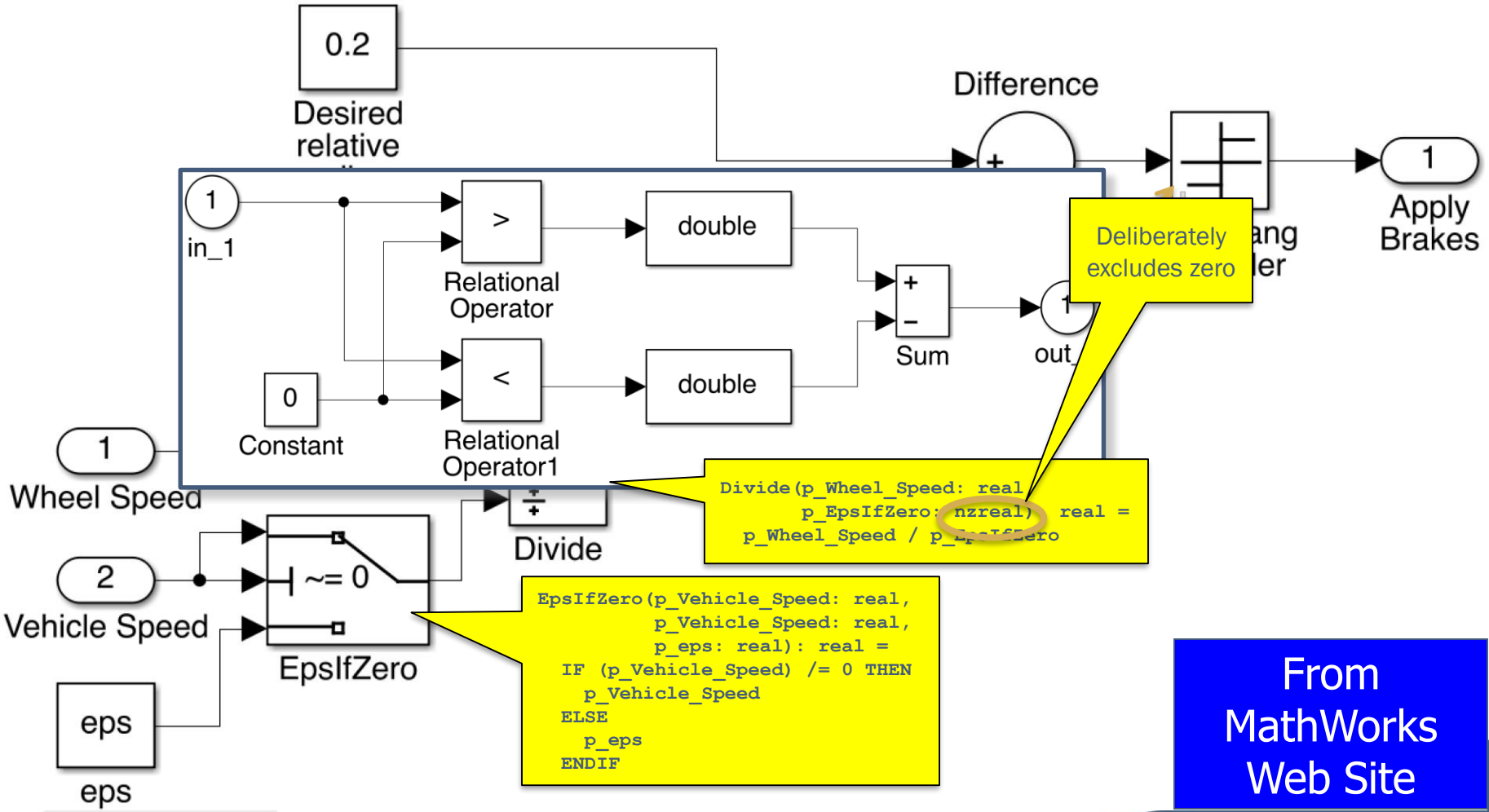
- 1-D Lookup Table
- 2-D Lookup Table
- Abs
- Action Port
- Assertion
- Assignment
- Bus Creator
- Bus Selector
- Compare to Constant
- Compare to Zero
- Constant
- Data Store Memory
- Data Store Read
- Data Store Write
- Data Type Conversion
- Data Type Duplicate
- Demux
- Display
- Enable Port
- From
- From Workspace
- Gain
- Goto
- Ground
- If
- If Action Subsystem
- In Port
- Integrator
- Logical Operator
- Math
- Merge
- Model Reference
- Mux
- n-D Lookup Table (for n ≤ 3)
- Out Port
- Product
- Pulse Generator
- Reference
- Relational Operator
- Saturate
- Scope
- Selector
- Shift Arithmetic
- Signal Conversion
- Signal Viewer Scope
- Sqrt
- Step
- Stop
- Subsystem
- Sum
- Switch
- Terminator
- Trigger Port
- Unary Minus
- Unit Delay
- Width

DEPENDABLE COMPUTING

Use Of Simulink2PVS



Antilock Brake System Example



From MathWorks Web Site

DEPENDABLE COMPUTING

ABS Model In PVS

```
% Whole system run, this can only be called after
% INIT and inputs setting
run(p_sys: sys_type, p_Wheel_Speed: real, p_Vehicle_Speed:
real): sys_type =
  LET v_EpsIfZero: real = EpsIfZero(p_Vehicle_Speed,
p_Vehicle_Speed, eps) IN
  LET v_Divide: real =
   Divide(p_Wheel_Speed, v_EpsIfZero) IN
  LET v_Relative_Slip: real =
   Relative_Slip(One, v_Divide) IN
  LET v_Use_ABS: real = Use_ABS(v_Relative_Slip) IN
  LET v_Difference: real =
   Difference(Desired_relative_slip, v_Use_ABS) IN
  LET v_Bang_bang_controller_sys:
   Bang_bang_controller.sys_type =
    Bang_bang_controller.run(p_sys`f_state
      `f_Bang_bang_controller_sys, v_Difference) IN
  LET v_Apply_Brakes: real =
Apply_Brakes(v_Bang_bang_controller_sys
                            `f_output`f_out_1_i2) IN
  (#
    f_state :=
      prepare_state(v_Bang_bang_controller_sys),
    f_output := prepare_output(v_Apply_Brakes)
   #)
```

DEPENDABLE COMPUTING

Properties

- **Property 1 specification:**

Property 1

```
Applies_Brake_When_Not_Slipping: THEOREM
 FORALL (v_sys: ABS_Controller_oem.sys_type,
         vSpeed: {v: nzreal | v >= (2 ^ -52)},
         wSpeed: {w: real | w > 0.8 * vSpeed }):
   f_Apply_Brakes(f_output(run(v_sys, wSpeed,
                               vSpeed))) = 1
```

- **Property 2 specification:**

Property 2

```
Lets_Off_Brake_When_Slipping: THEOREM
 FORALL (v_sys: ABS_Controller_oem.sys_type,
         vSpeed: {v: nzreal | v >= (2 ^ -52)},
         wSpeed: {w: real | w < 0.8 * vSpeed }):
   f_Apply_Brakes(f_output(run(v_sys, wSpeed,
                               vSpeed))) = -1
```

- **Property 3 specification:**

Property 3

```
Does_Neither_At_Threshold: THEOREM
 FORALL (v_sys: ABS_Controller_oem.sys_type,
         vSpeed: {v: nzreal | v >= (2 ^ -52)}):
   f_Apply_Brakes(f_output(run(v_sys, 0.8 * vSpeed,
                               vSpeed))) = 0
```

Properties

- Property 4 specification:

```
Applies_Brake_When_At_Rest_And_Wheels_Stopped: THEOREM
  FORALL (v_sys: ABS_Controller_oem.sys_type):
    f_Apply_Brakes(f_output(run(v_sys, 0, 0))) = 1
```

Property 4

Proof Failed

Wheel speed

Vehicle speed

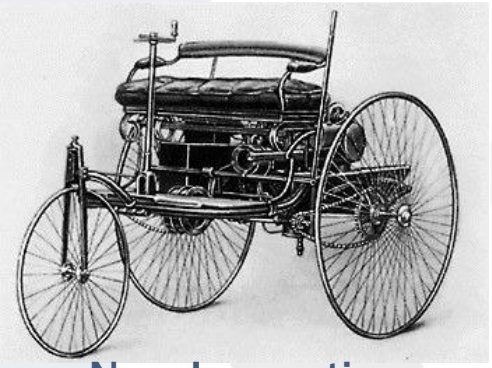
- Opposite of Property 4:

```
Releases_Brake_When_At_Rest_And_Wheels_Stopped: THEOREM
  FORALL (v_sys: ABS_Controller_oem.sys_type):
    f_Apply_Brakes(f_output(run(v_sys, 0, 0))) = -1
```

Proved

Recall That Model Came From MathWorks Web Site

Another Problem



New Innovative Concept

Model Based Development
Simulink, SCADE, Etc.

OEM Model
Design Environment

Specification

Supplier Model
Development Environment

Equivalent?

PRODUCTION

System

Implementation

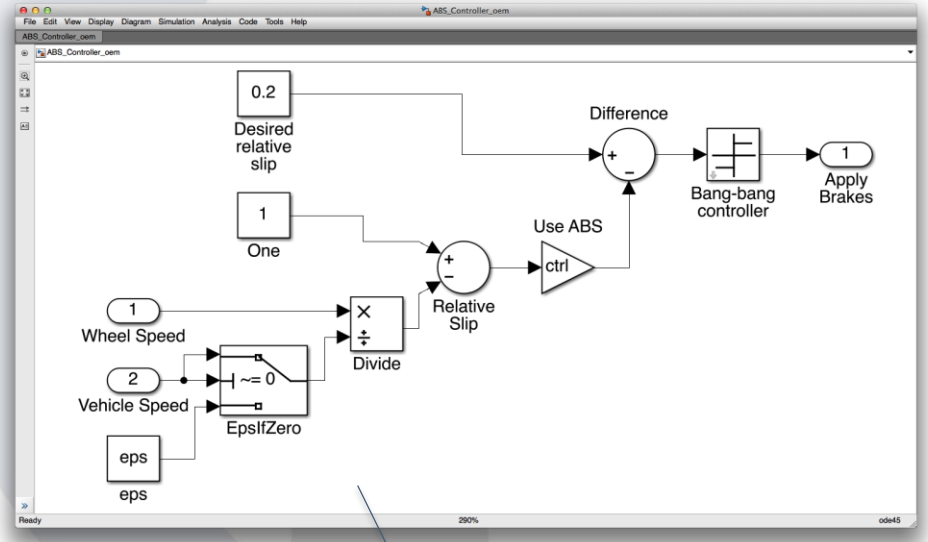
Sub System

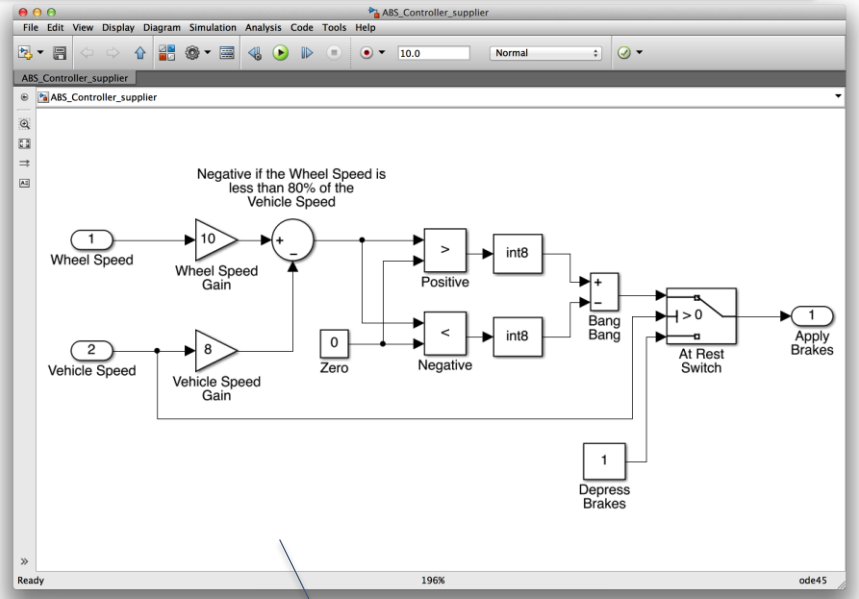
Problem Details

- Development models rarely take account of practical limitations of target platforms
- Production models must address these limitations
- Example:
  - Development model uses 32-bit integers
  - Target platform used for the production model only supports 16-bit integers
- Difference means two models will not be identical
- Such differences are common in engineering
- How can “equivalence” be established?
- What does “equivalence” mean?

DEPENDABLE COMPUTING

Example - ABS





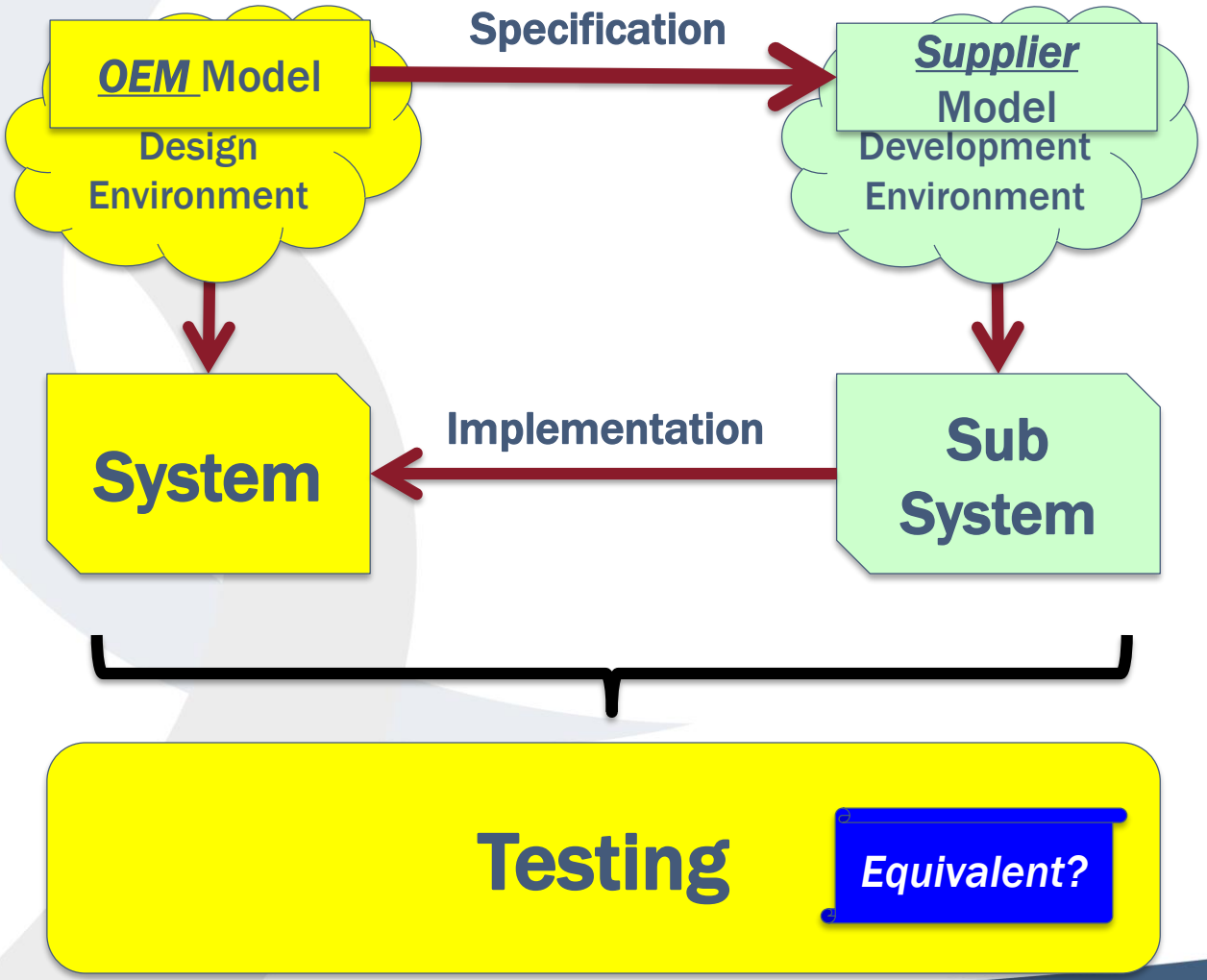
Hypothetical OEM Model

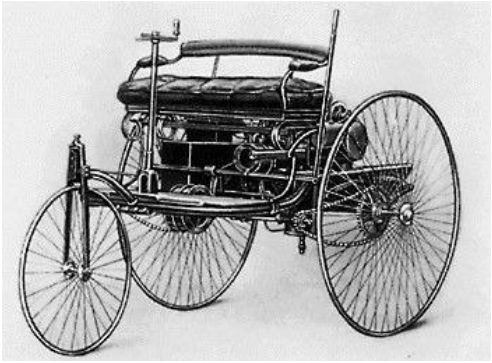
Hypothetical Supplier Model

Are They Equivalent?

DEPENDABLE COMPUTING

Assurance of "Equivalence"





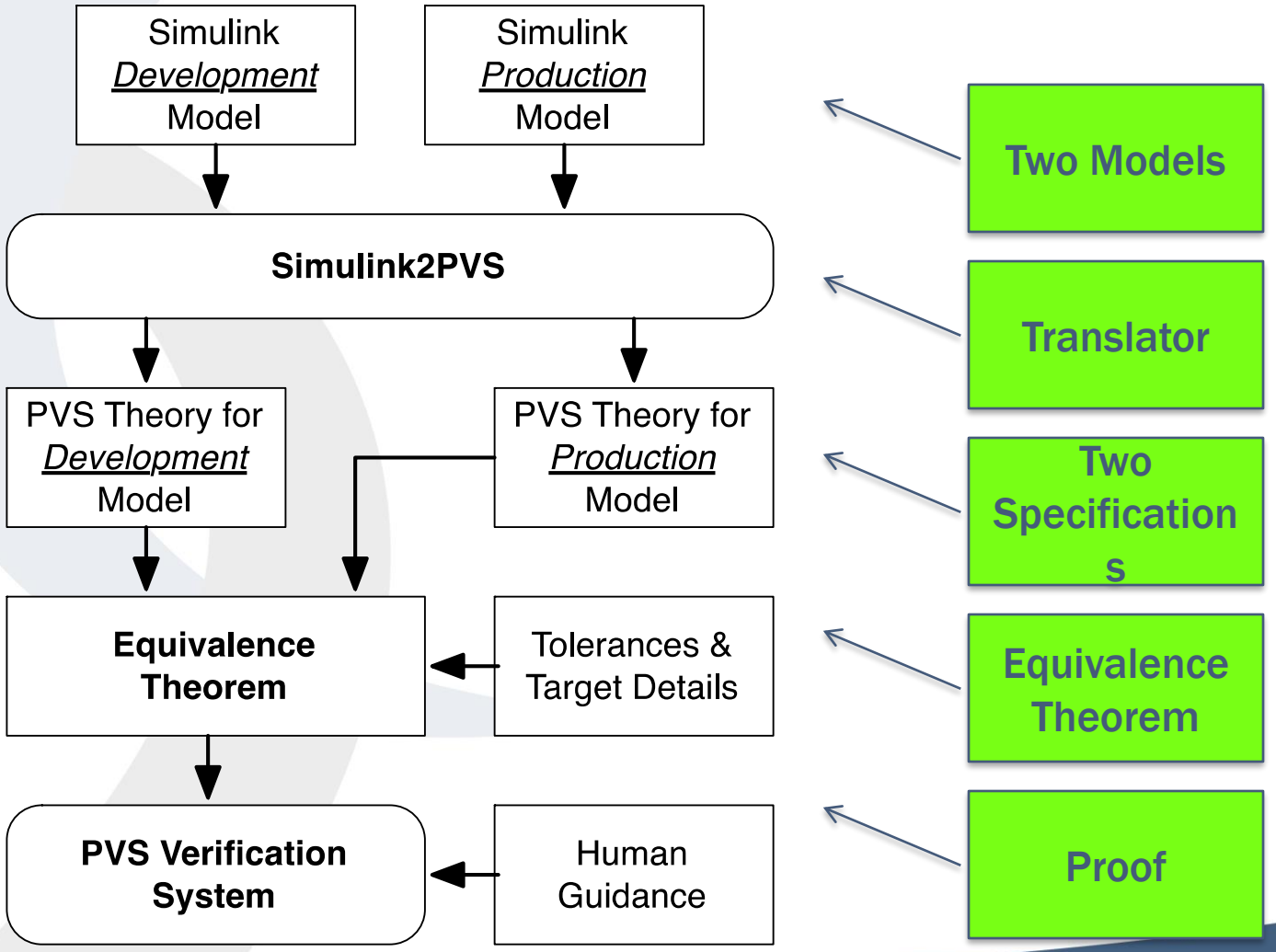
DEPENDABLE COMPUTING

Defn: *Constrained Equivalence*

- Two models exhibit ***constrained equivalence*** if:
  - All valid inputs to the first model produce the same output in the second model to within a specified tolerance
  - Inputs to the second model that are within a specified tolerance of the inputs to the first model produce the same output
- Predefined scaling factors and offsets might be used in determining whether two factors are the same

DEPENDABLE COMPUTING

Equivalence Proof System



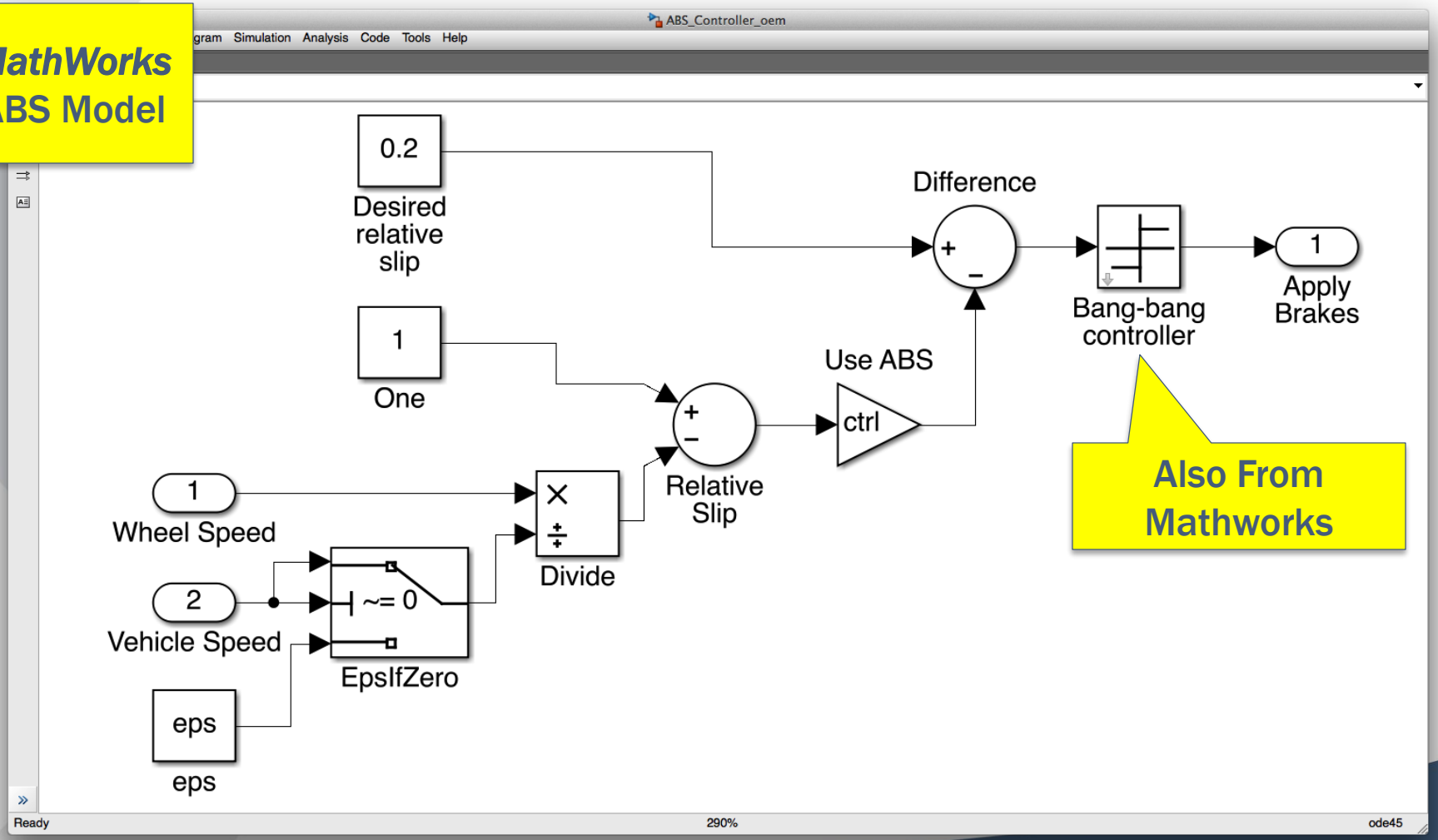
Exploratory Study

- Model of an automobile anti-lock brake system (ABS) controller
- Derived from an ABS model published by MathWorks
- Model:
  - Serves as the development model in the study
  - Relies on a “bang-bang” controller published separately by MathWorks
  - ABS logic is only valid when the driver is depressing the brake pedal

DEPENDABLE COMPUTING

ABS *Development* Model

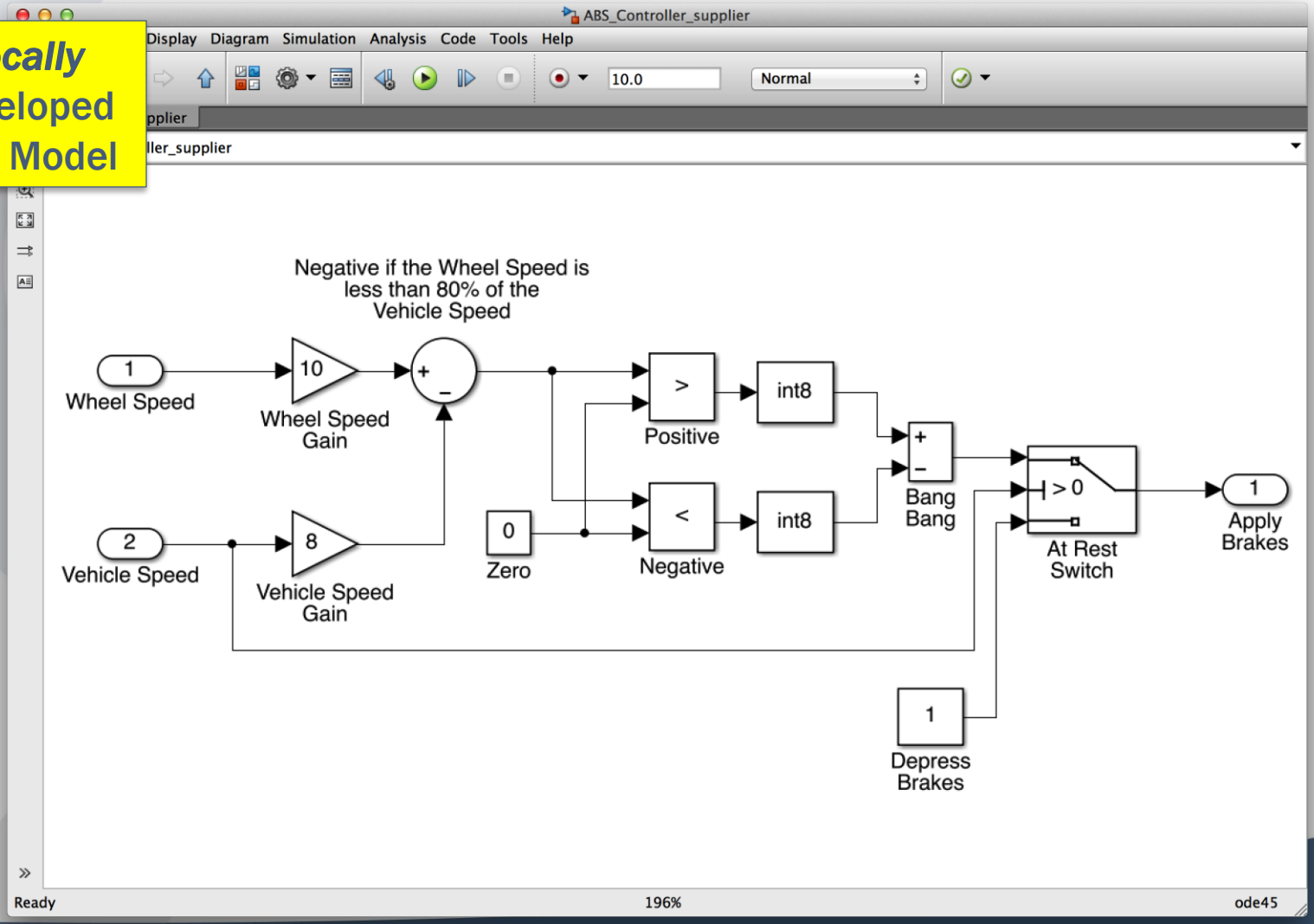
MathWorks ABS Model



DEPENDABLE COMPUTING

ABS *Production* Model

Locally **Developed ABS Model**



Computations

- *Development* model:
  - Apply brakes if

$$Wheel\ Slip < 0.2$$

  - where:

$$Wheel\ Slip = 1 - (Wheel\ Speed\ /Vehicle\ Speed)$$

- *Production* model:
  - Apply brakes if:

$$10 \times Wheel\ Speed > 8 \times Vehicle\ Speed$$

- Identical *provided*:

$$Vehicle\ Speed\ /= 0$$

DEPENDABLE COMPUTING

Constrained Equivalence Theorem

```
Constrained_Equivalence: THEOREM
    FORALL (v_sys: ABS_Controller_production.sys_type,
        vSpeed: {i: nonneg_int32 | i <= 100000},
        wSpeed: {i: nonneg_int32 | i <= 100000}):
    f_Apply_Brakes(f_output(ABS_Controller_development.run
        (conv_sys(v_sys), vSpeed / 100, wSpeed / 100))) =
    f_Apply_Brakes(f_output(ABS_Controller_production.run
    (v_sys, vSpeed, wSpeed)))
```

- **Predicate states:**

  Application of brakes by the two models equivalent for vehicle and wheel speeds with integer values in range 0 to 100,000

- Divisions by 100 in development model are scale factors necessary to align the speed measurement units
- Integer values are meaningful, because data supplied by speed sensors are discrete

Theorem Proof

- Proof by parts:
  - Car moving and wheels not slipping
  - Car moving and wheels are slipping
  - Car at rest
- And the envelope please….

  As before, theorem is false
- Informally:
  - Stationary, wheels not moving, want brakes on
  - ***Development*** model does not do this – ***error***
- Easily analyzed, easily fixed
- Not necessarily easy to find.

DEPENDABLE COMPUTING

Conclusion

- Demonstration of model equivalence is necessary element of engineering
- Problem arises from inevitable separation of:
  - Design/development engineering
  - Production engineering
- “Constrained equivalence” provides basis
- Mechanical proof has been demonstrated to be:
  - Feasible
  - Probably cost effective compared to testing

U.K.

U.S.

Questions?