Approximate Formal Verification Using Model-Based Testing

Rance Cleaveland

Department of Computer Science

UMIACS
University of Maryland Institute for Advanced Computer Studies

UNIVERSITY OF MARYLAND

The Institute for Systems Research

What This Talk Is

- … a position statement
- … a discussion of a research program
- … a review of some things I have been working on over the past decade

Formal Methods

- Mathematically rigorous approaches to specifying, verifying systems
- Why? To increase confidence!
  - If the specification is trusted, verification yields trust in system
  - If specification is not trusted, proving it is consistent with system builds trust in both

The Elements of Formal Methods

- Formal semantics of systems

  Systems must be mathematical objects!

- Formal specifications

  Mathematical descriptions of desired behavior

- Formal verification

  Proofs that systems satisfy specifications

Verification = Proof

- Model checking

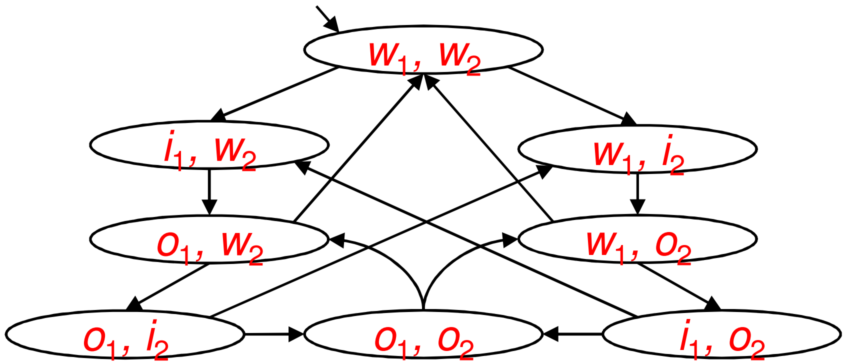
  Proof constructed “automatically”

- Theorem proving

  Proof constructed “automatedly”

Example: Temporal Logic

- Systems: Kripke structures



- Specifications: Temporal Formulas
  - E.g. $AG\ (\neg i_1 \lor \neg i_2)$
  - "It is always the case that either i_1 or i_2 is false."
- Verification: Model checking

Another Example: Design-By-Contract

- Systems: code (class definitions)
- Specifications
  - Pre / postconditions
  - Invariants
  - `asserts`
- Verification: Theorem proving

Status of Formal Methods

- Noteworthy successes!
- We are not at the stage where success is expected

Why?

- "Scalability"

  Building proofs is laborious, even for machines

- Inability to predict level of effort
  - Difficulty of proof not correlated to usual measures of system complexity
  - Work needed to coax proof out of tools not easy to estimate
- Need for highly trained (= expensive) workforce

My Perspective

- Proving is hard, but guarantees are very strong
- If proofs are not possible / feasible
  - *Must test* to conduct V&V
  - Benefits of formal specifications are difficult to explain in this case
- "Prove If You Can, Test If You Cannot" (PIYC/TIYC)

We should focus on formal specifications that support proof *and* testing!

PIYC / TIYC

- "Pick-tick"
  - Prove If You Can.
  - Test If You Cannot.
- A formalism supports PIYC / TIYC if
  - Full formal verification can be undertaken
  - So can less complete V&V
    - Testing
    - Inspections
    - Etc.
- In other words: full and *approximate* verification are both possible

What This Talk Is About

- PIYC / TIYC in practice
  - Model-based testing (MBT)
    - Models used as software specifications
    - MBT used to check equivalence between specs, software
  - Instrumentation-Based Verification (IBV)
    - Specifications given in same notation as software
    - Verification = instrument software, check for errors
- Context
  - Automotive control software
  - MATLAB® / Simulink® / Stateflow® / Reactis®

Talk Agenda

- Automotive software and MBD
  - MATLAB / Simulink / Stateflow
  - Verification in MBD
- MBT (using Reactis®)
- IBV (also using Reactis)
- Conclusions

Some Software Companies

- ATI Accurate Technol…
- Philips
- ADVICS — Providing safety and comfort on roads around the world
- Suzuki
- Caterpillar
- Guidant
- Hyundai
- Toyota
- AV
- MBDA
- Whirlpool
- Boeing
- Pratt & Whitney — A United Technologies Company
- Lockheed Martin
- Honda
- Ford
- Hella — Ideas today for the cars of tomorrow
- Magna Steyr — more value • more car
- Aisin
- Bombardier
- Mazda
- Audi
- SAIC
- John Deere
- Hitachi — Inspire the Next
- Land Rover
- BAE Systems
- Continental Temic
- Cummins
- Fujitsu
- GE
- Motorola
- General Dynamics
- Raytheon
- Denso
- bertrandt
- National Aeronautics and Space Administration (NASA)
- IDI Technology
- United Technologies
- BMW — Sheer Driving Pleasure
- dSPACE
- Detroit Diesel Corporation
- GM
- Honeywell
- Kia Motors
- Renault
- Siemens
- Delphi — Tomorrow's Technology
- Elbit Systems
- Jatco
- Volvo
- Nissan
- Visteon
- Northrop Grumman
- Mitsubishi Heavy Industries, Ltd.
- Daimler
- DAF
- Rockwell Collins
- smiths
- The MathWorks
- Titan — National Security Solutions
- TRW

Automotive Software

- Driver of innovation

  90% of new feature content based on software [GM]

- Rising cost

  50% of Prius cost due to software [Toyota]

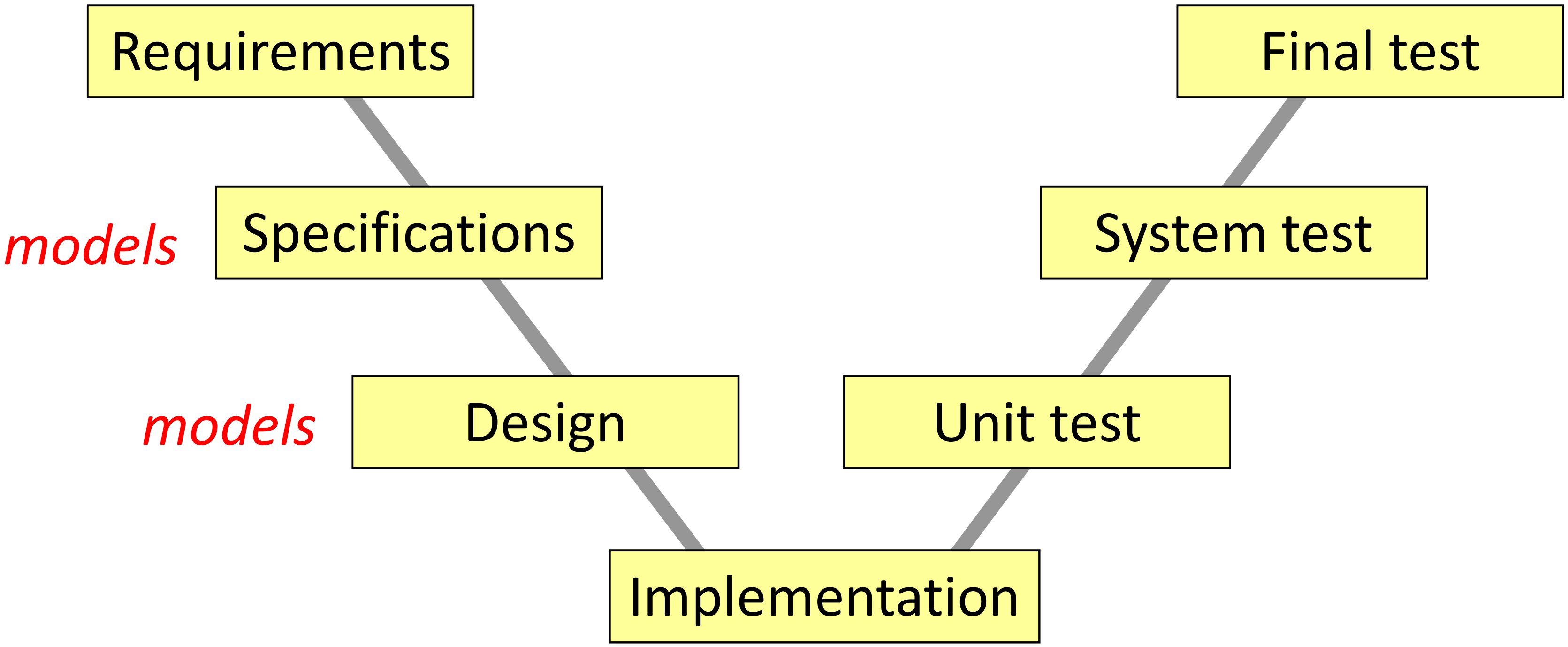
- Warranty, liability, quality

  High-profile recalls in Germany, Japan, US

A Grand Challenge

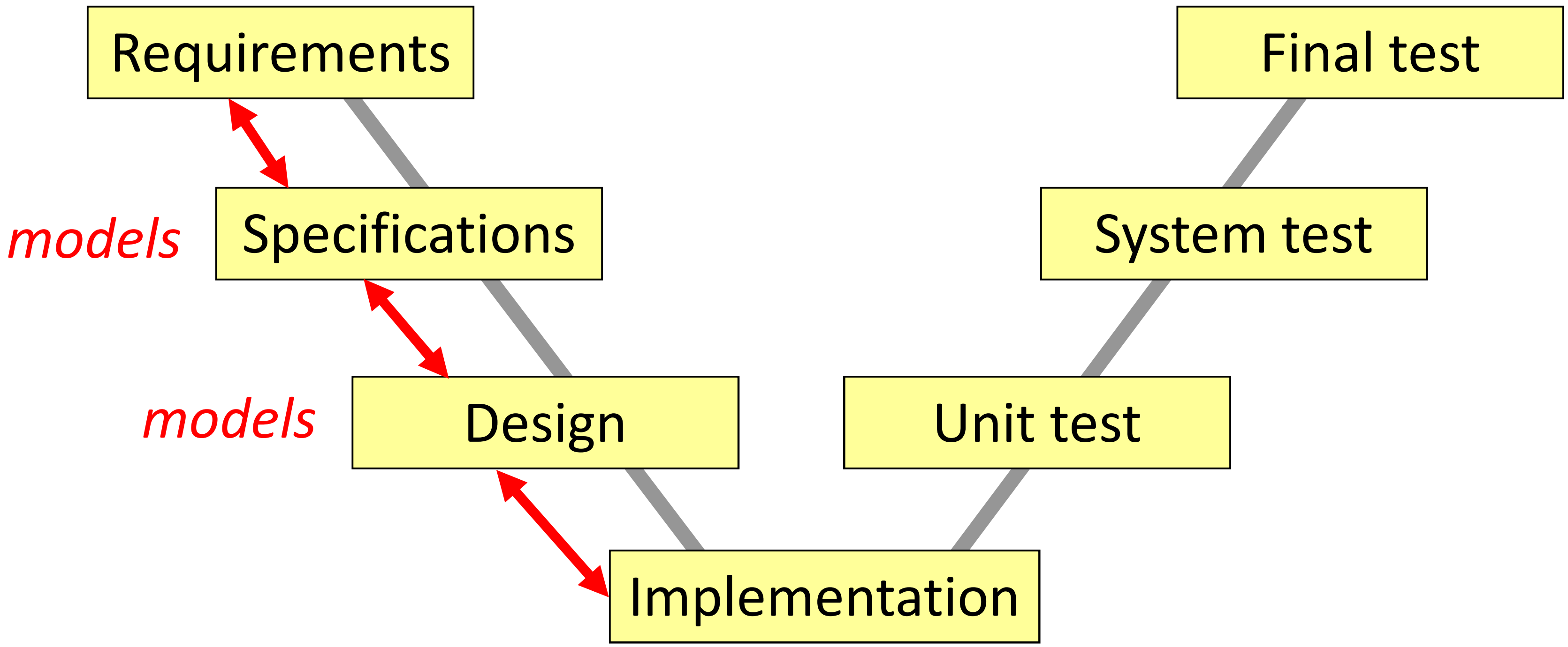
- Ensure high quality of automotive software while
  - ... preserving time to market
  - … containing cost
- Key approach: *Model-Based Development* (MBD)
  - Use executable models during development
  - Dominant language: MATLAB / Simulink / Stateflow

Model-Based Development



Main Motivation: *autocode*

More Benefits of MBD

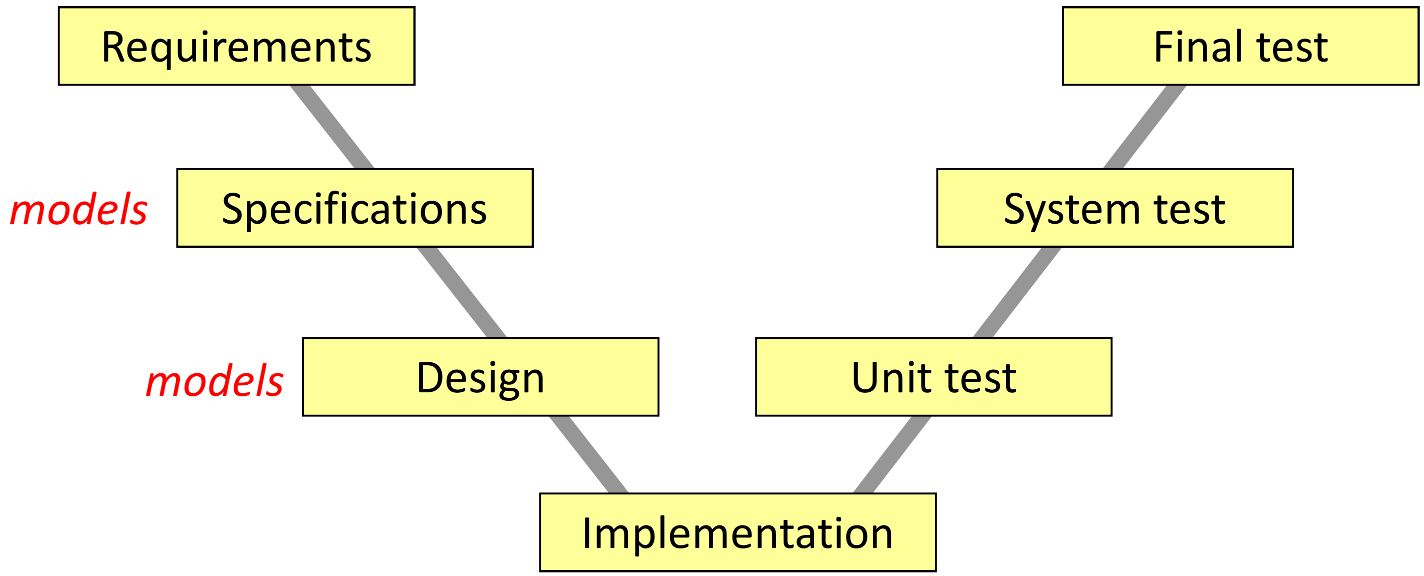


Models formalize specifications, design

Models facilitate communication among teams

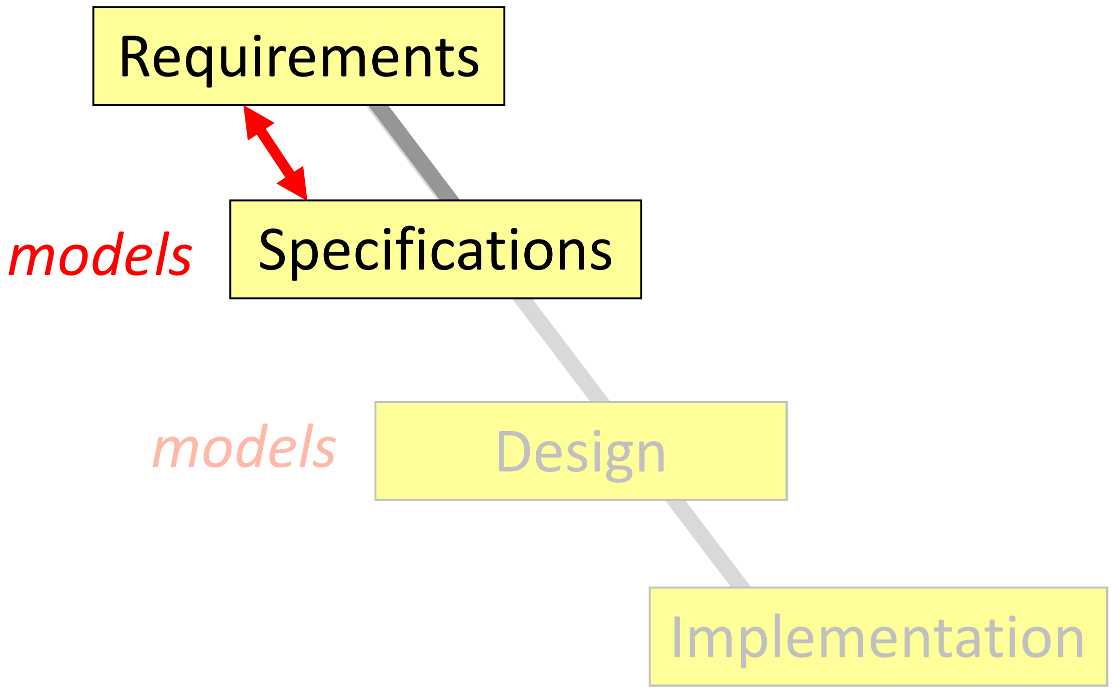
Models support V&V, testing

A Sample Automotive MBD Flow



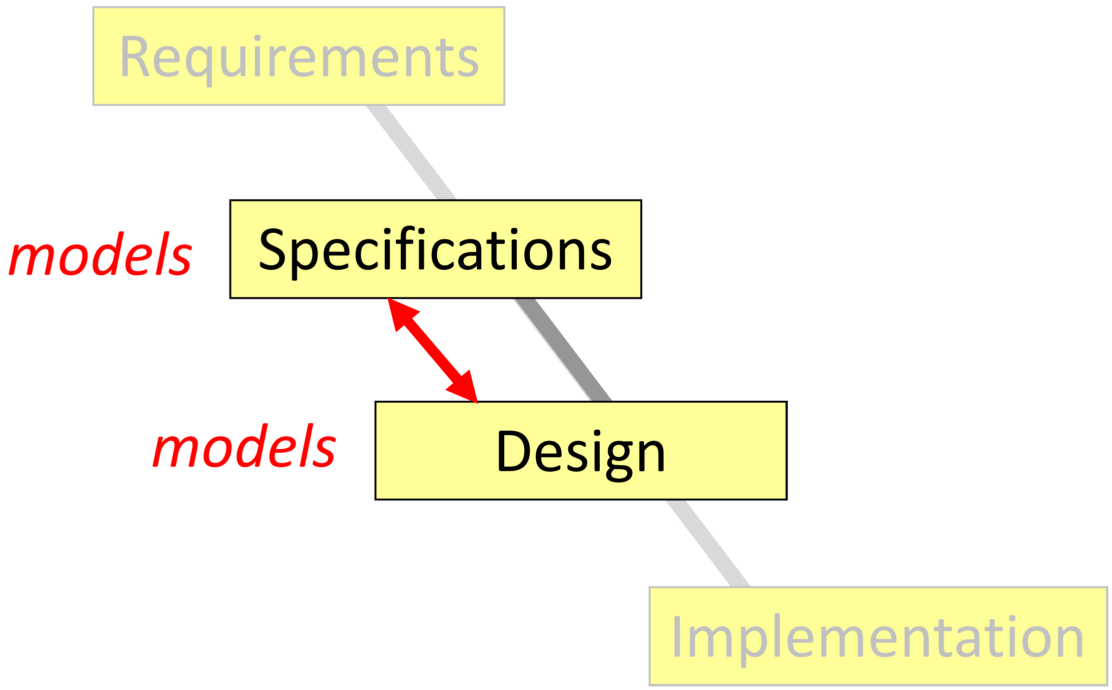
Requirements	Documents
Specifications	Floating-point models from controls engineers
Designs	Fixed-point models from platform engineers
Implementation	C from autocoding, software developers
Testing	Hardware-in-the-loop (HIL) testing from test engineers

MBD Verification Problem #1



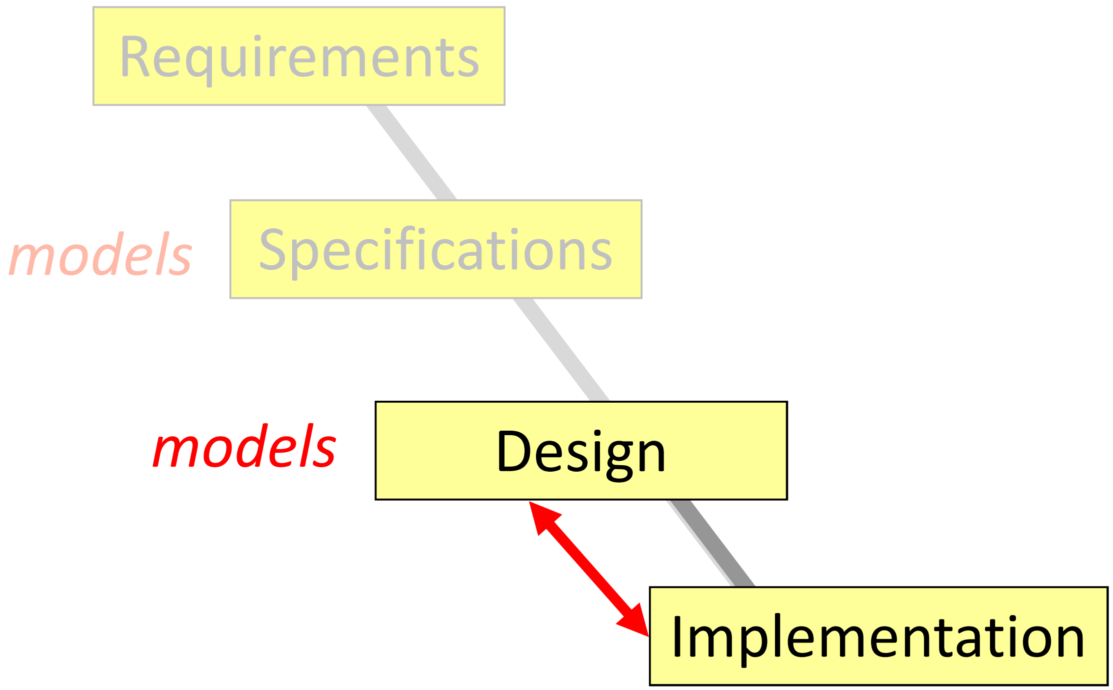
Do specifications satisfy requirements?

MBD Verification Problem #2



Does design meet specifications?

MBD Verification Problem #3



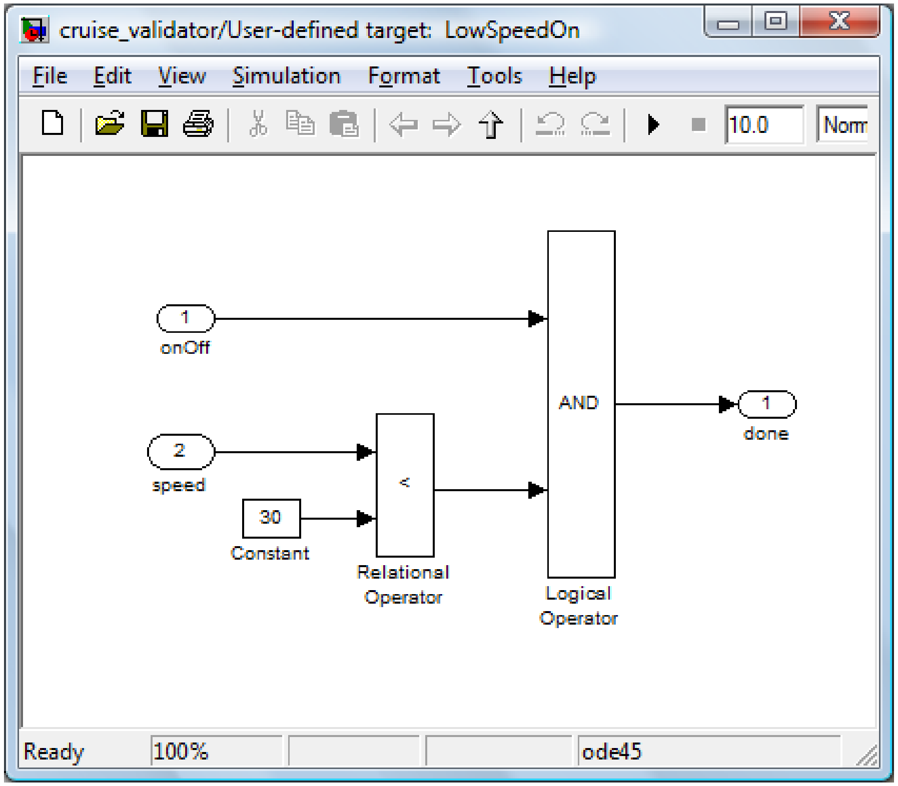
Does implementation meet design?

PIYC / TIYC for MBD

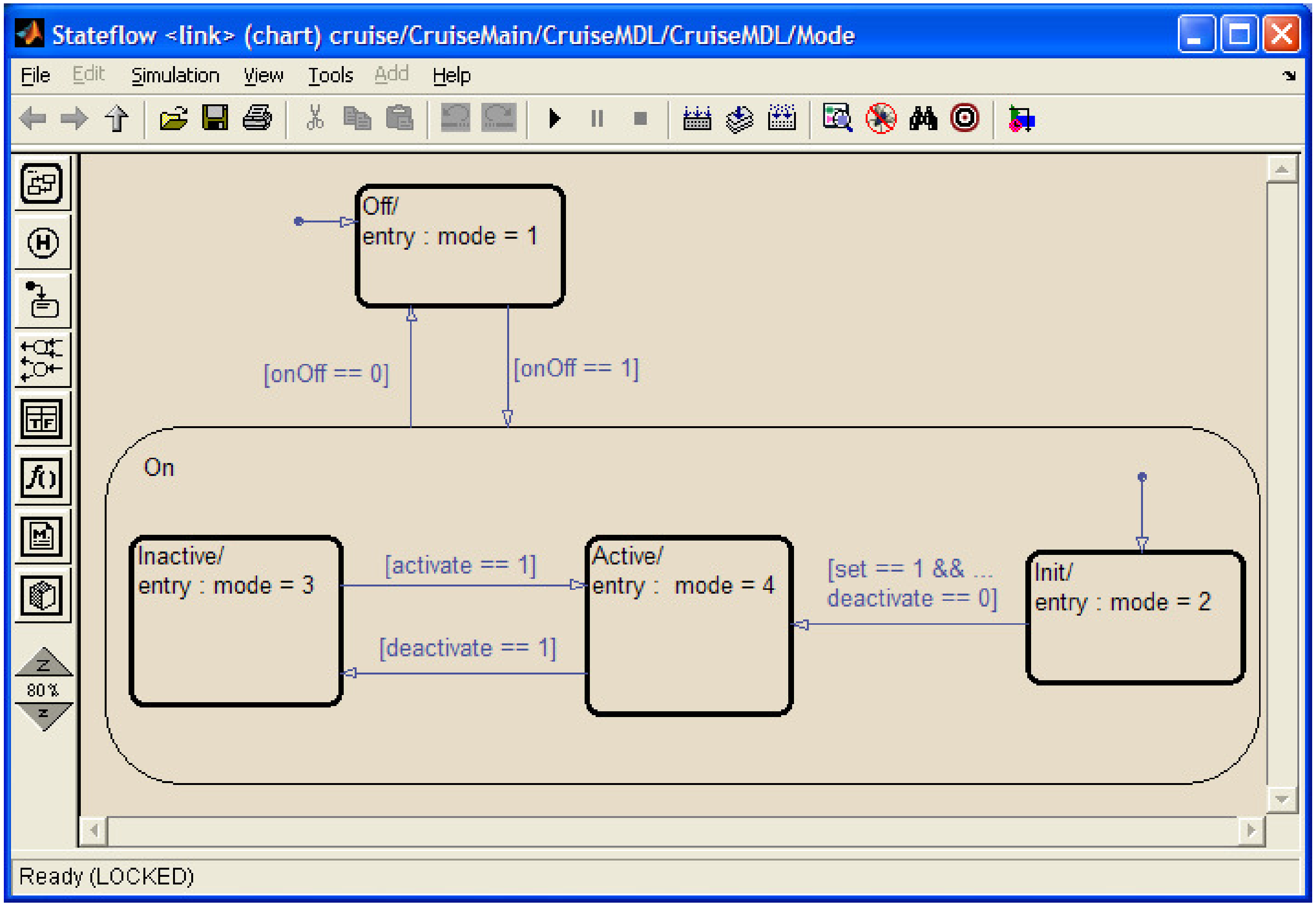
- Formalize verification problems mathematically
  - Formal semantics of systems
  - Formal specifications
  - Formal definition of satisfaction
- Give testing-based *approximate* verification strategies

Simulink

- Block-diagram modeling language / simulator of The MathWorks, Inc.
- Hierarchical modeling
- Continuous- and discrete-time simulation



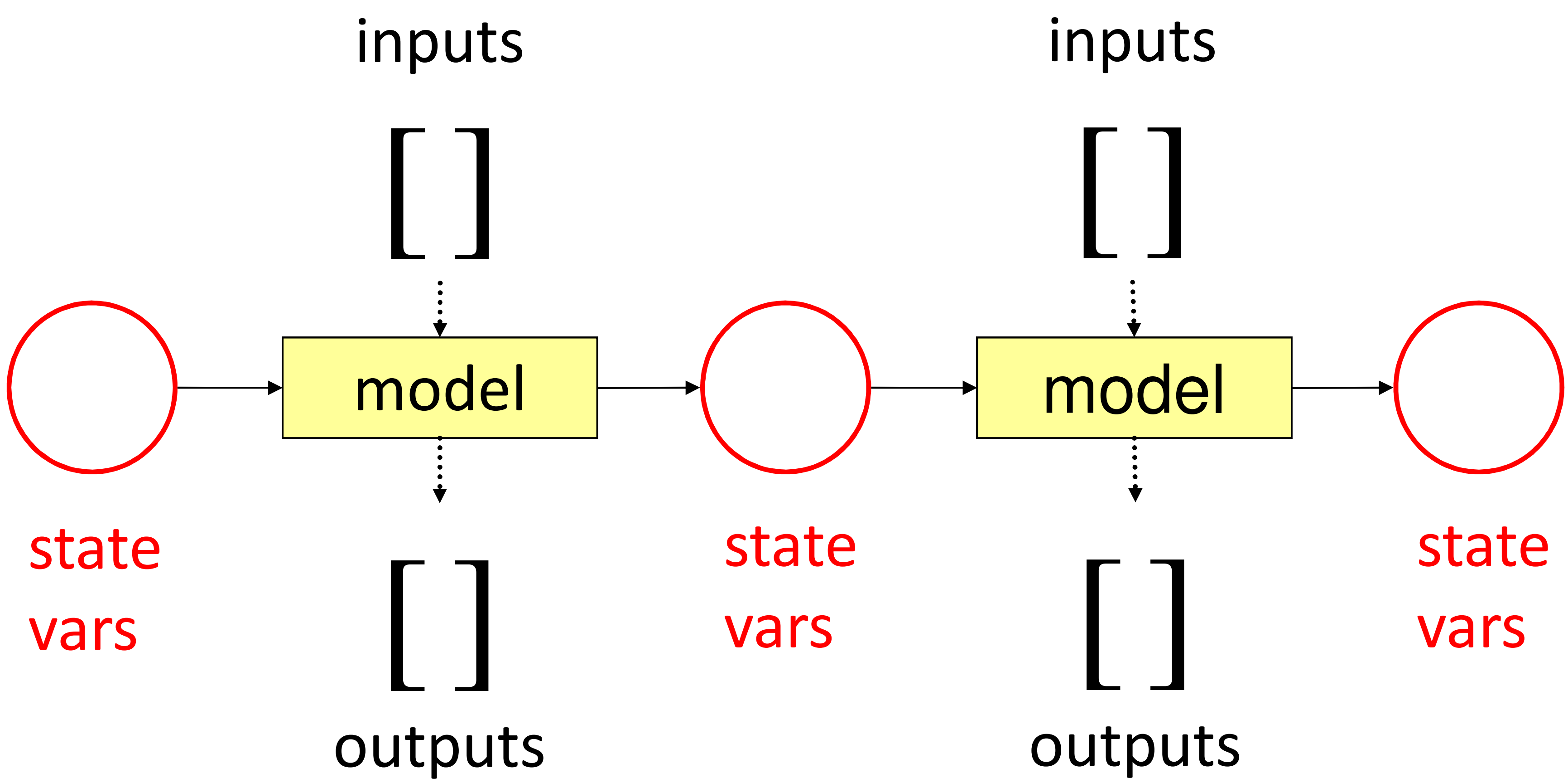
Stateflow


Stateflow <link> (chart) cruise/CruiseMain/CruiseMDL/CruiseMDL/Mode
File Edit Simulation View Tools Add Help
Off/
entry : mode = 1
[onOff == 0]
[onOff == 1]
On
Inactive/
entry : mode = 3
[activate == 1]
[deactivate == 1]
Active/
entry : mode = 4
[set == 1 && ...
deactivate == 0]
Init/
entry : mode = 2
80%
Ready (LOCKED)

Semantics

- Simulink has different “solvers” (= semantics)
  - Continuous: inputs / outputs are signals
  - Discrete: inputs / outputs are data values
- Physical modeling: continuous solvers
- (Digital) controller modeling: discrete solvers
  - Synchronous
  - Run-to-completion
  - Time-driven

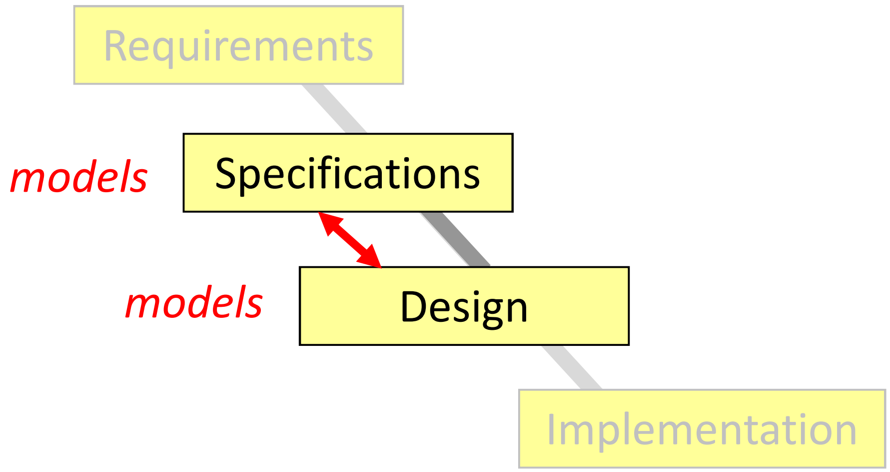
Example


inputs
[ ]
model
state
vars
[ ]
outputs
inputs
[ ]
model
state
vars
[ ]
outputs
state
vars

Discrete Simulink Semantics

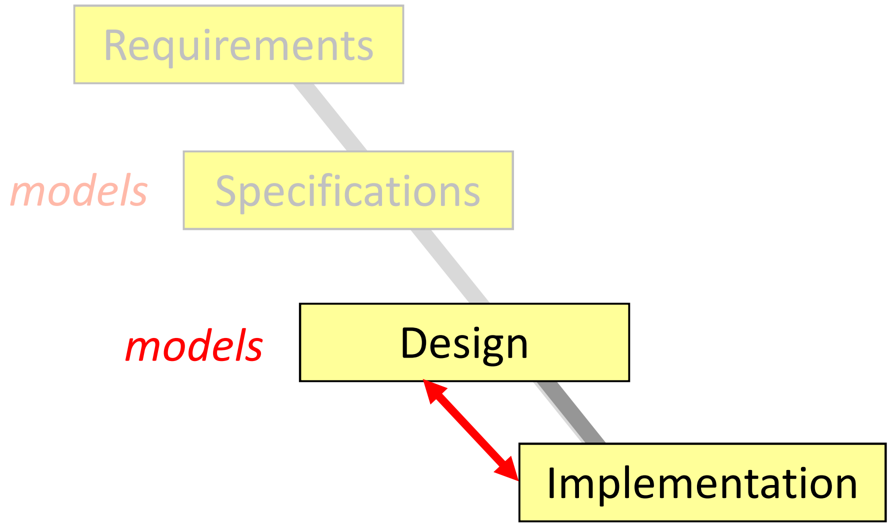
- Simulink models are (deterministic) Mealy machines
  - States are assignments of values to state variables
  - Transitions are computed by model
- Can thus speak of *language* of model *M*
  - I = set of possible input vectors for *M*
  - O = set of possible output vectors for *M*
  - L(*M*) = {*w* ∈ (I x O)* | *w* is (timed) sequence of transition labels of execution of *M* }

Formalizing MBD Problem #2



- Specification, design models are both Mealy machines
- MBD Problem #2
  - Given: (spec) model *S*, (design) model *D*
  - Determine: does L(*S*) = L(*D*)?
  - Note: some mappings between sequences in L(S), L(D) may be needed (e.g. if *S* is floating point, *D* fixed point)

Formalizing MBD Problem #3



- Semantics of implementation *I* needs to yield Mealy machine also!
- MBD Problem #3
  - Given: (design) model *D*, implementation *I*
  - Determine: does L(*D*) = L(*I*)?
  - Note: some mappings between sequences in L(D), L(I) may be needed

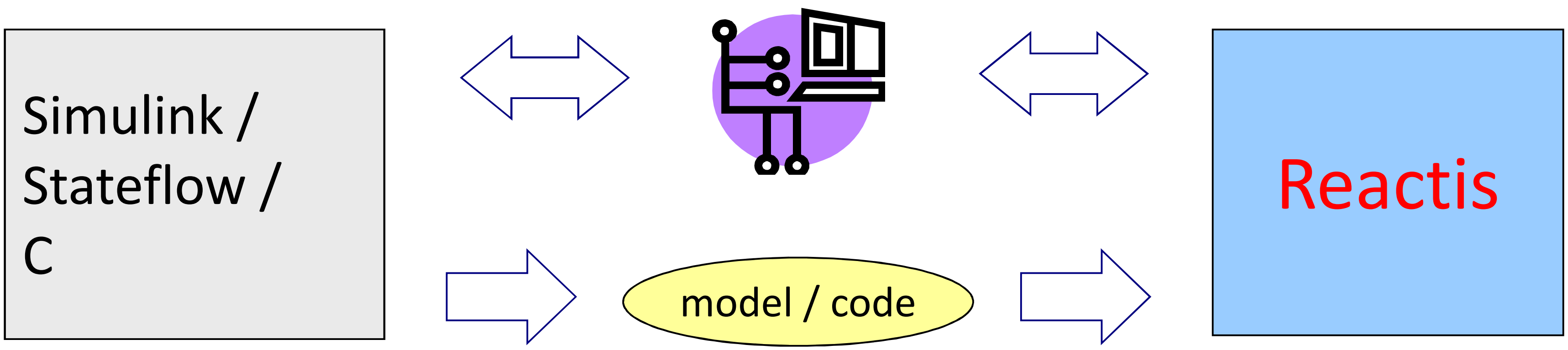
PIYC / TIYC for Problem #2 (and #3)

- Can prove instances of Problem #2
  - *S*, *D* are deterministic Mealy machines
  - Can use language-equivalence checkers to compute L(*M*) = L(*S*)
  - Not done in practice because state spaces too big
- *Approximate verification*: use testing
  - Standard model-based testing
    - Generate test cases from *S*
    - Run them on *D*
    - Compare outputs
  - "Back-to-back" testing (e.g. ISO 26262)
    - Do MBT
    - Also, generate tests from *D*, run them on *S*, compare results

Reactis®, Reactis for C

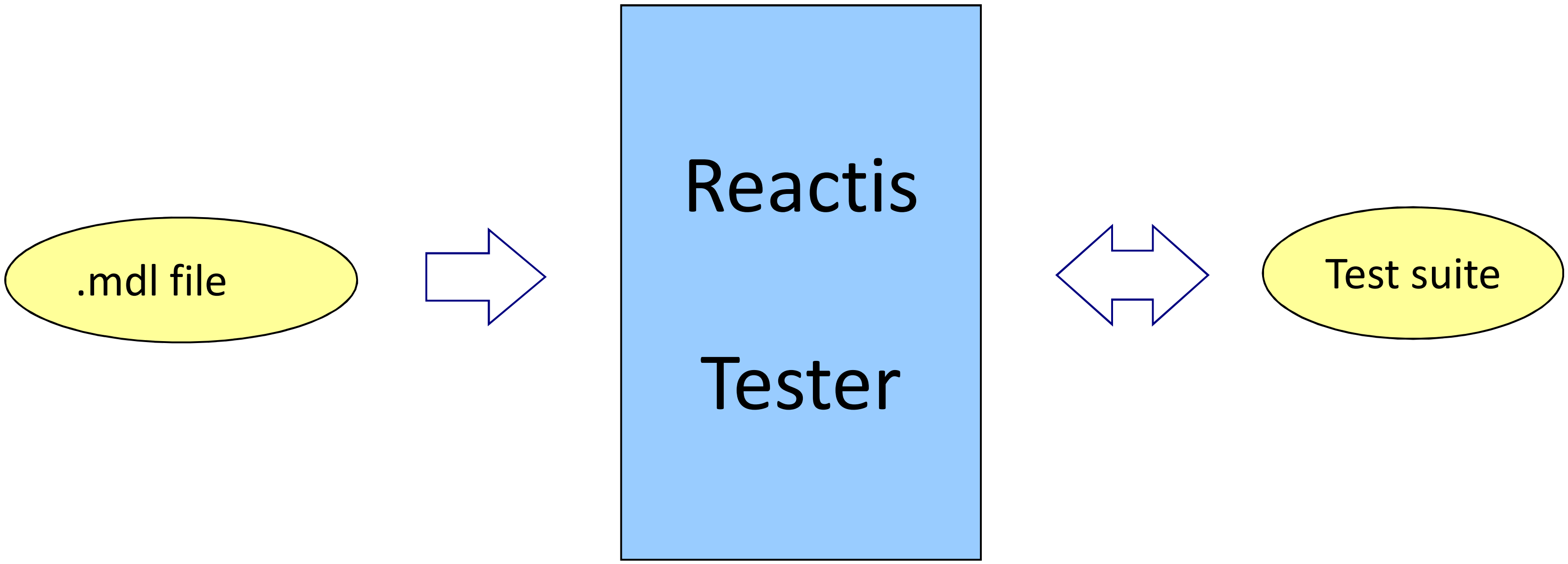
Automatic testing tool from Reactive Systems Inc.

Tester	Generate tests from models, C code
Simulator	Run, fine-tune tests
Validator	Validate models / code

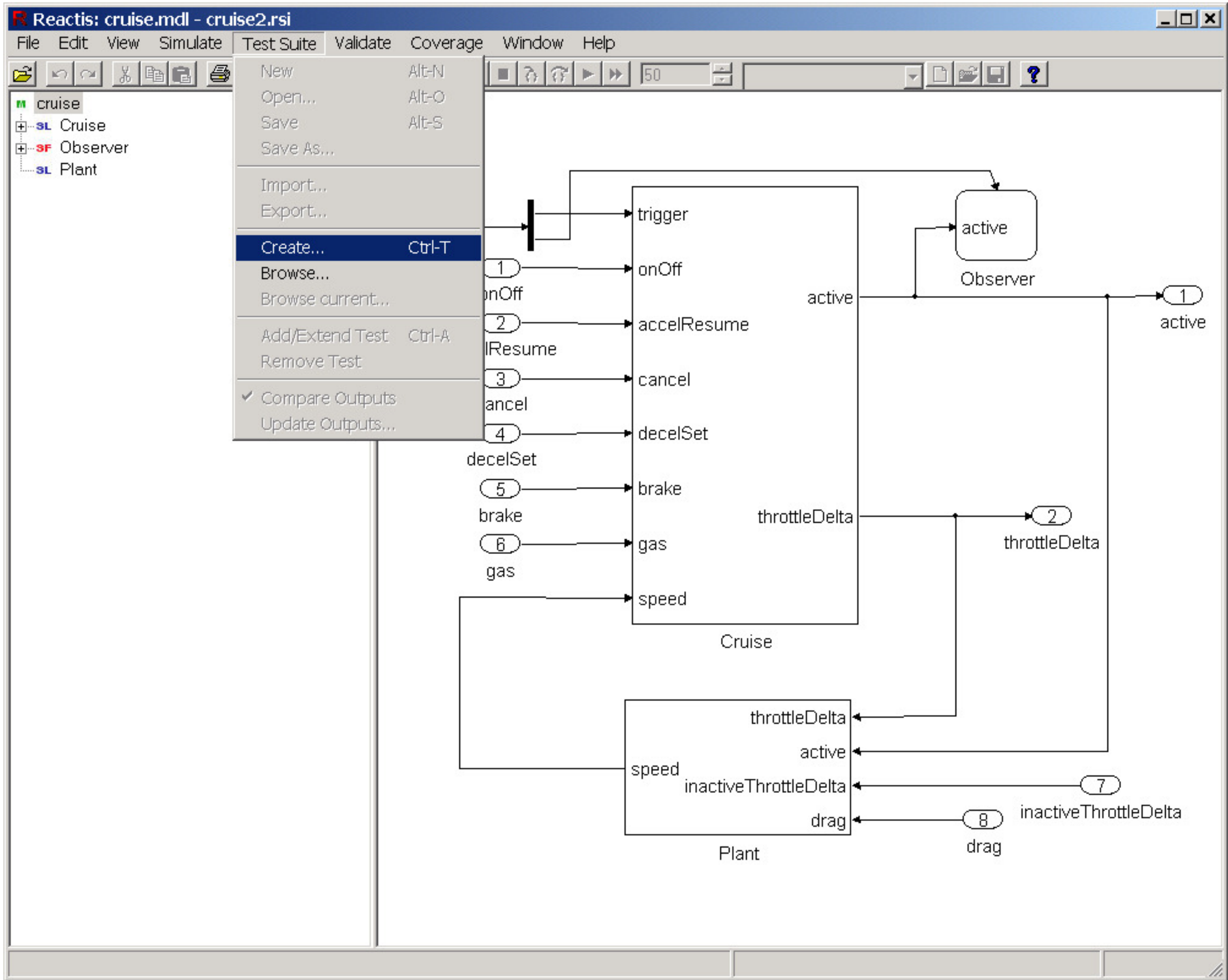


Reactis Tester

- Model / code in; tests out
- Model / code, tests in; better tests out



Launching Tester



Generated Test Data

Reactis Test-Suite Browser: cruise.rst

File View Help

Test 2 (5 steps)

Port	Step 1	Step 2	Step 3	Step 4	Step 5
Inputs					

1: onOff	0.0	1.0	0.0	1.0	1.0
2: accelResume	0.0	1.0	1.0	1.0	1.0
3: cancel	1.0	0.0	0.0	1.0	1.0
4: decelSet	0.0	0.0	1.0	0.0	1.0
5: brake	1.0	1.0	0.0	1.0	0.0
6: gas	1.0	0.0	1.0	0.0	1.0
7: inactiveThrottleDelta	0.1	0.0	0.1	-0.1	0.0
8: drag	-0.0093...	-0.0089...	-0.0094...	-0.0088...	-0.0089...
Outputs					

1: active	0.0	0.0	0.0	0.0	0.0
2: throttleDelta	-0.1	0.0	-0.1	0.0	0.0
___t___	0.0	1.0	2.0	3.0	4.0

Configuration Variable	Value
InitialSpeed	15.79179838897

Test Generation with Reactis

- Test = simulation run = sequence of I/O vectors = element of L(*M*)
- Goal: maximize model coverage (e.g. branch, state, MC/DC, etc.)
- Method: guided simulation (US Patent 7,644,398)
  - Think: state-space search
    - Models = Mealy machines
    - Test generation = state-space traversal
    - Search termination condition: coverage of model (= transition computation)
  - Choose input data to guide search to uncovered parts of model (= transition computation)
    - Monte Carlo
    - Constraint solving (currently, linear constraints, SAT)

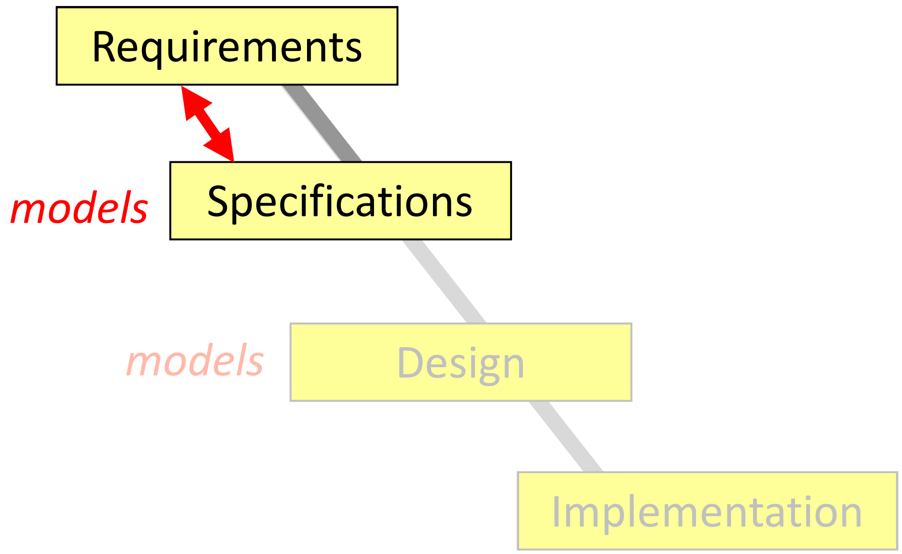
Experience

- Main use case for Reactis
- In use at 75+ companies around the world

Summary So Far

- PIYC / TIYC = "Prove If You Can / Test If You Cannot"
  - Formal specifications support both formal verification, testing
  - Testing can be viewed as "approximate verification"
- Two examples of PIYC / TIYC in model-based practice
  - The formalizations involve language equivalence
  - The testing-based approximations rely on structural coverage for termination

Formalizing MBD Problem #1



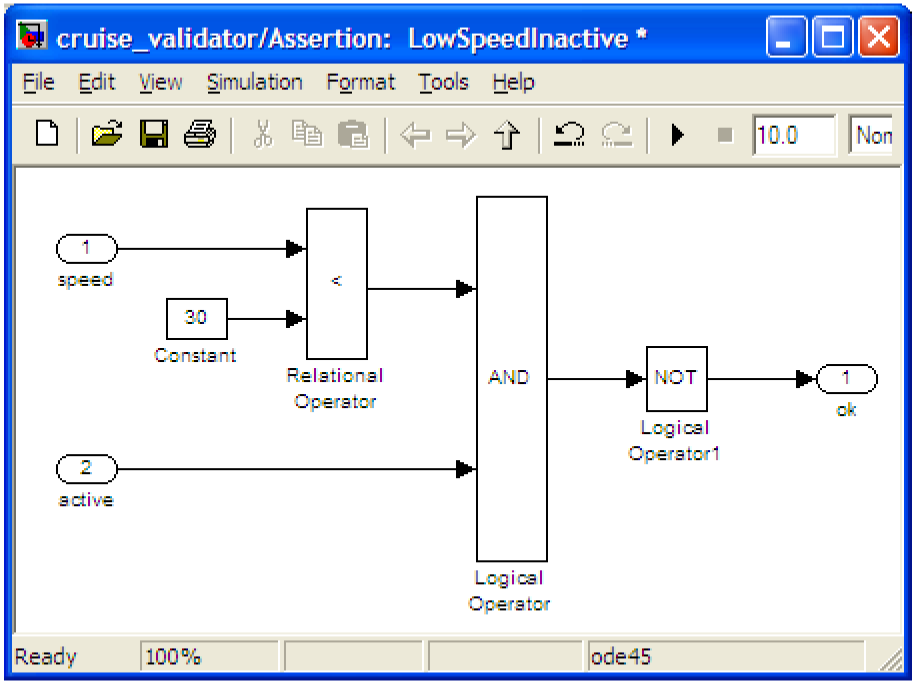
- We would like a PIYC / TIYC approach for this problem
- Need:
  - Formalized requirements
  - Formalized notion of satisfaction
- A useful idea: *Instrumentation-Based Verification*

IBV: Requirements

- Formalize requirements as *monitor models*

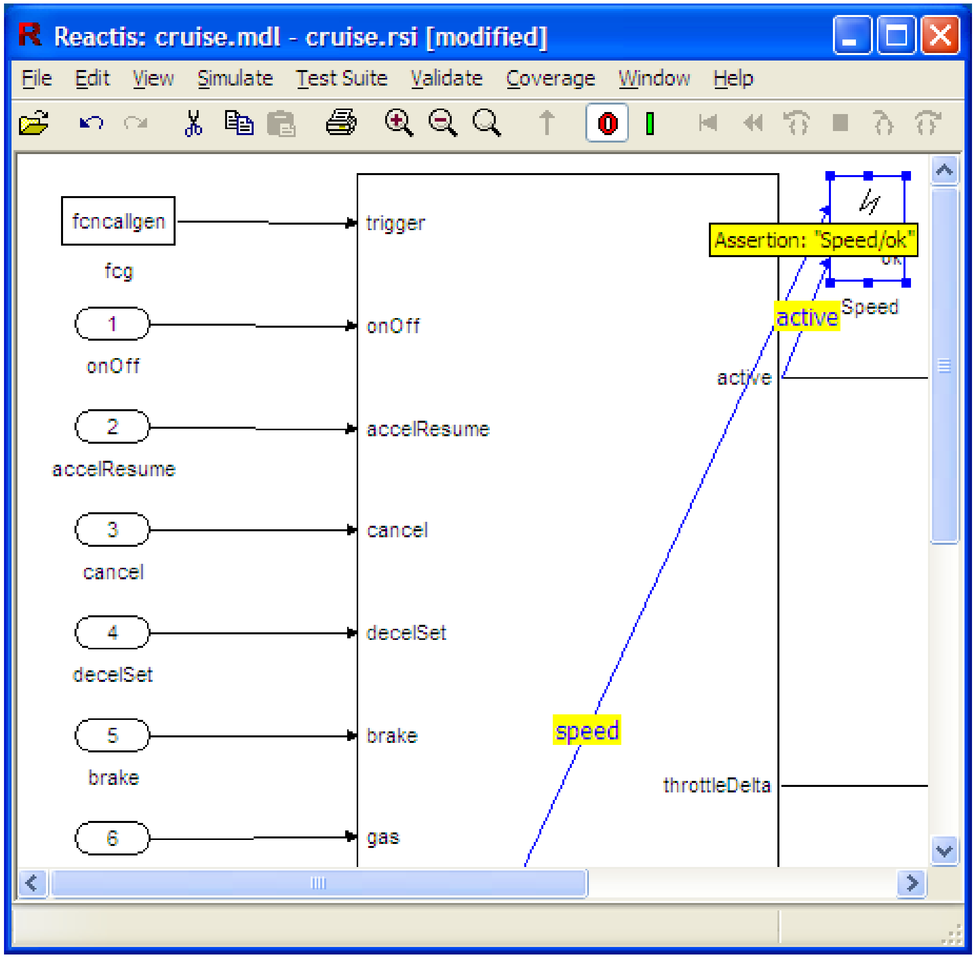
- Example

  If speed is < 30, cruise control must remain inactive



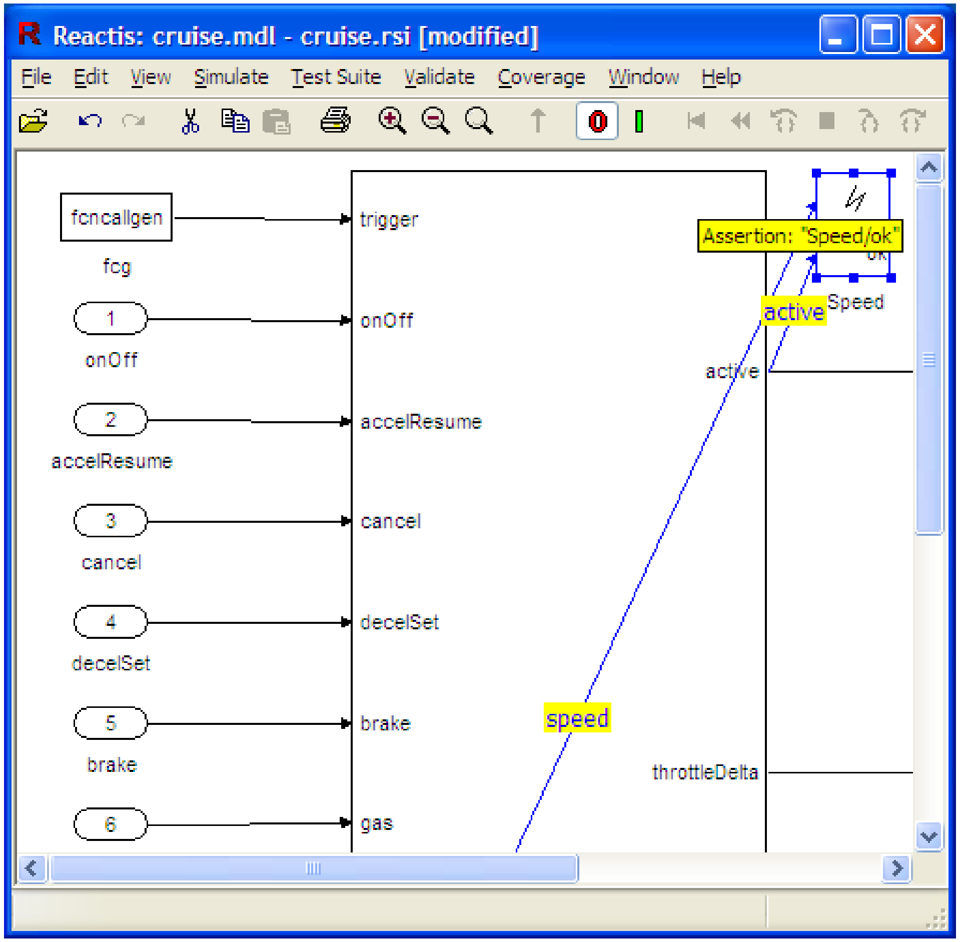
IBV: Satisfaction

- Instrument design model with monitors
- Model satisfies monitors if:
  - For every input sequence …
  - … every monitor model output remains *true*
- Reachability problem!
  - Proof possible
  - State space an issue



Approximate Verification for Problem #1

- Use coverage testing on instrumented model
  - Better scalability
  - If booleans part of coverage criteria:
    - Test generator tries to make monitor outputs false
    - Skeptical testing!
- Reactis
  - Supports instrumentation
  - Acts as skeptical tester
  - Reports violations



Related Work

- Run-time monitoring

  Havelund et al., Lee et al., Godefroid, …

- Automaton-based model-checking

  Holzmann et al., Vardi et al., Kurshan et al., …

- Statistical model checking

  Clarke et al., Legay et al., Smolka et al., …

What About Model Checking?

- Temporal logic often used to formalize requirements
- Model checkers tell whether temporal-logic formulas are true or not
- Can this be adapted to Problem #1?

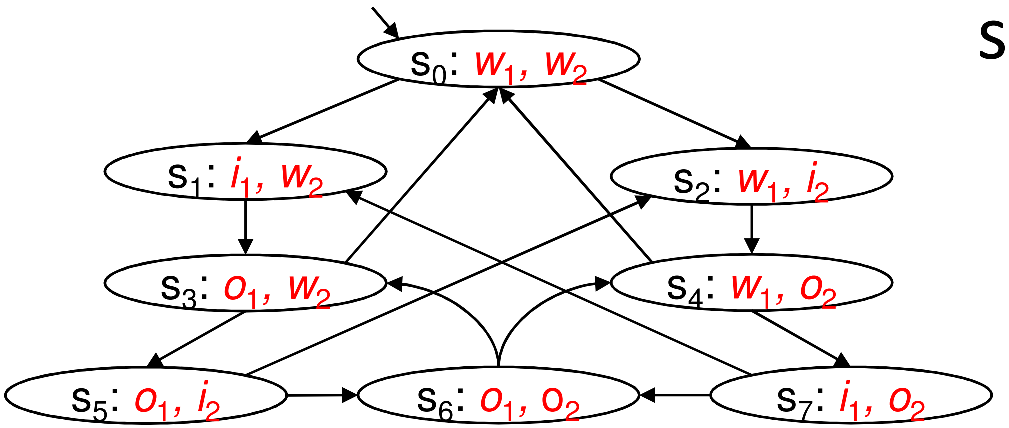
Model Checking in (1/5) Slides: Linear-Time Temporal Logic (LTL)

- Temporal Logic: modal (propositional) logics for time
  - Usual propositional operators: atomic propositions (aka propositional variables), ∧, ∨, ¬, ⇒, ...
  - Modal operator for evolution over time: U (until)
  - Derived modal operators: F (eventually), G (always)
- Examples
  - $G(\neg i_1 \vee \neg i_2)$ "At least one process is not in its critical section"
  - $G(w_1 \Rightarrow (F\ i_1))$ "If a process is waiting then it eventual is in its critical section"

Model Checking in (2/5) Slides: LTL Semantics

- LTL formulas interpreted with respect to executions of *Kripke structures*

Kripke structure



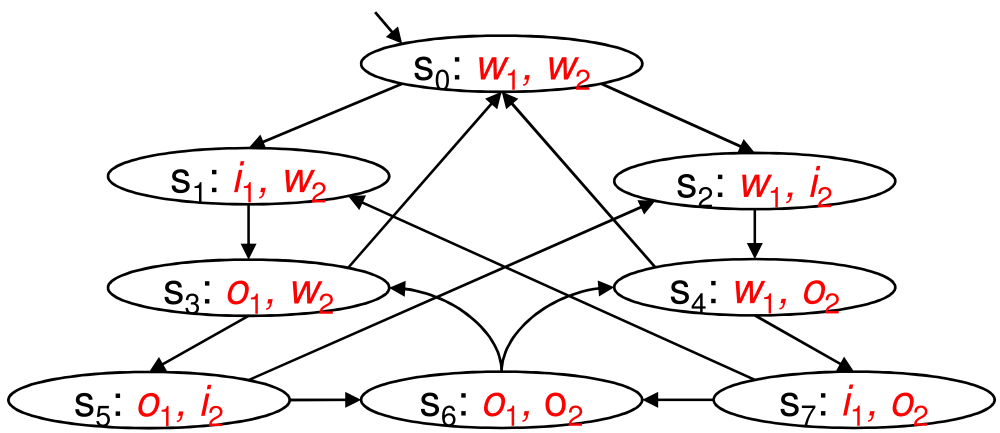
Execution

$s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_0 \rightarrow s_1 \ldots$

- Examples
  - $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_0 \rightarrow s_1 \ldots \models G\ (\neg i_1 \lor \neg i_2)$
  - $s_0 \rightarrow s_1 \rightarrow s_3 \rightarrow s_0 \rightarrow s_1 \ldots \not\models G\ (w_2 \Rightarrow (F\ i_2))$

Model Checking in (3/5) Slides: CTL* / CTL

- CTL* / CTL support *branching time*
  - Path quantifiers A ("all paths"), E ("some path") mixed in with LTL
  - (State) formulas interpreted with respect to states
  - AG $(\neg i_1 \vee \neg i_2)$: "For all paths, it is always the case that i_1 or i_2 is false"



- Examples
  - $s_0 \models AG\,(\neg i_1 \vee \neg i_2)$
  - $s_0 \not\models AG\,(w_2 \Rightarrow (F\, i_2))$

Model Checking in (4/5) Slides: LTL / CTL Model Checking

- LTL
  - System (Kripke Structure) satisfies (LTL) formula ϕ iff every execution satisfies ϕ
  - Typical model-checking approach: construct (Büchi) automaton accepting all (infinite) sequences satisfying ϕ
  - Sample tool: SPIN
- CTL
  - Subset of CTL* that requires a path quantifier (A/E) in front of every modality (F/G/U)
  - System satisfies CTL formula ϕ iff start state of Kripke structure does
  - Typical model-checking approach: use fixpoint iteration to compute all states satisfying subformulas of ϕ, then ϕ
  - Typical tool: (nu)SMV

Model Checking in (5/5) Slides: Adding Time

- Metric Temporal Logic
  - Add time bounds to modalities
  - E.g. $F_{[0,5]}\,a$: "Between 0 and 5 time units from now, *a* will hold"
- Timed CTL
  - Add path quantifiers to MTL
  - E.g. $AF_{[0,5]}\,\phi$: "Along all paths, it is the case that between 0 and 5 times units from now, *a* will hold"

So, What About Temporal Logic?

Can it be adapted to Problem #1?

Of Course It Can

- "Whenever the brake pedal is pressed, the cruise control shall become inactive."

  $AG\ (brake \Rightarrow \neg active)$

- "Whenever actual, desired speeds differ by more than 1 km/h, the cruise control shall fix within 3 seconds."

  $AG(|speed - dSpeed| > 1 \Rightarrow AF_{\leq 3}|speed - dSpeed| \leq 1)$

Common Criticisms of Temporal Logic

- Formulas hard to comprehend for non-specialists

  Compare:

  – AG (|speed–dSpeed| > 1 ⇒ $AF_{\leq 3}$ |speed–dSpeed| ≤ 1)

  –
  $$H(s) = P\frac{Ds^2 + s + I}{s + C}$$
  $$\mathrm{Output(t)} = P_{\mathrm{contrib}} + I_{\mathrm{contrib}} + D_{\mathrm{contrib}}$$
  $$P_{\mathrm{contrib}} = K_p\, e(t)$$
  $$I_{\mathrm{contrib}} = K_i \int_0^t e(\tau)\, d\tau$$
  $$D_{\mathrm{contrib}} = K_d \frac{de}{dt}$$

- Complex formulas hard to develop, understand

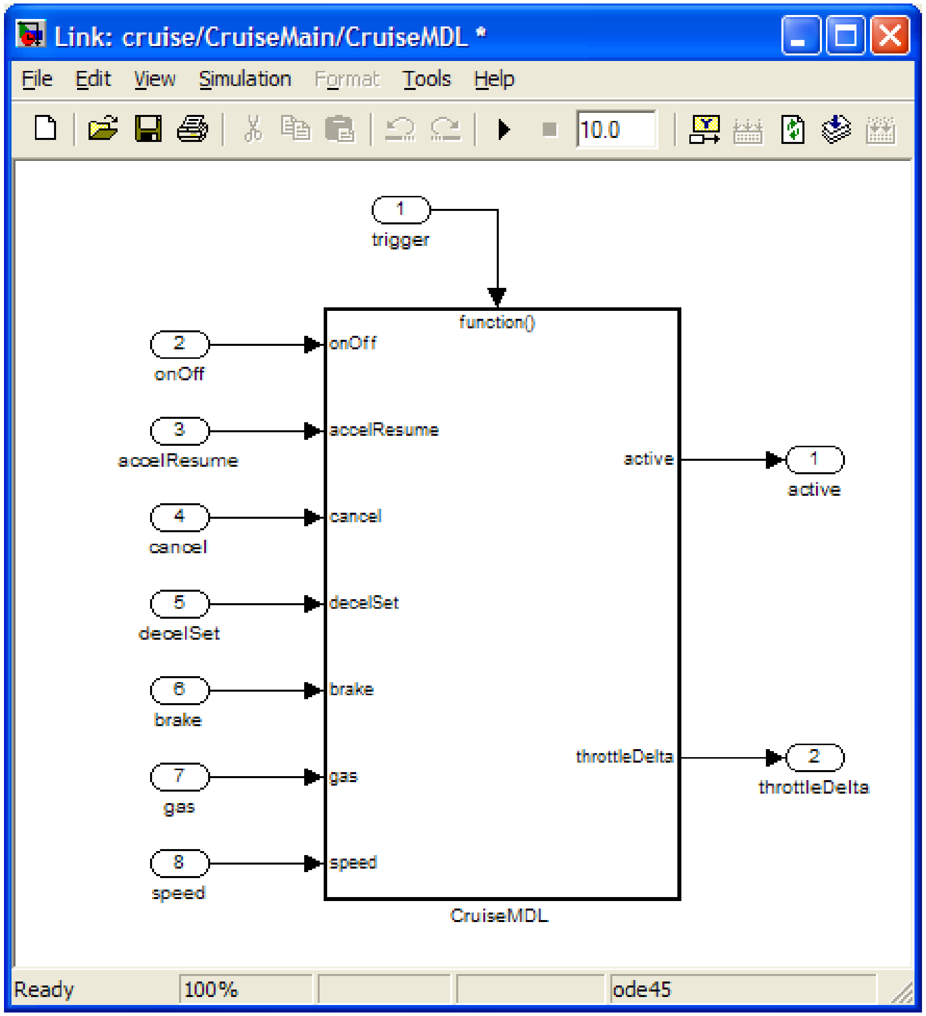
  An argument for simpler requirements?

Better Criticisms

- A second notation
- Specification debugging
- Scope issues

  AG (|speed – dSpeed| > 1 ⇒ $AF_{\leq 3}$ |speed – dSpeed| ≤ 1)

  "dSpeed"?
  - Not an input
  - Not an output
  - Internal variable!
- PIYC / TIYC?



Verification via Model Checking

- Yes
  - Full proofs of correctness (in principle)
  - Automatic!
- No
  - Combinatorial complexity

    State-explosion: number of states grows exponentially in number of bits
  - When will it work?

What about Temporal Logic and PIYC / TIYC?

- Relating testing to branching time, infinite executions not so obvious
- Run-time monitoring
  - Needs updating of TL semantics (finite sequences)
  - Need to relate specification-level concepts (“active”) to system level
  - Usual focus has been on code, requirements
- Statistical model checking
  - Another form of approximate verification
  - Need probabilistic assumptions about different transitions

IBV Intended to Address These Criticisms

- One notation; existing tools can support requirements formalization, debugging
- Scope issues addressed implicitly
- Instrumentation is executable, hence debuggable
- Testing currently scales better than proof
  … but proof still possible with right tools

Automotive Pilot Study #1

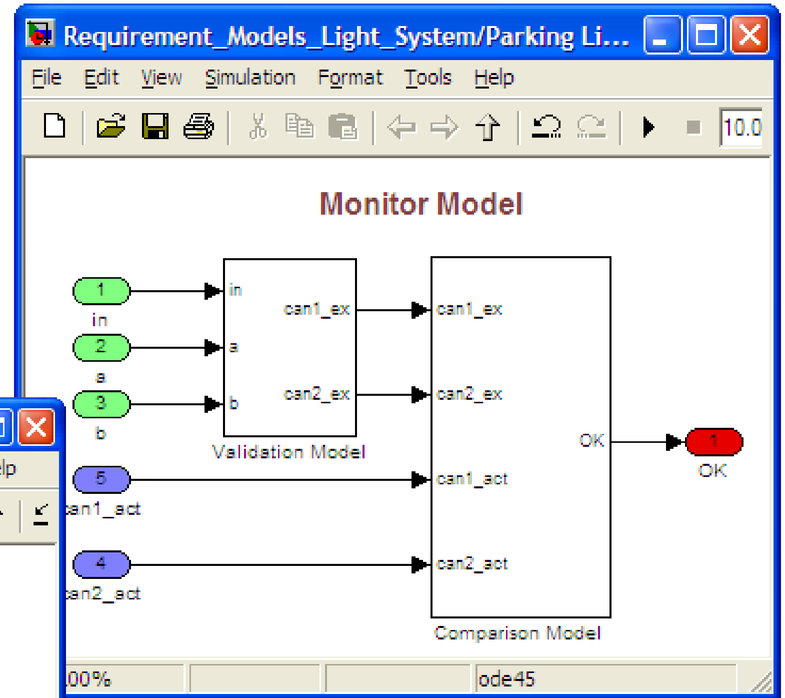
- Emergency Blinking Function (EBF)
  - Part of body computer module
  - Artifacts
    - Requirements document (300+ pages)
    - Code (200+ KLOC)
- Question: Will IBV work?

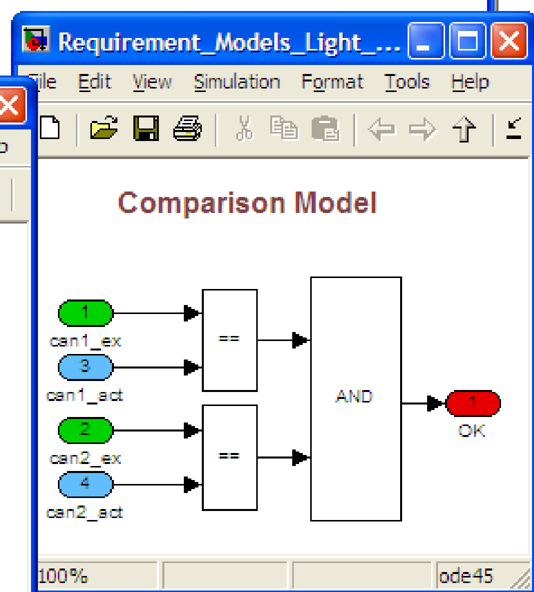
Pilot Study #1 (cont.)

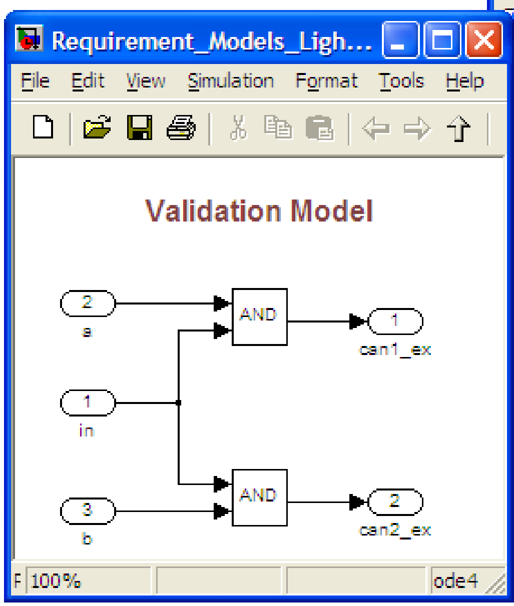
- Tasks
  - Code monitors from requirements
  - Code Simulink design model from C
  - Use Reactis to compare design, requirements
- Study details
  - Time frame: 3 months
  - Personnel: PhD student, Fraunhofer employee

From Requirements to Monitors

"[This] is the complete description of the control of the CAN output signals can1 and can2 produced by Function A. Function A can be activated only with in = 1. The activation takes place when either the CAN bus messages a or b is present…."







From Code to Models

- Goal: reverse-engineer model from code
  - Model-based design not used in development
  - Will IBV work for "production-strength" design?
- Part of EBF (250 SLOC) converted
  - Inports / state variables: read-before-write vars.
  - Outports: vars. written, not read
  - Resulting model: about 75 blocks

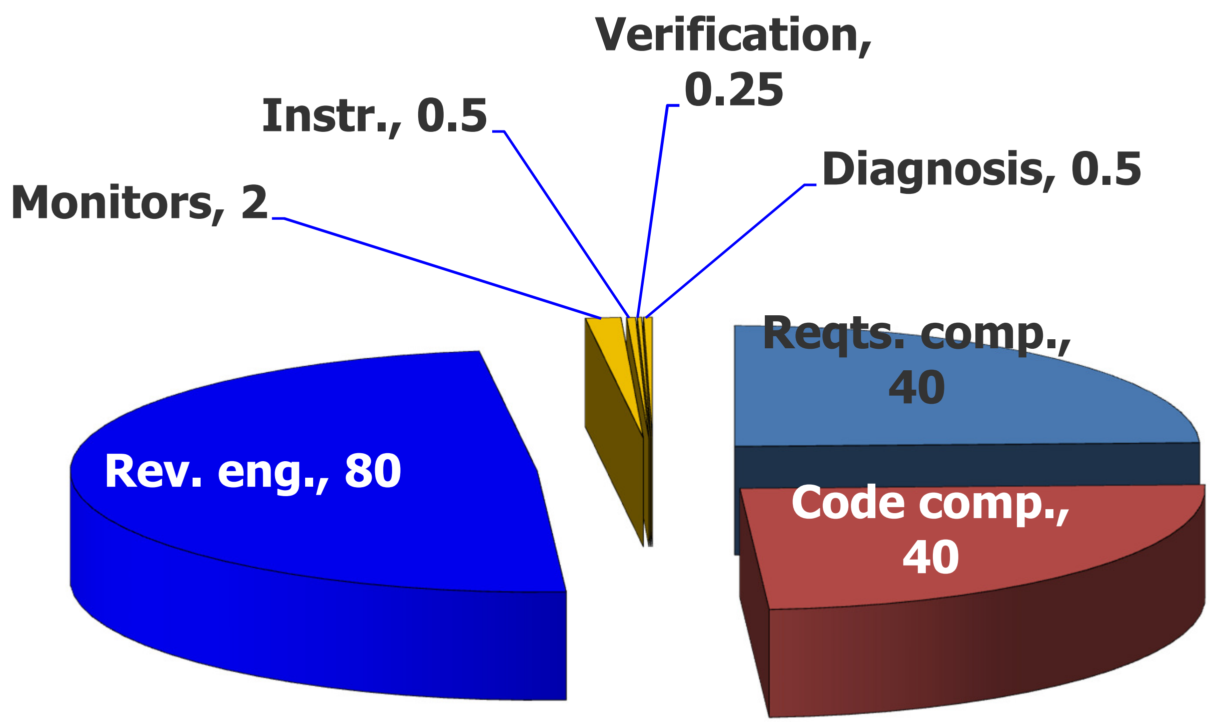
Conducting the Verification

- Reactis IBV features used
  - Instrument model with monitors
  - Generate tests automatically
- Results
  - Test suites contained 80-120 test vectors
  - Time needed: ±20 sec
  - *Omission in requirements discovered*

Requirement Issue

- Missing reset transitions in requirements
- Code was correct

Effort Data (Person-hours)



Preliminary Conclusion

- “It worked” …
- … for one feature
- … one (very complex) requirement
- … using PhDs

Automotive Pilot Study #2

- More exterior-lighting functions
- More monitor models
- No PhDs: one intern
  - B.S. in Computer Science
  - Significant expertise in Simulink
  - No automotive experience

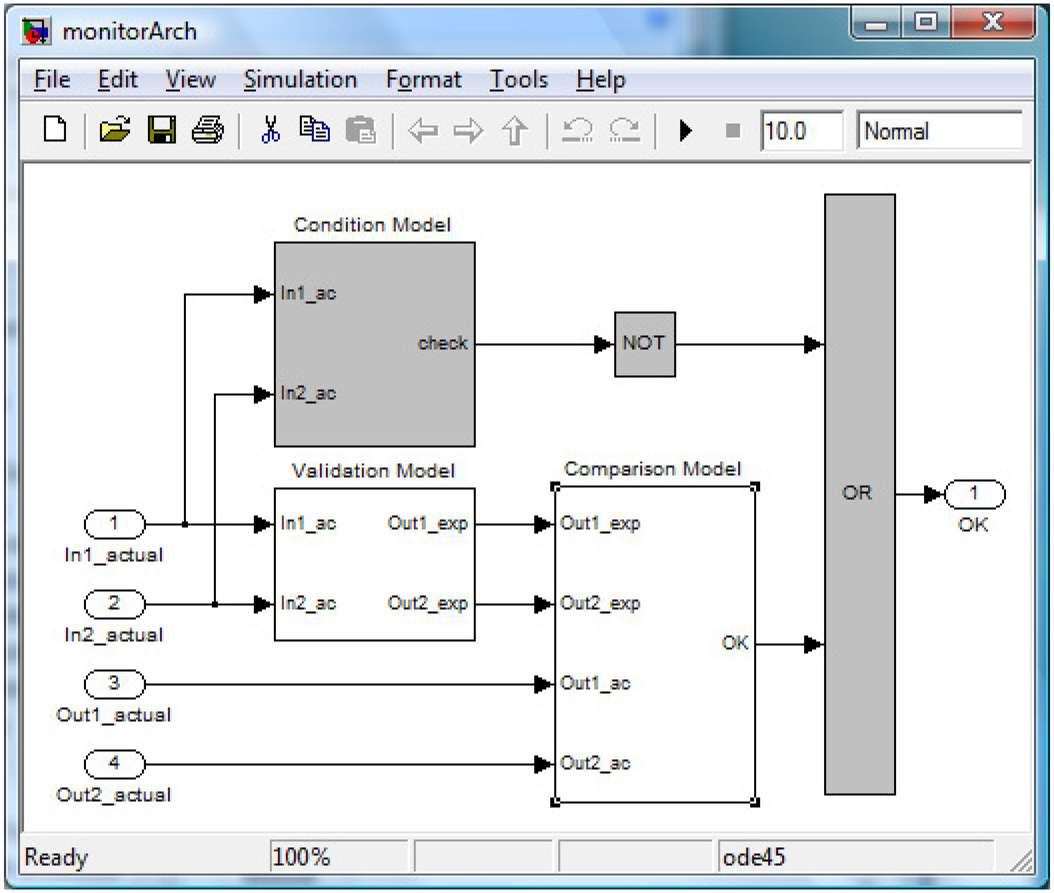
Approach

- Identify number of requirements for each exterior-lighting function
  - Count sentences
  - Read sections, beginning with fewest sentences
- Formalize requirements as monitor models
- Develop design models for functions
- Verify

Monitor Model Architecture Change

Needed for *conditional requirements*

- Behavior only specified for certain situations
- "If timeout occurs switch off light"

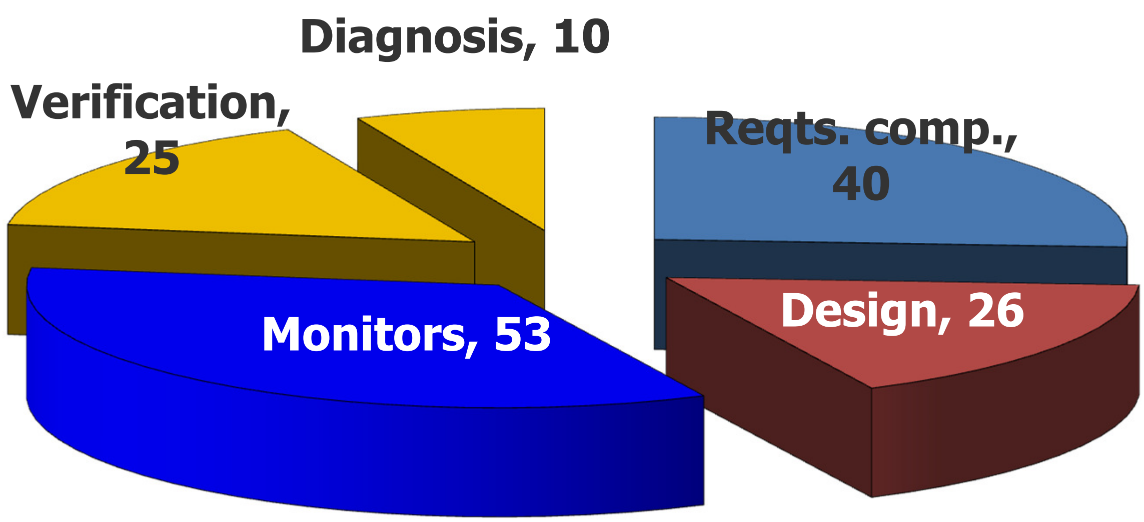


Results

- 62 monitor, 10 design models created
- Enhancements to the monitor architecture
- Verification results
  - 11 inconsistencies in requirements

    "If the ignition is off, the light must be off"
    "If the light switch is on, the light must be on"
  - Why?
    - Evolving document
    - Multiple teams
    - "The implementors will know what to do"

Effort (Person-hours)



Discussion

- Requirements modeling
  - First study: 2 hours (1.2% of total) — 1 reqt. (2 hrs. / reqt.)
  - Second study: 53 hours (34.4% of total) — 62 reqts. (50 min. / reqt.)
- Design model development
  - First study: 80 hrs. (49.0% of total) — Reverse engg. (80 hrs. / model)
  - Second study: 26 hours (16.9% of total) — Forward engg. (2.6 hrs. / model)
- Verification
  - First study: 45 min. (0.5% of total) — 1 reqt. (45 min. / reqt.)
  - Second study: 25 hours (16.2%) — 62 reqts. (25 min. / reqt.)
- Fault diagnosis
  - First study: 30 min. (0.3% of total) — 1 reqt., 1 error (30 min. / error)
  - Second study: 10 hours (6.5% of total) — 62 reqts., 11 errors (55 min. / error)

More Discussion

- Was the requirements document was modified?
  - No
  - Reasons:
    - Document developed with customer, requires customer sign-off to change
    - Developers know domain better
- Requirements
  - Not always the "gold standard" for system behavior
  - Rather: one description of the system that should ideally be consistent with other descriptions

Yet More Discussion

- When did we "prove if we could"?
  – We didn't …
  – … because of lack of available tool support
- Did we debug monitor-models while developing them?

  Yes!

Summary

- PIYC / TIYC approaches identified, applied for model-based development
  - Model-based testing: equivalence checking
  - Instrumentation-based verification (IBV): requirements checking
- Idea of PIYC / TIYC: gain benefit from formalism even if formal verification infeasible
  - Model-based testing: models serve as source of tests, oracles
  - Instrumentation-based verification (IBV): monitors act as oracles
- Requirements are not always what is required

  Requirements documents are often "just another description"

Ongoing / Future Work

- PIYC / TIYC
  - How to characterize the “gap” between proof, testing?
  - How to combine formal, approximate verification?
- IBV model checking for Simulink / Stateflow
- IBV for code
- System comprehension via testing, machine-learning

Thanks!

Rance Cleaveland

rance@cs.umd.edu

