Formal Methods Meets Testing

Rance Cleaveland

Department of Computer Science

NSF CPS Frontier Project CNS-1446365

CyberCardia

UMIACS
University of Maryland Institute for Advanced Computer Studies

UNIVERSITY OF MARYLAND

The Institute for Systems Research

This Talk: PIYC / TIYC

- "Pick-tick"
  - Prove If You Can.
  - Test If You Cannot.
- More precisely
  - Formal specifications should support verification
  - They should also support testing
  - Testing should be seen as "approximate formal verification"

Formal Methods

- Mathematically rigorous approaches to specifying, verifying systems
  - Originally: software, hardware design
  - Key people: Clarke, Dijkstra, Hoare, Lamport, Milner, Pnueli, …
- 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 (e.g. state machines)

  Systems must be mathematical objects!

- Formal specifications (e.g. temporal logic)

  Mathematical descriptions of desired behavior

- Formal verification = proof
  - Model checking: Proofs done automatically
  - Theorem proving: Proofs done “automatedly”

Status of Formal Methods

- Noteworthy successes!
  - sel4 OS kernel verification
  - Railway signaling
  - Paris Roissy VAL shuttle
  - Mars Rover
  - Satellite control
  - …
- 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

Testing

- How verification and validation happens in practice
  - Incomplete, but
  - Scalable, and
  - Mandated (regulation)
- Terminology
  - Black box / white box
  - Hardware-in-the-loop
  - Model-based testing (MBT)
  - …

My Perspective

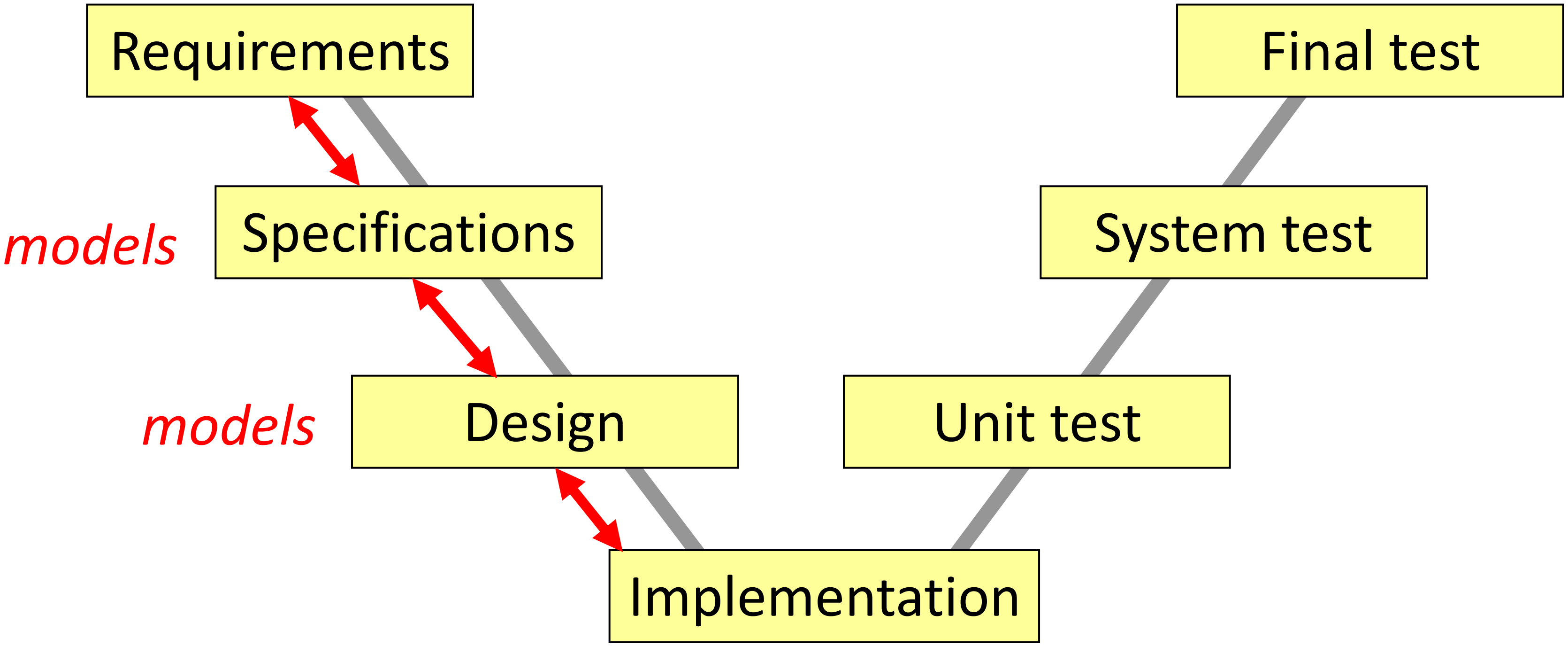
- Proving is hard, but guarantees are very strong
- If proofs are not possible
  – Must test to conduct V&V
  – Benefits of formal specifications in this case?
- “Prove If You Can, Test If You Cannot” (PIYC/TIYC)

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

Rest of Talk: PIYC / TIYC in Practice

- Model-based testing
  - Models used as software specifications
  - MBT used to check software *vis à vis* specs
- Instrumentation-Based Verification (IBV)
  - Specifications given in same notation as software
  - Verification = instrument software, check for errors
- Context
  - Automotive control software and Model-Based Development (MBD)
  - MATLAB® / Simulink® / Stateflow® / Reactis®

Model-Based Development (MBD) in Automotive



Main Motivation: *autocode*

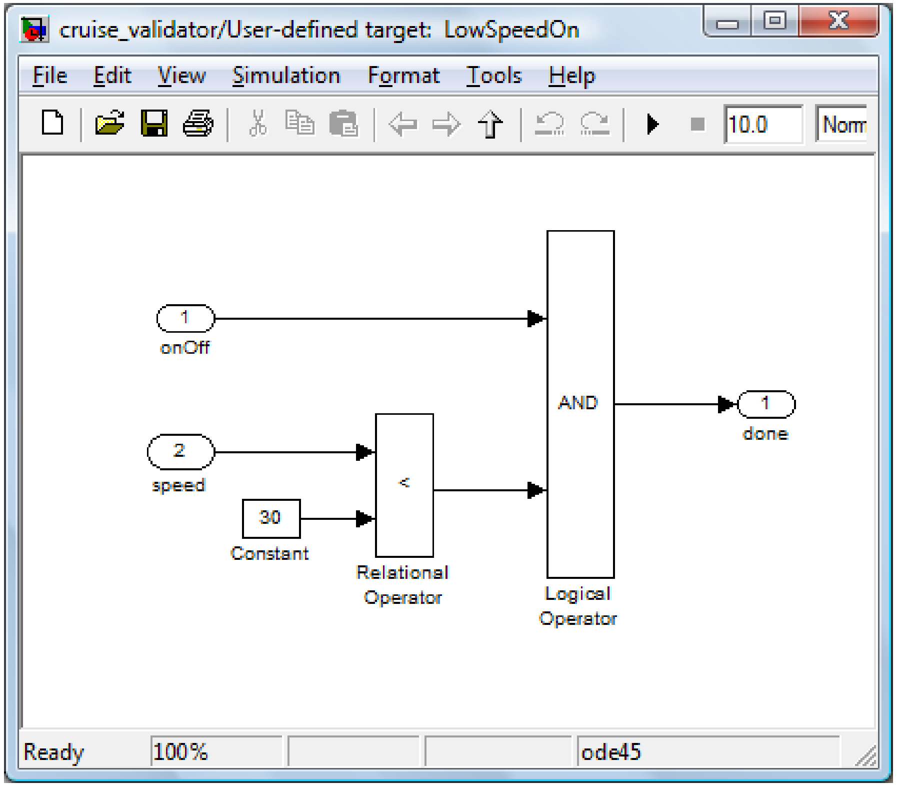
Models also support *V&V, testing*

PIYC / TIYC for MBD

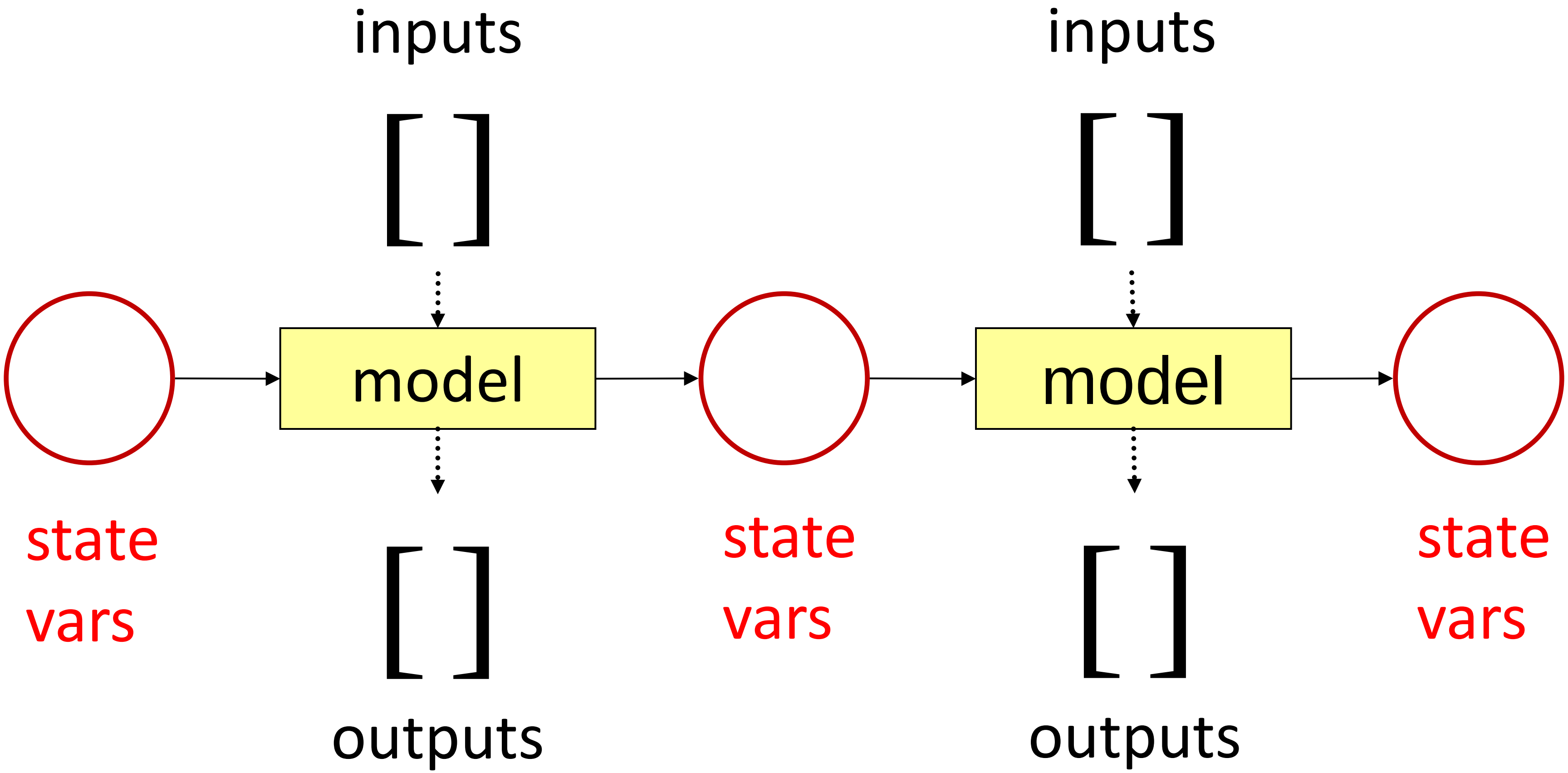
- Formalize verification problems mathematically
- Give testing-based *approximate* verification strategies based on formalizations

Simulink

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



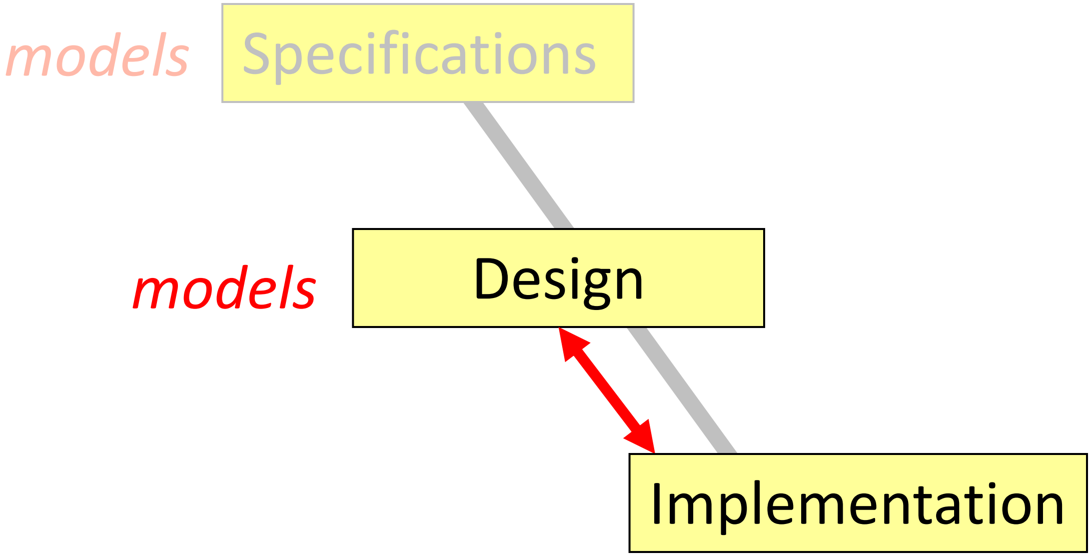
Discrete Simulink Semantics



Discrete Simulink Semantics (cont.)

- Simulink models are *Mealy machines*
  - States are assignments 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 sequence of transition labels of execution of *M* }

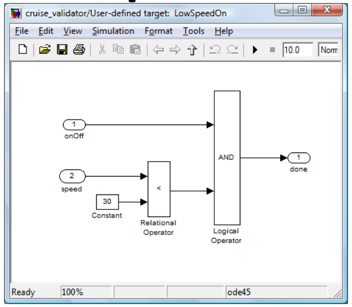
MBD Verification Problem #1

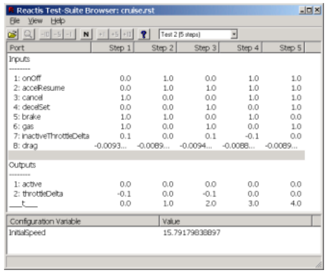


Does implementation meet design?

Model-Based Testing

- An emerging approach to this problem
  - From Simulink model …
  - … generate test cases
  - … and run them on system
  - … comparing results
- Model serves as
  - Specification
  - Test oracle

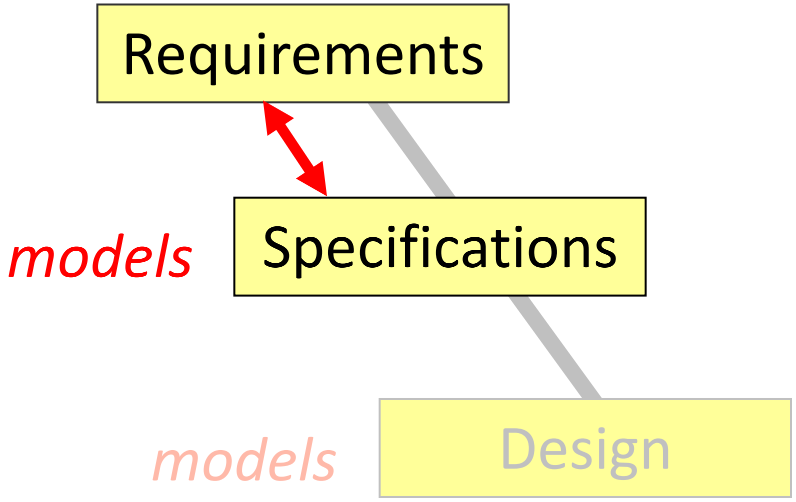




Improving MBT

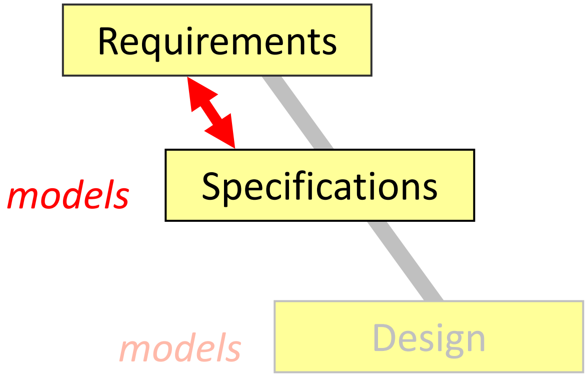
- Recall formalization of Problem #1
  - Given *M, S*
  - Prove L(*M*) = L(*S*)
  - Classical MBT: generate tests from *M,* run on *S*
- ***If L(M) = L(S) is goal, why not also generate tests from S, run them on M?***
- Result: "back-to-back testing"
  - Reactis used to generate tests from M
  - Reactis for C used to generate tests from C code
  - Controversial!
    - "You can't generate tests from implementations"
    - But formalization suggests this is perfectly reasonable!

MBD Verification Problem #2



Do specs satisfy requirements?

Formalizing Verification Problem #2

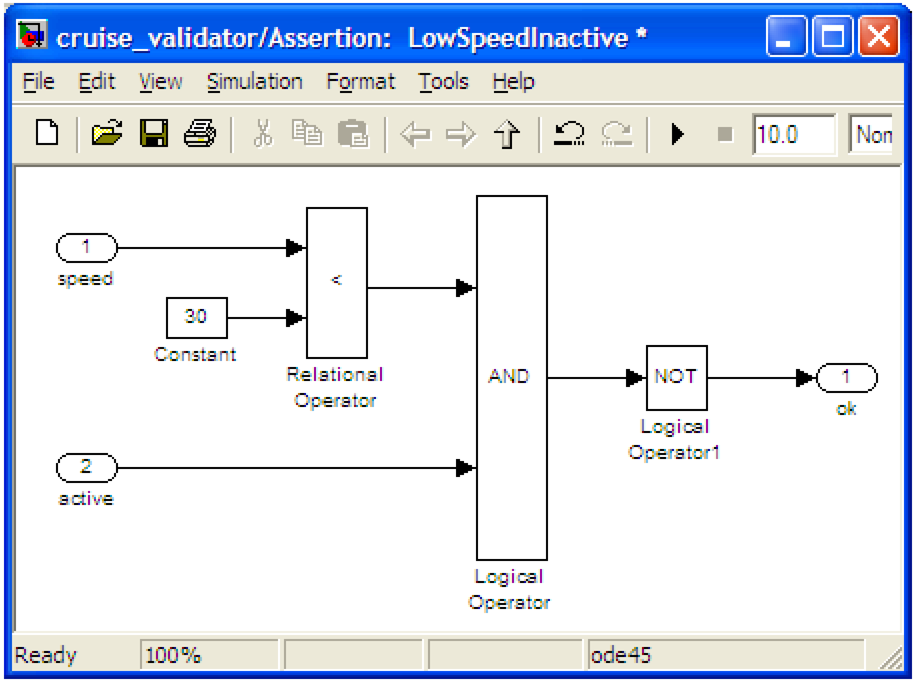


- Need following for PIYC / TIYC:
  - Formalized requirements
  - Formalized notion of satisfaction
- Our approach: *Instrumentation-Based Verification*

IBV: Requirements

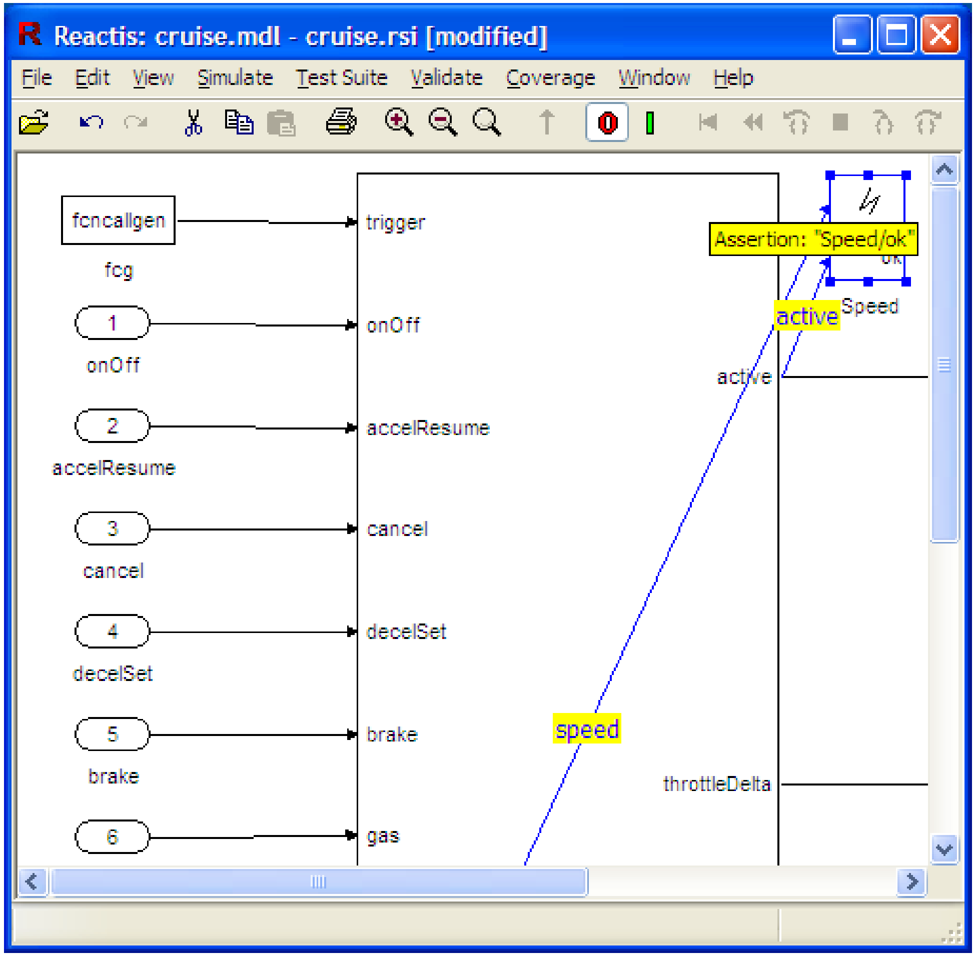
- Formalize requirements as *monitor models*
- Example

  If speed is < 30, cruise control must remain inactive



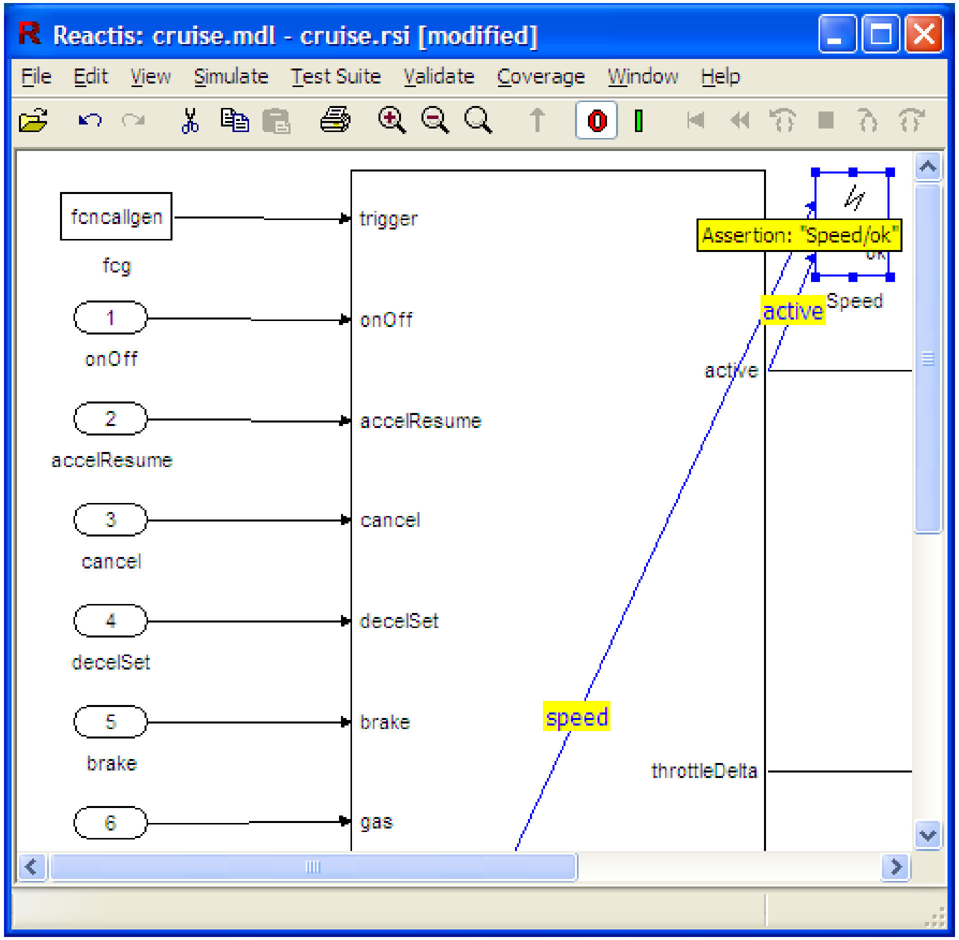
IBV: Satisfaction

- 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 #2

- 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



Summary

- PIYC / TIYC!
  - Formalize specs
  - View testing as "approximate" formal verification
- Applications in model-based testing, verification against requirements

Provocative Statement!

- Down with Temporal Logic!
- Really?
  - – Of course not! Great vehicle for research
  - – But PIYC/TIYC? Not so much …

Specification Reconstruction

- V&V needs requirements specifications
  - Requirements then checked using testing, formal methods, etc.
  - Quality of V&V depends on quality of specification
- Problem!
  - Specification must be maintained, updated, checked
  - Implicit requirements often not documented
  - “Emergent behavior?”
- *Specification reconstruction*
  - Given system (model) …
  - … automatically propose requirements

White-Box Invariant Reconstruction

- Invariants: one type of requirements
  - Invariant stipulates relationship that should be preserved among state variables as system evolves
  - E.g.

    If the brake pedal is pressed, the cruise control must immediately disengage
- Invariant reconstruction via models, data mining, IBV
  - Generate test cases
  - Compute proposed invariants using data mining
  - Check proposed invariants using IBV
  - Repeat

(Joint work with Christoph Schulze)

Data Mining

- Tools for inferring patterns in (time-series) data
  - Input: table

Time	*x*	*y*
0	1	0
1	-1	-1
2	2	1
…	…	…

  - Output: patterns (= formulas)

    e.g. $-1 \leq x \leq 2$

    $0 \leq x \leq 3 \;\rightarrow\; y \geq 0$

Association Rules

- An important class of patterns!
  - Form: $\wedge \phi_i \rightarrow \wedge \gamma_j$
    - ϕ_i, γ_j are propositions involving variables, constants
    - $\{\phi_i\}$, $\{\gamma_j\}$ are disjoint
    - Traditionally: $j = 1$
  - E.g. $x = 1 \rightarrow y = 0$
- Our work: find invariants in form of association rules
  - LHS: propositions involving inputs, “incoming state”
  - RHS: proposition involving outputs, “outgoing state”

Apriori Algorithm

- Widely used association-rule mining technique
- Developed in 1993-94 by Agrawal et al.
  – SIGMOD 1993
  – VLDB 1994
- Implemented in many data-mining tools (Weka, Magnum Opus, …)

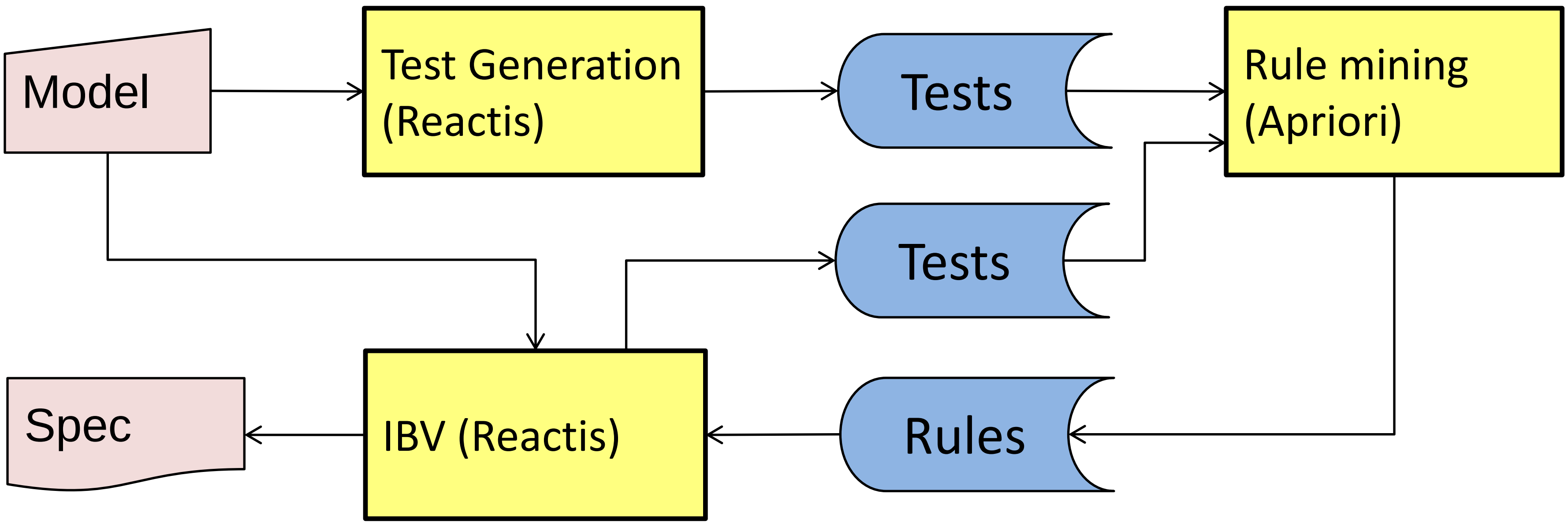
Invariant Reconstruction

- General idea
  - Treat test results (I/O sequences) as “tables”
  - Invariants: association rules with coverage ≥ 1, strength = 1
  - Use Apriori to compute invariants involving inputs (antecedent), outputs (consequent)
- Additionally
  - Ensure test cases satisfy structural coverage criteria (e.g. branch coverage) to ensure “thoroughness”
  - Use IBV to double-check proposed invariants

What About IBV Tests?

- In IBV, coverage testing of instrumented model used to check for monitor violations
- Tests inducing violations can be used to remove invariants subsequent “minings”
- They also can be a source of other invariants

Our Approach in Detail



- Reactis creates tests to do IBV check
- These tests are “cycled back through” the data-mining tool, together with original tests

Pilot Study: Production Body-Electronic Application

- Artifacts
  - Simulink model (ca. 75 blocks)
  - Requirements spec formulated as state machine
  - Requirements correspond to 42 invariants defining transition relation, e.g.

    $state = 1 \wedge pressed = true \rightarrow new\_state = 2$
- Goal: Compare our approach, random testing
  - Completeness (% of 42 detected?)
  - Accuracy (% false positives?)

Experimental Results

- Hypothesis: coverage-testing yields better invariants than random testing
- Coverage results (one iteration of test generation)

  95% of inferred invariants true

  97% of requirements inferred

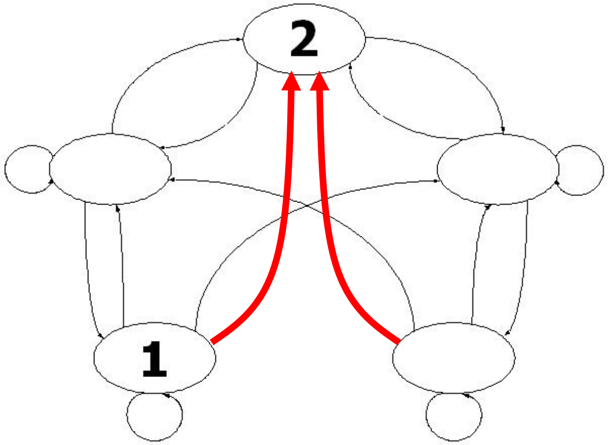
  Two missing requirements detected
- Random results:

  55% of inferred invariants true

  40% of requirements inferred
- Hypothesis confirmed (for this case study)

Requirement Issue

- Missing reset transitions in requirements
- Code was correct



Procedural Issues

- How do you trust generated invariants in absence of "requirements baseline"?
- Our approach: *Jaccard similarity*

Jaccard Similarity Measures

- Jaccard: a tool for measuring set similarity
- Let A, B be sets. Then the Jaccard similarity measure, J(A,B), of A and B is:

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|}$$

- Facts
  - $0 \leq J(A,B) \leq 1$
  - Closer to 1 means "more similar"

Jaccard and Invariant Generation

- High Jaccard similarity means more "stable"
  - For coverage: average Jaccard score is 0.87
  - For random: average is 0.65
- Another use: iteration termination
  - Our approach allows iteration of "test / generate / check"
  - When to terminate: use Jaccard! (i.e. terminate when successive invariant sets are "similar enough")

Provocative Statement!

Focus at UMD in CyberCardia

- Foundations, tools for reasoning about CPS
  - Formal modeling of CPS
  - Formal specification, verification
- This year: Specification reconstruction
  - Given model M, infer temporal properties that M (likely) satisfies
  - Motivations
    - Model understanding
    - Specification updating
    - Means for "jump-starting" formal specificiations in often unfamiliar notations
- See poster (48-50)!

Specific Results in 2017

- Linear temporal-logic query checking
  - Problem
    - Given Kripke structure M, LTL “template” phi[x]
    - Find most general solution phi’ for missing formula x so that M satisfies phi[x:=phi’]
  - Algorithmic solution based on model checking developed, implemented, evaluated
  - Work presented at AVoCS/FMICS 2017
- Invariant mining from test data
  - Problem
    - Given (Simulink) model M, state variables of interest
    - Propose invariants describing relationships among variables
  - Approach: use data-mining on test data coupled with retesting to generate likely invariants
  - Evaluation used 11 models from automotive, medical-device domain
  - Work presented at EMSOFT 2017