Software Verification / Testing

Rance Cleaveland

Department of Computer Science
and
Fraunhofer Center for Experimental Software Engineering

University of Maryland

20 October 2011

This Talk

Some recent developments in software verification and testing

Software Verification?

- Related to, but different from, IEEE definition
- Traditionally, in CS: *formal methods*
  - Given software, spec
    - Software = “code”
    - Spec = “requirement” = logical formula
  - Prove software meets spec
- (Informal verification often called “validation”.)

Model Checking

- Verification = proof
- Model checking: automated proof!
  - Given software, spec
  - Model checker tries to build proof
- Ongoing research: applicability
  - Decidability
  - Scalability
- Embedded control applications!

Software Testing

- Most often-used method for checking software correctness
  - Select tests
  - Run software on tests
  - Analyze results
- Traditionally
  - Manual, hence time-consuming, expensive
  - In control applications: hard to test software by itself

UNIVERSITY OF MARYLAND

CMACS
Computational Modeling and Analysis for Complex Systems

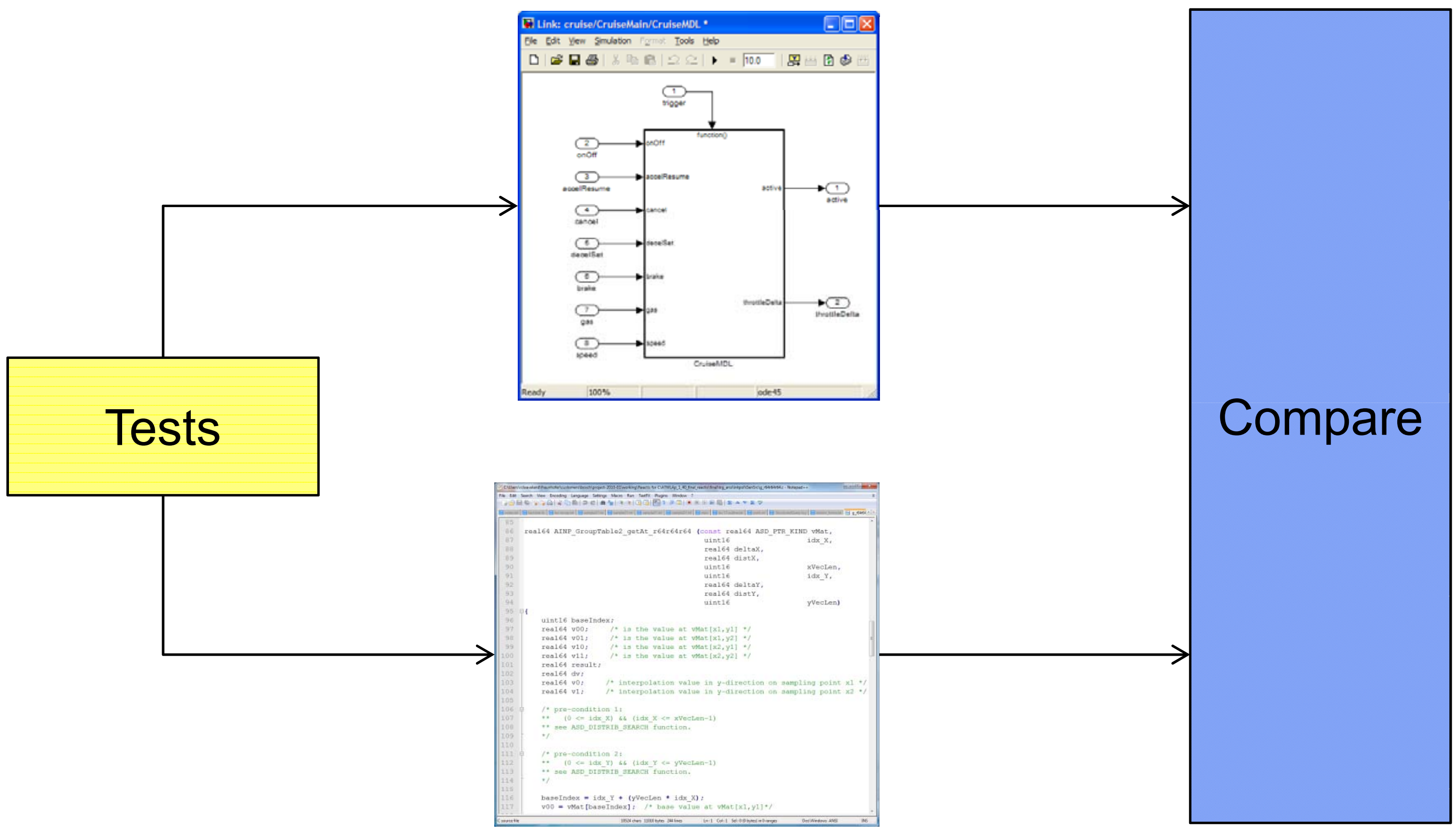
Exciting Developments

- Combine
  - Formal specs
  - Testing
- To automate testing "scalably"
  - Model-based testing
  - Instrumentation-based verification
  - Requirements reconstruction

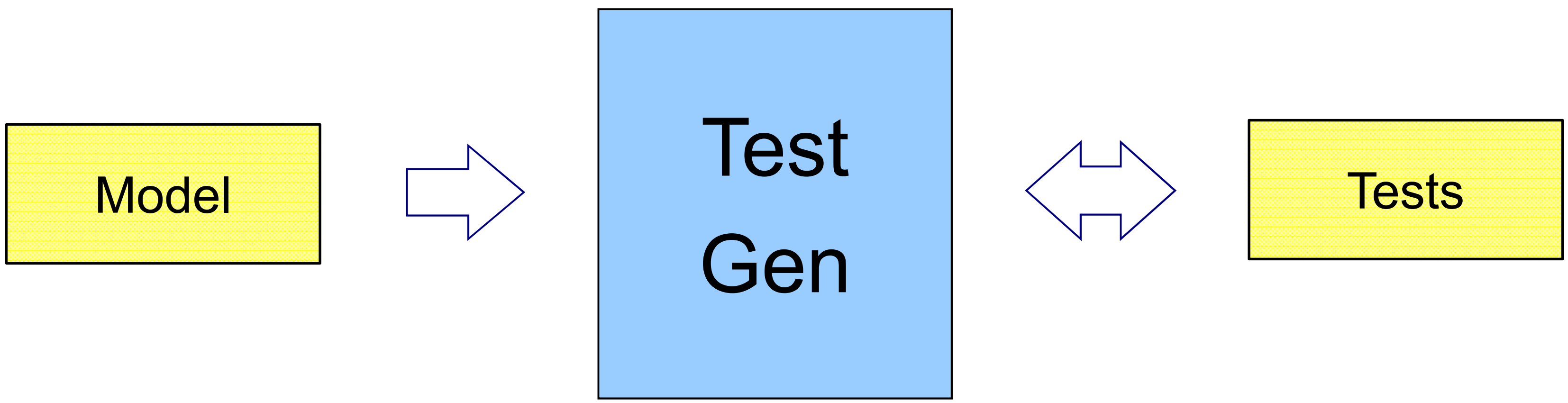
Model-Based Testing

- Develop specs as executable models
  - Simulink
  - State machines
  - Etc.
- Use model to determine correct test response
  - Automates "results analysis"
  - Models, tests needed

Model-Based Testing (cont.)



Tests Can Be Generated from Models!



- Functionality provided by tools like Reactis® for Simulink / Stateflow
- Goal: automate test generation task by creating tests that cover model logic
- Reactis: guided simulation algorithm

Applying Model-Based Testing

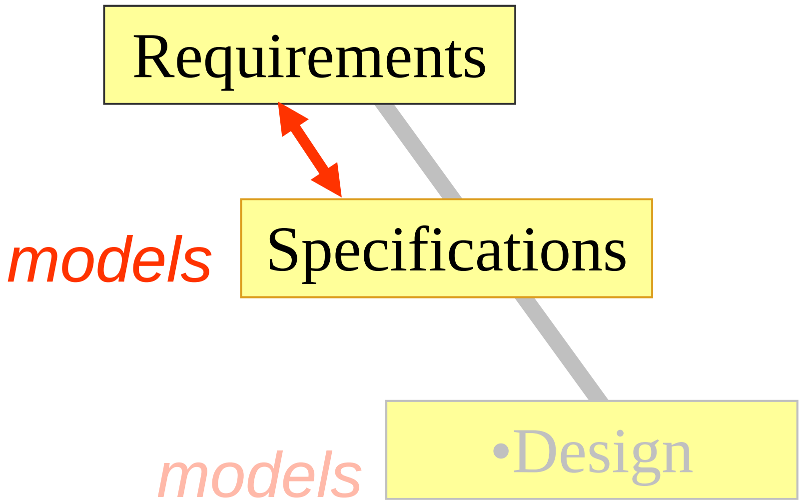
- Widespread in automotive, less so in aero / medical-device
  - Regulatory issues
  - Need for models
  - Modeling notations, support
- What about models?
  - Sometimes result of earlier design phases
  - Models as reusable testing infrastructure

Challenges

- Technical
  - Algorithms for test generation
  - Modeling languages
- Procedural
  - Integration into existing QA processes
  - Regulatory considerations

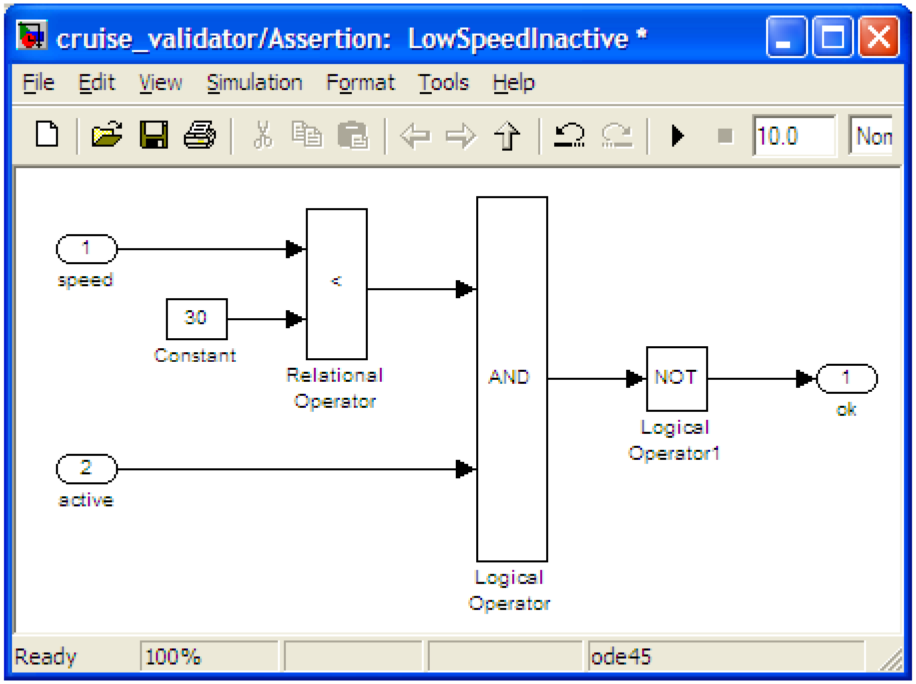
Instrumentation-Based Verification

- Model-based testing assumes model correct
- IBV: a way to check model correctness vis a vis requirements



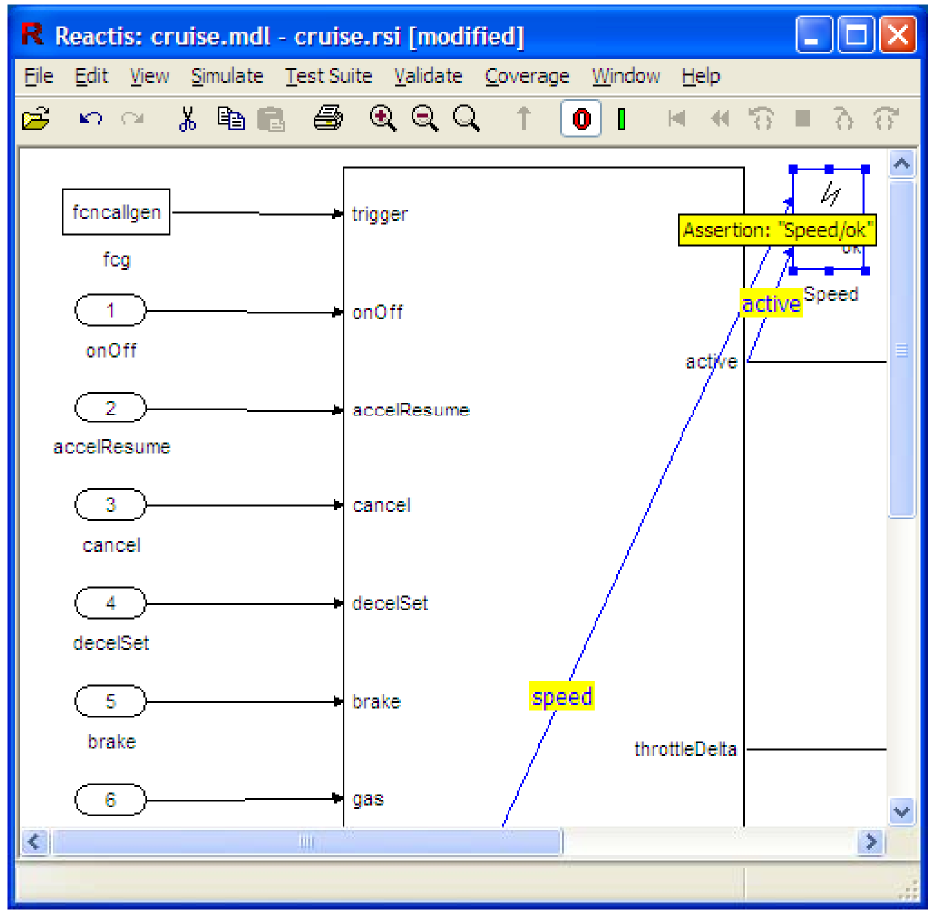
Instrumentation-Based Verification: Requirements

- Verification needs formalized requirements
- IBV: formalize requirements as *monitor models*
- Example
  "If speed is < 30, cruise control must remain inactive"



Instrumentation-Based Verification: Checking Requirements

- Instrument design model with monitors
- Use *coverage testing* to check for monitor violations
- Tool: Reactis®
  - Product of Reactive Systems, Inc.
  - Separates instrumentation, design
  - More info: www.reactive-systems.com



Applying Instrumentation-Based Testing

- Robert Bosch production automotive application
  - Requirements: 300-page document
  - 10 subsystems formalized (20% of system)
    - 62 requirements formalized as monitor models
    - IBV applied
    - 11 requirements issues identified
- Another Bosch case study: product-line verification using IBV
- A number of other case studies

Requirements Reconstruction

- The Requirements Reconstruction problem
  - Given: software
  - Produce: requirements
- Why?
  - System comprehension
  - Specification reconstruction
    - Missing / incomplete / out-of-date documentation
    - “Implicit requirements” (introduced by developers)

Invariants as Requirements

- Some requirements given as invariants
  - *"When the brake pedal is depressed, the cruise control must disengage"*
- State machines can be viewed as invariants
  - States: values of variables
  - Transitions: invariants
  - *"If the current state is A then the next state can be B"*
- Another project with Robert Bosch

Invariant Reconstruction

- Generate test data satisfying *coverage criteria*
- Use *machine learning* to propose invariants
- Check invariants using *instrumentation-based verification*

Machine Learning: *Association Rule Mining*

- Tools for inferring relationships among variables based on time-series data
  - Input: table

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

  - Output: relationships ("association rules")
    e.g. $0 \le x \le 3$ -> $y \ge 0$

Association Rules and Invariant Reconstruction

- General dea
  - Treat tests (I/O sequences) as data
  - Use machine learning to infer relationships between inputs, outputs
- Our insight
  - Ensure test cases satisfy coverage criteria to ensure “thoroughness”
  - Use IBV to double-check proposed relationships

Pilot Study: Production Automotive Application

- Artifacts
  - Simulink model (ca. 75 blocks)
  - Requirements formulated as state machine
  - Requirements correspond to 42 invariants defining transition relation
- Goal: Compare our approach, random testing [Raz]
  - Completeness (% of 42 detected?)
  - Accuracy (% false positives?)

Experimental Results

- Hypothesis: coverage-testing yields better invariants than random testing
- Coverage results:

  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

Summary

- Intersection of formal methods, testing can yield practical verification approaches
  – Model-based testing
  – Instrumentation-based verification
- Automated test generation can be used to infer invariants