Invisible Formal Methods:
Generating Efficient Test Sets
With a Model Checker

John Rushby
with Grégoire Hamon and Leonardo de Moura

Computer Science Laboratory
SRI International
Menlo Park, California, USA
Full Formal Verification is a Hard Sell: The Wall

Reward (assurance)

Effort

PVS

interactive theorem proving

John Rushby, SRI

Invisible FM and AutoTestGen: 2
Newer Technologies Improve the Value Proposition

Reward (assurance)

Effort

SAL

model

checking

automated

theorem proving

and abstraction

ICS

PVS

interactive

theorem proving

But only by a little

John Rushby, SRI
The Unserved Area Is An Interesting Opportunity

Reward (assurance)

Effort

Conjecture: reward/effort climbs steeply in the invisible region

John Rushby, SRI

Invisible FM and AutoTestGen: 4
Invisible Formal Methods

• **Use the technology of formal methods**
  - Theorem proving, constraint satisfaction, model checking, abstraction, symbolic evaluation

• **To augment traditional methods and tools**
  - Compilers, debuggers

• **Or to automate traditional processes**
  - Testing, reviews, debugging

• **To do this, we must unobtrusively (i.e., invisibly) extract**
  - A formal specification
  - A collection of properties

• **And deliver a useful result in a familiar form**
Invisible Formal System Specifications

- Traditionally, there was nothing formal (i.e., mechanically analyzable) prior to the executable program
  - Requirements, specifications, etc. were just natural language words, and pictures
- So one response is to apply formal methods to programs
  - E.g., extended static analysis
- But for embedded systems, industry has adopted model based design (MBD) at a surprisingly rapid pace
  - Matlab (Simulink/Stateflow): over 500,000 licenses
  - Statecharts
  - Scade/Esterel
- Some of these (e.g., Stateflow) have less-than-ideal semantics, but it’s possible to cope with them
  - E.g., our paper in FASE ’04
Invisible Property Specifications

- MBD provides formal specifications of the system
- But what properties shall we apply formal analysis to?
- One approach is to analyze structural properties
  - E.g., no reliance on 12 o’clock rule in Stateflow
  - Similar to table checking in SCR
  - Prove all conditions are pairwise disjoint
  - And collectively exhaustive
- Another is to generate structural test cases
- Either for exploration
  - E.g., “show me a sequence of inputs to get to here”
- Or for testing in support of certification and verification
Vast resources are expended on testing embedded systems.
Invisible FM Example: Generating Unit Tests

- Let’s focus initially on testing individual units of a program
- Executable model provides the oracle
- Various criteria for test generation
  
  **Functional tests**: tests are derived by considering intended function or desired properties of the unit (requires higher-level specifications, which we do not have)
  
  **Boundary tests**: tests designed to explore inside, outside, and on the boundaries of the domains of input variables
  
  **Structural tests**: tests are designed to visit interesting paths through the specification or program (e.g., each control state, or each transition between control states)

- Let’s look at the standard method for structural test generation using model checking
Example: Stopwatch in Stateflow

Inputs: **START** and **LAP** buttons, and clock **TIC** event

Example test goals: generate input sequences to exercise **Lap_stop** to **Lap** transition, or to reach junction at bottom right

John Rushby, SRI

Invisible FM and AutoTestGen: 10
Generating Structural Tests

• Problem: find a path that satisfies a desired test goal
  ◦ E.g., reach junction at bottom right
• Symbolically execute the path, then solve the path predicate to generate concrete input sequence that satisfies all the branch conditions for the path
  ◦ If none, find another path and repeat until success or exhaustion
• Repeat for all test goals
• Solving path predicates requires constraint satisfaction over theories appearing in the model (typically, propositional calculus, arithmetic, data types)
  ◦ E.g., ICS and its competitors
  ◦ For finite cases, a SAT solver will do
• Can be improved using predicate abstraction (cf. Blast)
Generating Tests Using a Model Checker

- Method just described requires custom machinery
- **Can also be done using off-the-shelf model checkers**
  - Path search and constraint satisfaction by brute force
- Instrument model with **trap variables** that latch when a test goal is satisfied
  - E.g., a new variable `jabr` that latches `TRUE` when junction at bottom right is reached
- Model check for "always not jabr"
- **Counterexample will be desired test case**
- Trap variables add negligible overhead (’cos no interactions)
- For finite cases (e.g., numerical variables range over bounded integers) any standard model checker will do
  - Otherwise need infinite bounded model checker as in **SAL**

John Rushby, SRI

Invisible FM and AutoTestGen: 12
Tests Generated Using a Model Checker

John Rushby, SRI

Invisible FM and AutoTestGen: 13
Model Checking Pragmatics

**Explicit state**: good for complex transition relations with small statespaces

**Depth first search**: test cases generally have many irrelevant events and are too long
- E.g., 24,001 steps to reach junction at bottom right

**Breadth first search**: test cases are minimally short, but cannot cope with large statespaces
- E.g., cannot reach junction at bottom right

**Symbolic**: test cases are minimally short, but large BDD ordering overhead in big models
- E.g., reaches junction at bottom right in 125 seconds

**Bounded**: often ideal, but cannot generate tests longer than a few tens of steps, and may not be minimally short
- E.g., cannot reach junction at bottom right
Useful Optimizations

- **Backward slicing** (called *cone of influence reduction* in model checking) simplifies model relative to a property by eliminating irrelevant state variables and input events
  - Allows explicit state model checker to reach junction at bottom right in 6,001 steps in just over a second (both depth- and breadth-first)
  - And speeds up symbolic model checker

- **Prioritized traversal** is an optimization found in industrial-scale symbolic model checkers
  - Partitions the frontier in forward image computations and prioritizes according to various heuristics
  - Useful with huge state spaces when there are many targets once you get beyond a certain depth
Efficient Test Sets

• Generally we have a set of test goals (to satisfy some coverage criterion)

• Want to discharge all the goals with
  ○ Few tests (restarts have high cost)
  ○ Short total length (each step in a test has a cost)

• Independent of the method of model checking, generating a separate test for each goal produces very inefficient tests
  ○ E.g., Lap to Lap_stop test repeats Running to Lap test

• Can “winnow” them afterward

• Or check in generation for other goals discharged fortuitously
  ○ So won’t generate separate Running to Lap test if it’s already done as part of Lap to Lap_stop test
  ○ But effectiveness depends on order goals are tackled
Tests Generated Using a Model Checker (again)

Lots of redundancy in the tests generated

John Rushby, SRI

Invisible FM and AutoTestGen: 17
Generating Efficient Test Sets

- Minimal tour-based methods: difficulty is high cost to compute feasibility of paths (or size of problem when transformed, e.g., to colored tours)

- So use a greedy approach

- Instead of starting each test from the the start state, we try to extend the test found so far

- Could get stuck if we tackle the goals in a bad order

- So, simply try to reach any outstanding goal and let the model checker find a good order
  - Can slice after each goal is discharged
  - A virtuous circle: the model will get smaller as the remaining goals get harder

- Go back to the start when unable to extend current test
Less redundancy, and longer tests tend to find more bugs
Scriptable Model Checkers

- But how do we persuade a model checker to do all this?
- Several modern model checkers are scriptable
- E.g., SAL is scriptable in Scheme
- For SAL, the method described is implemented in less than 100 lines of Scheme
  - Extensions use bounded model checking
    - Parameterized incremental search depth
  - (Re)starts use either symbolic or bounded model checking
    - Parameterized choice and search depth
  - Optional slicing after each extension or each restart
  - Optional search for non-latching trap variables
- Extending tests allows a bounded model checker to reach deep states at low cost
  - 5 searches to depth 4 much easier than 1 to depth 20
(define (iterative-search module goal-list
    scan prune slice innerslice bmcinit start step stop)
  (let* ((goal (list->goal goal-list module))
    (mod (if slice (sal-module/slice-for module goal) module))
    (path (if bmcinit
              (sal-bmc/find-path-from-initial-state
                mod goal bmcinit 'ics)
              (sal-smc/find-path-from-initial-state mod goal)))))
  (if path
    (extend-search mod goal-list path scan prune
                   innerslice start step stop)
    #f)))
Core Of The SAL Test Generation Script

(define (extend-search module goal-list
    path scan prune innerslice start step stop)
    (let ((new-goal-list (if prune (goal-reduce scan goal-list path)
                                (minimal-goal-reduce scan goal-list path))))
      (cond ((null? new-goal-list) (cons '() path))
            ((> start stop) (cons new-goal-list path))
            (else
             (let* ((goal (list->goal new-goal-list module))
                    (mod (if innerslice
                                 (sal-module/slice-for module goal) module))
                    (new-path
                     (let loop ((depth start))
                      (cond ((> depth stop) '())
                            ((sal-bmc/extend-path
                              path mod goal depth 'ics))
                            (else (loop (+ depth step))))))))
             (if (pair? new-path)
                 (extend-search mod new-goal-list new-path scan
                                 prune innerslice start step stop)
                 (cons new-goal-list path)))))))
Some Experimental Results

• Generates full state and transition coverage for stopwatch with three tests in a couple of minutes
  ○ 12 steps for the statechart
  ○ 101 steps for mid right junction (actually redundant)
  ○ 6,001 steps for junction at bottom right

• Generates full state and transition coverage for shift scheduler from a 4-speed automatic transmission in two tests
  ○ Lengths 31 and 55 (total 86)
  ○ Standard method used 25 tests and 229 steps
  ○ Model has 23 states and 25 transitions
Some Experimental Results (ctd)

- Rockwell Collins has developed a series of flight guidance system (FGS) examples for NASA
- SAL translation of largest of these kindly provided by UMN
- Model has 490 variables, 246 states, 344 transitions
- Single test case of length 39 covers all but 3 transitions
  - How can that be?
  - The three outstanding goals are genuinely unreachable
- Also working on large medical device example
  - Exposes weaknesses in current Stateflow translator
    - And insertion of trap variables for MC/DC tests
Optimizations (TBD)

- **Symbolic model checking**
  - Precompute the reachable states (as a BDD)
  - Tests can then be “read off”
  - Infeasible for big systems (unless sliced)

- **Bounded model checking**
  - Precompute the $k$-fold composition of the transition relation
  - May also be able to learn hints for the SAT solver
Embellishments

• Method starts new test when current test cannot be extended

• **Would do better to try to construct an extension from some intermediate point of some previous test**

• Can search from **all** of these in parallel
  ◦ Just initialize the search to the disjunction of all states encountered in previously generated tests
  ◦ Expensive expression for bounded model checker but may have a compact BDD for symbolic model checker

• **Have the code for this but haven’t integrated it yet**

• In general, can initialize the search with **any** states you already know how to reach
  ◦ E.g., by random testing
  ◦ Or previous campaign of functional testing
Some Commercial Tools Do Similar Things

- Ketchum (aka. FormalVera and Magellan) from Synopsys
- Reactis from Reactive Systems Inc (RSI)

- Related: 0-in, DART
Test Coverage

- Need criteria to suggest when we have tested enough
- Vast literature on this topic
- Many criteria are based on structural coverage of the program
- E.g., DO178B Level A, MISRA require MC/DC coverage
  - Not allowed to generate tests from the program structure
  - But generating tests from the structure of the model is ok and likely to achieve high coverage on the program
- Plausible methodology uses structural generation from model to pick up the uncovered goals following normal testing
So Are The Test Sets Any Good?

- Heimdahl et al. found (in a limited experiment using the Rockwell FGS examples) that tests generated by traditional model checking were poor at detecting seeded errors (random testing did better).

- They conjectured this was because the tests were so short (average length about 1.25).

- We hypothesize that long tests found by our method will be more effective.
  - *In process of checking this on UMN example*

- Heimdahl also observed model checker often finds “sneaky” ways to achieve goals.

- Good coverage criteria may not be so good for generation.

- An invitation to invent new criteria for generation.
Generating Good Test Sets

- Use different (better) structural coverage criteria
- Our method is independent of criteria chosen
  - We target trap variables
  - How you set them is up to you
- Require paths to satisfy some test purpose
- Derive tests from requirements and/or domain boundaries
- Possibly combined with coverage ideas
Test Purposes

- **Constraints on the tests to be generated**—for example
  - At least 7 steps
  - Keep $x$ in $[-12..7]$ and different to $y$
  - No more than two START events in succession

- **Specify test purpose (TP) as a state machine**—for example
  - In Stateflow (engineers stay in familiar notation)
  - In system language of model checker
  - By automatic translation from property language

Raise **OK** variable while input sequence satisfies the purpose

- **Synchronously compose SUT and TP**
  - I.e., TP is a synchronous observer

- **Perform test generation as before but target conjunction of OK with trap variables**
Requirement-Driven Tests

- **Specify requirements by synchronous observers**—for example
  - In Stateflow (engineers stay in familiar notation)
  - By automatic translation from property language

- Then target structural coverage in the observer

- Or cross product of observer and SUT

- Related idea in Motorola VeriState
Boundary Value Tests

- Currently, we use the symbolic and bounded model checkers of SAL

- The infinite bounded model checker would be ideal, but it currently does not generate concrete counterexamples (because ICS does not do full model generation)

- Next versions of ICS/SAL-inf-bmc will do counterexamples, and it will be possible to choose maximum, minimum, middle values for variables subject to arithmetic constraints

- Generate tests as before, but instantiate arithmetic variables to max, min, middle values
Higher Level Tests

- Higher-level tests are more challenging

- Integration tests: similar to compliance testing, well studied in telecom area

- System tests and hardware (or simulator) in the loop tests
  - Typically want to drive system to some interesting state
  - But composition may be nondeterministic
  - And we may not have control of all components
    - E.g., hardware network may or may not drop packets

- Test generation problem becomes one of controller synthesis

- This also can be solved by the technology of model checking
  - Witness model checker of SAL is intended for this
Still Higher Level Tests

- Can have hardware devices in the loop that are not discrete systems
  - E.g., engine and gearbox with their external loads
  - More generally, the plant and its environment

- These are described by continuous variables and differential equations (in Simulink)
  - Sometimes combined with discrete elements
  - I.e., hybrid systems

- Controller synthesis for hybrid systems is very hard

- Hybrid abstraction (in Hybrid SAL) reduces hybrid systems to discrete conservative approximations

- Can then do controller synthesis via model checking as before
Eventual Goal: Tightened Vee Diagram

time and money

requirements

design/code

system test

unit/integration test

John Rushby, SRI

Invisible FM and AutoTestGen: 37
Summary: Automated Test Generation

- Simple ideas that significantly improves the efficiency of test sets generated by a model checker
  - Extend current test to new goals
  - Search to any uncovered goal
  - Slice model as goals are covered
  - Further improvement: (re)start from any visited state

- Simple implementation in scriptable model checker (SAL)
- Generation is efficient also
- Independent of test criteria: just set the trap variables
- Many opportunities for further research in test generation
- The paper, SAL Scheme scripts, and examples, are available from http://www.csl.sri.com/users/rushby/abstracts/sefm04
Summary: Formal Methods

- It is now fairly routine to have model checkers as backends to theorem provers (e.g., PVS), or proof assistants as front ends to model checkers (e.g., Cadence SMV)
- But we envisage a larger collection of symbolic computational procedures
  - Decision procedures, abstractors, invariant generators, model checkers, static analyzers, test generators, ITPs
- Interacting through a scriptable tool bus
- The bus manages symbolic and concrete artifacts
  - Test cases, abstractions, theorems, invariants
  Over which it performs evidence management
- Focus shifts from verification to symbolic analysis
  - Iterative application of analysis to artifacts to yield new artifacts, insight and evidence
Integrated, Iterated Analysis
Summary: *Invisible* Formal Methods

- Model-based design methods are a (once-in-a-lifetime?) opportunity to get at formal artifacts early enough in the lifecycle to apply useful analysis within the design loop.
- And formal analysis tools are now powerful enough to do useful things without interactive guidance.
- The challenge is to find good ways to put these two together:
  - Deliver analyses of interest and value to the developers.
  - Or certifiers.
  - But must fit in their flow.
- So can shift from technology push to pull.

*Invisible* (or disappearing) formal methods is our slogan for this approach: apply formal automation to familiar practices.
Summary: Technology

- The technology of automated deduction (and the speed of commodity workstations) has reached a point where we can solve problems of real interest and value to developers of embedded systems.

- Embodied in our systems:
  - **SAL.csl.sri.com**: symbolic analysis laboratory
    - Provides state-of-the-art model checking toolkit (explicit, symbolic, witness, bounded, infinite-bounded)
    - Tool bus (soon)
  - **PVS.csl.cri.com**: comprehensive interactive theorem prover
  - **ICS.csl.sri.com**: embedded decision procedures

- And in numerous papers accessible from [http://fm.csl.sri.com](http://fm.csl.sri.com), including our Roadmap.
Vision: 21st Century Software Engineering

- Symbolic analysis could become the dominant method in systems development and assurance
- And programming could be supplanted by construction of logical models
- And deduction will do the hard work
A Bigger Vision: 21st Century Mathematics

- The industrialization of the 19th and 20th century was based on continuous mathematics
  - And its automation
- That of the 21st century will be based on symbolic mathematics
  - Whose automation is now feasible
  - Allows analysis of systems too complex and numerically too indeterminate for classical methods
- Example: symbolic systems biology
  - Knockouts in E.Coli (SRI; Maude)
  - Cell differentiation in C.Elegans (Weizmann; Play-in/out)
  - Delta-Notch signaling (SRI, Stanford; Hybrid SAL)
  - Sporolation in B.Subtilis (SRI; Hybrid SAL)