Model-based Software Testing and Analysis with C# and NModel

Jonathan Jacky

University of Washington
Seattle, USA

Obtain NModel and samples from:

http://www.codeplex.com/NModel

http://staff.washington.edu/jon/modeling-book/mbta-samples/
(all one URL with no break)

NModel design goals

- Acceptability
  - Familiar modeling language, programming environment
- Simplicity, stability, teachability
  - Core ideas of central importance, permanent value
  - Fundamentals more important than state of the art BUT innovative features included where they permit simplicity
- Lightweight, few dependencies
  - Only C#, .NET, no special compiler or runtime
  - Does not require Visual Studio
- Minimal built-in functionality
- Extensible through user programming
- Platform for research

NModel capabilities I: Model-based testing

- Offline, deterministic
- On-the-fly, nondeterministic
- Optional programmable strategy using coverage points
- Controllable and observable, whole continuum
- Simple, flexible test harnessing
- Concurrent systems (partial order)
- Scenario control to limit testing to interesting scenarios

NModel framework

- Library for writing model programs and tools
- Three simple command line tools written using the library:
  - mpv: model program viewer, visualization and analysis
  - otg: offline test generator, by FSM traversal
  - ct: conformance tester, test runner for both offline and on-the-fly tests

Demo: Traffic light

- Sample traces
- Namespace
- Finite domain here
- State
- Enabling conditions
- Actions, attributes
- ModelProgram, factory method
- mpv, command line arguments, “response files”
- Exploration, simulation (animation)
- Terms
- Alternative notation for FSMs (multiple languages)

Traces are the key

- The purpose of a model program is to generate and check traces

- *Trace* (or *run*): a sequence of actions from the initial state to an accepting state
- *Actions*, units of behavior that precede in discrete steps
- Actions may have arguments
- *Specification* (*contract*): describes all valid traces
- *Scenario*: describes some (maybe just one) interesting traces
- (Demo: Traffic light)
- (more example traces on ms pps 75, 85, 149 etc.)

Actions and arguments

- Choice of actions and arguments determines the level of abstraction.
- Example: API
  - Actions are method calls and returns, arguments are method arguments and return values
- Example: network protocol
  - Actions are message types, arguments are message contents (fields)

Model programs

- Contract model program
  - Purpose: generate all valid traces, check any trace
  - Usually written in C#
  - Methods correspond to actions
- Scenario model program
  - Purpose: describe some (maybe just one) traces for testing or analysis
  - Usually written as FSM (term representation of graph)
  - Transitions (arcs and labels) correspond to actions

Demo: Client/Server (1)

- Based on embedded control system: remote instrument logger (ms page 14)
- Client, server communicate over TCP/IP socket
- Implementation defect: receive buffer too small, results in history-dependent failures
- Purpose of model: automatically generate test suite complex enough to reveal defect
- Really a *reactive system*, but for now test in *sandbox* where all actions are *controllable*

Demo: Client/Server (2)

- Implementation (with defect)
- Sample trace for sandbox testing (ms p. 75)
- Contract model program
- Finitize with [Domain]
- Explore contract model program
  - Many interleavings
  - Dead states
- Offline test generation by traversing contract model program, otg + mpv
  - Split actions for call/return

Demo: Client/Server (3)

- Stepper (test harness)
  - Simple, flexible, requires programming
  - Can adapt to different implementation (static vs. objects, .NET vs. something else, etc.)
  - Works with terms
- Test runner
  - Here executes test suite generated offline by otg
- Minimal transition coverage does not reveal the defect! Motivation for on-the-fly testing

Demo: Client/Server (4)

- Use composition for scenario control
- Composition synchronizes on shared actions and interleaves unshared actions
- Express scenarios in FSM notation
- Compose scenario with contract model
- Generate test suite from product
  - Smaller test suite (one run) because many interleavings have been eliminated

Client/Server Demo (5)

- On-the-fly testing
- Default random strategy
- Random tests on contract model produce few interesting runs
- Random tests on product produce interesting runs, reveal defect

Client/Server Demo (6): What was not included

- Custom strategy (for longer runs)
- Reactive system (where server responses are observable actions)

NModel capabilities II: Design Analysis

- Essential for validating models prior to testing
- Alternative to specification languages, model checking
- Safety, liveness analyses
- Both state-based (reachability) and scenario-based (intersection of FSMs)
- Scenario control to limit analysis to interesting scenarios

State-based analyses

- Safety analysis: are unsafe states ever reachable?
- Liveness analysis: are accepting states always reachable?
- Reactive system demo
  - Based on safety-critical embedded system

Scenario-based analysis

- Is a given run possible?
  - Simple user interface demo
- Is a given scenario possible?
  - Reactive system demo

Structuring models

- Combine independently-written models
- Synchronize on shared actions, interleave unshared actions
- Features: share state, C#, often used for parameter generation and other precondition strengthening
- Composition: does not share state, usually FSMs, can even be used for parameter generation
- Little experience but many opportunities (example: Client/Server demo)

Modeling library data types

- Collections: Tuples, Sets, Maps, Bags, Sequences, Value arrays
  - Value types: immutable, structural equality
  - Needed for state equality
- Objects: Labeled instance
  - Implemented by field maps
  - Abstract values, isomorph elimination is possible

Notable implementation features

- Different modeling languages supported
- Term representation of actions
- Features, composition used for both scenario control and structuring models
- Modeling library data types
  - – Value types for collections
  - -- Abstract values for objects