Building a Model-Based Test Engine for Dynamic Test Automation

J. Benjamin Simo
Ben@QualityFrog.com
Ben_Simo@StandardAndPoors.com

Software Is Automation

It makes sense to automate software testing.

Software helps us do many tasks faster, better, and cheaper.

Why does automation seldom deliver faster, better, and cheaper testing?

Common Test Automation Pitfalls

1. Tests are difficult to maintain and manage
2. Test results are difficult to understand
3. Application changes and bugs can prevent tests from completing
4. Tests repeat the same test over and over and over and … don't find new bugs

Common Test Automation Methods

- Record/Playback (1st Generation)
  - Anyone can automate tests
- Data Driven Tests
  - Repeat test with different data
- Application-Specific Framework
  - Reusable components
- Keyword Driven Framework (3rd Generation)
  - Application-independent reusable components
  - Separate test definition from script coding

Why are the promises of test automation rarely realized?

Too many automation efforts attempt to automate manual testing.

Too many testers (and managers) think their automated tests do the same things a manual tester does.

James Bach's Rules of Test Automation

Rule #1

A good manual test cannot be automated.

Rule #1B

If you can truly automate a manual test,
it couldn't have been a good manual test.

Rule #1C

If you have a great automated test, it's not the same as the manual test that you believe you were automating.

Manual Tests Cannot Be Automated
Monday, July 31st, 2006
http://www.satisfice.com/blog/archives/58

Intelligently Designed Tests

- Testers create scripted test cases from mental models of an application's expected behavior at the time of scripting.
- Testers perform exploratory testing based on a tester's growing mental model of an applications expected and actual behavior at the time of execution.

What if testers could document their mental model and let the computer generate and execute tests?

Artificial Intelligence Meets Random Selection

- Model behavior instead of scripting specific test procedures
- Randomly generate and execute tests based on the model

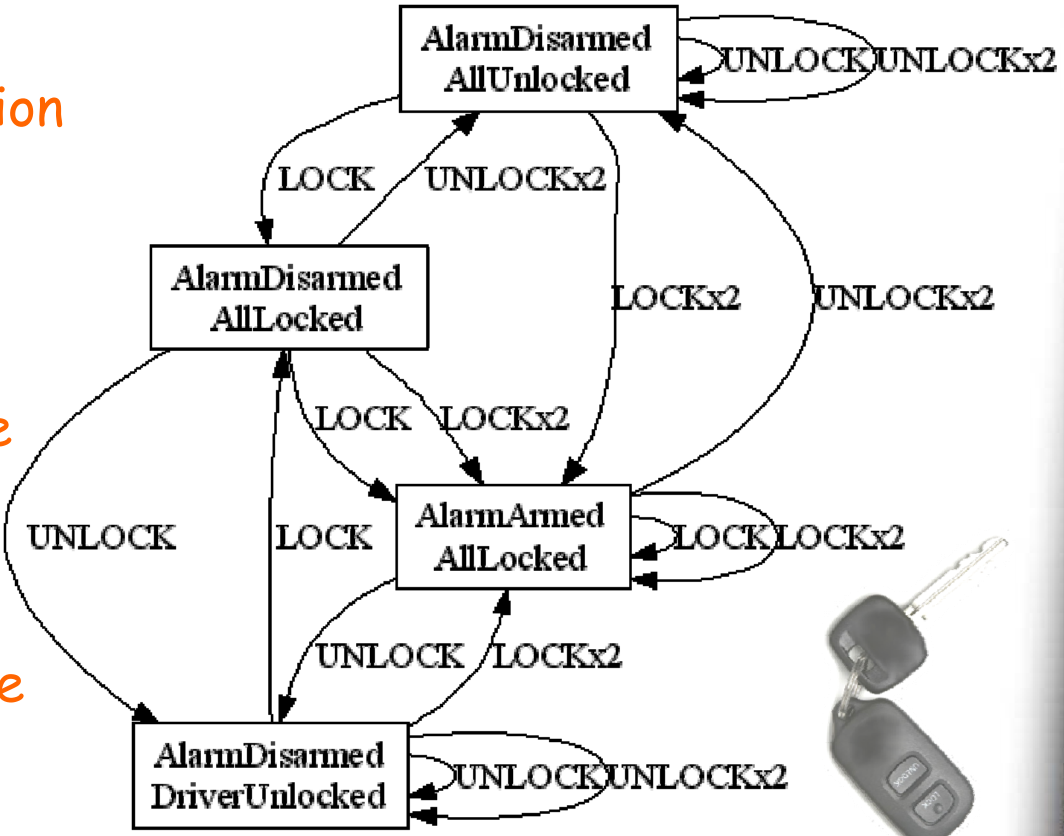
This is ^ Model-Based Testing
Automated

How do we model behavior?

Finite State Machine

Behavior model composed of states, transitions, and actions

- State: The current condition that reflects past changes
- Transition: A change in state
- Action: Event that causes a change of state



Action Table

State transitions and the actions that trigger them can be defined in an action table

Action	StartState	EndState
LOCK	AlarmDisarmed_AllUnlocked	AlarmDisarmed_AllLocked
LOCK	AlarmDisarmed_DriverUnlocked	AlarmDisarmed_AllLocked
LOCK	AlarmDisarmed_AllLocked	AlarmDisarmed_AllLocked
LOCK	AlarmArmed_AllLocked	AlarmArmed_AllLocked
LOCKx2	AlarmDisarmed_AllUnlocked	AlarmArmed_AllLocked
LOCKx2	AlarmDisarmed_DriverUnlocked	AlarmArmed_AllLocked
LOCKx2	AlarmDisarmed_AllLocked	AlarmArmed_AllLocked
LOCKx2	AlarmArmed_AllLocked	AlarmArmed_AllLocked
UNLOCK	AlarmDisarmed_AllUnlocked	AlarmDisarmed_AllUnlocked
UNLOCK	AlarmDisarmed_DriverUnlocked	AlarmDisarmed_DriverUnlocked
UNLOCK	AlarmDisarmed_AllLocked	AlarmDisarmed_DriverUnlocked
UNLOCK	AlarmArmed_AllLocked	AlarmDisarmed_DriverUnlocked
UNLOCKx2	AlarmDisarmed_AllUnlocked	AlarmDisarmed_AllUnlocked
UNLOCKx2	AlarmDisarmed_DriverUnlocked	AlarmDisarmed_DriverUnlocked
UNLOCKx2	AlarmDisarmed_AllLocked	AlarmDisarmed_AllUnlocked
UNLOCKx2	AlarmArmed_AllLocked	AlarmDisarmed_AllUnlocked

State Validation Table

The requirements for each state can be defined in a state validation table

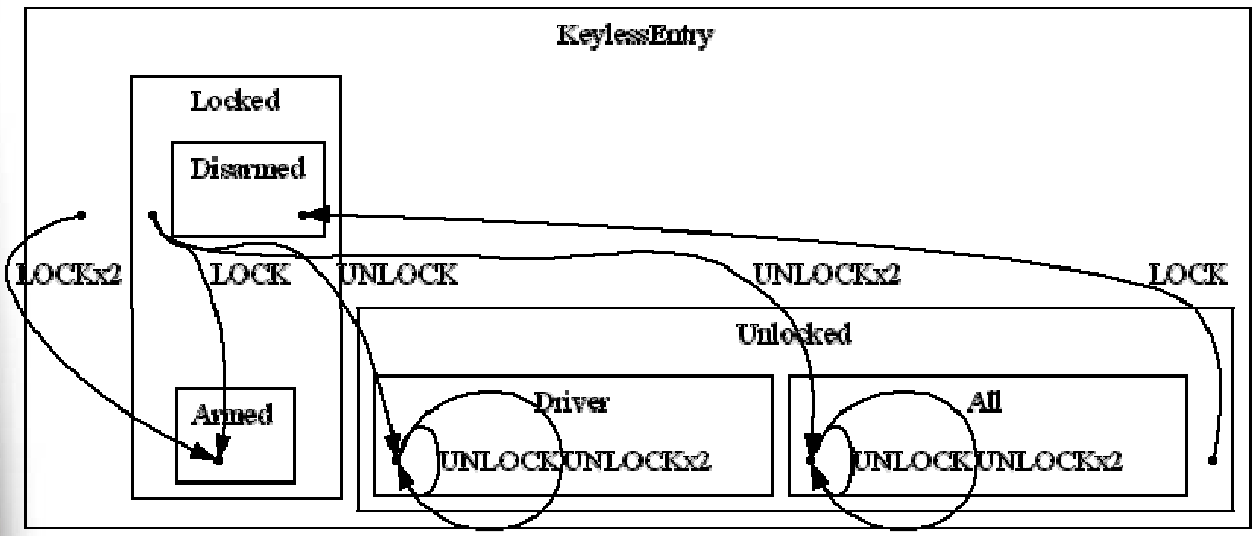
State	**Validation Requirement**
AlarmDisarmed_AllUnlocked	Alarm is disarmed
AlarmDisarmed_AllUnlocked	All doors are unlocked
AlarmDisarmed_DriverUnlocked	Alarm is disarmed
AlarmDisarmed_DriverUnlocked	Driver door is unlocked
AlarmDisarmed_DriverUnlocked	Passenger doors are locked
AlarmDisarmed_AllLocked	Alarm is disarmed
AlarmDisarmed_AllLocked	All doors are locked
AlarmArmed_AllLocked	Alarm is armed
AlarmArmed_AllLocked	All doors are locked

Why aren't more people doing Model-Based Testing?

- Requires a change in thinking
- Difficult to create and manage large models
- Lack of tools

Hierarchical State Machine

- Finite state machines can be simplified by breaking them down into small pieces
- Most states have hierarchical relationships
  - Child (sub) states inherit all attributes of the parent (super) state and have additional attributes that are specific to the child

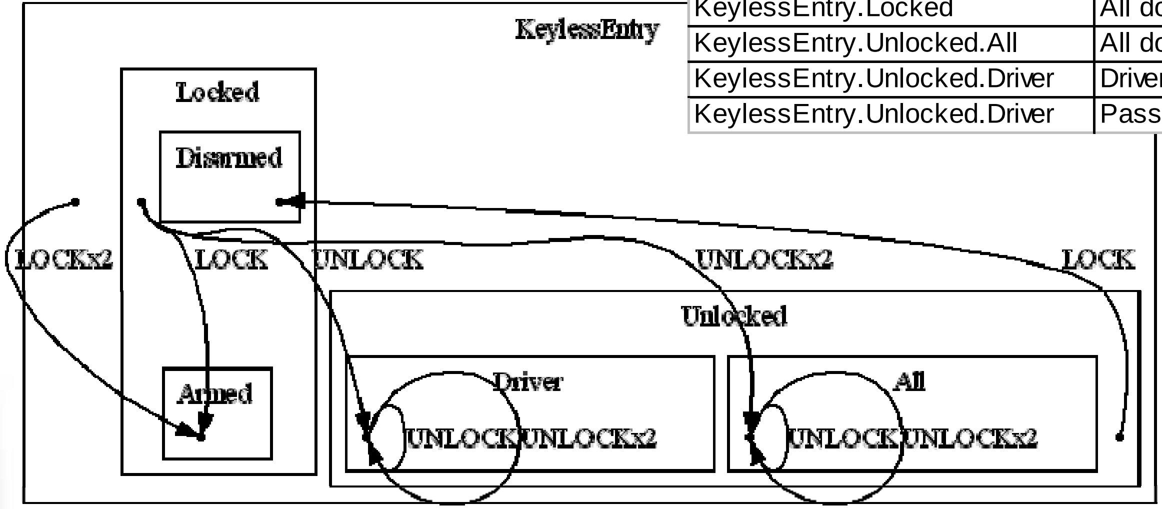


Hierarchical State Machine

Action	Start	
LOCK	Alarm	
LOCK	Alarm	
LOCK	Alarm	
LOCK	Alarm	
LOCKx2	Alarm	
LOCKx2	Alarm	
LOCKx2	Alarm	
LOCKx2	Alarm	
UNLOCK	AlarmDisarmed_AllUnlocked	AlarmDisarmed_AllUnlocked
UNLOCK	AlarmDisarmed_DriverUnlocked	AlarmDisarmed_DriverUnlocked
UNLOCK	AlarmDisarmed_AllLocked	AlarmDisarmed_DriverUnlocked
UNLOCK	AlarmArmed_AllLocked	AlarmDisarmed_DriverUnlocked
UNLOCKx2	AlarmDisarmed_AllUnlocked	AlarmDisarmed_AllUnlocked
UNLOCKx2	AlarmDisarmed_DriverUnlocked	AlarmDisarmed_DriverUnlocke
UNLOCKx2	AlarmDisarmed_AllLocked	AlarmDisarmed_AllUnlocked
UNLOCKx2	AlarmArmed_AllLocked	AlarmDisarmed_AllUnlocked

Action	StartState	EndState
LOCK	KeylessEntry.Locked	KeylessEntry.Locked.Armed
LOCK	KeylessEntry.Unlocked	KeylessEntry.Locked.Disarmed
LOCKx2	KeylessEntry	KeylessEntry.Locked.Armed
UNLOCK	KeylessEntry.Locked	KeylessEntry.Unlocked.Driver
UNLOCK	KeylessEntry.Unlocked.*	KeylessEntry.Unlocked.*
UNLOCKx2	KeylessEntry.Locked	KeylessEntry.Unlocked.All
UNLOCKx2	KeylessEntry.Unlocked.*	KeylessEntry.Unlocked.*

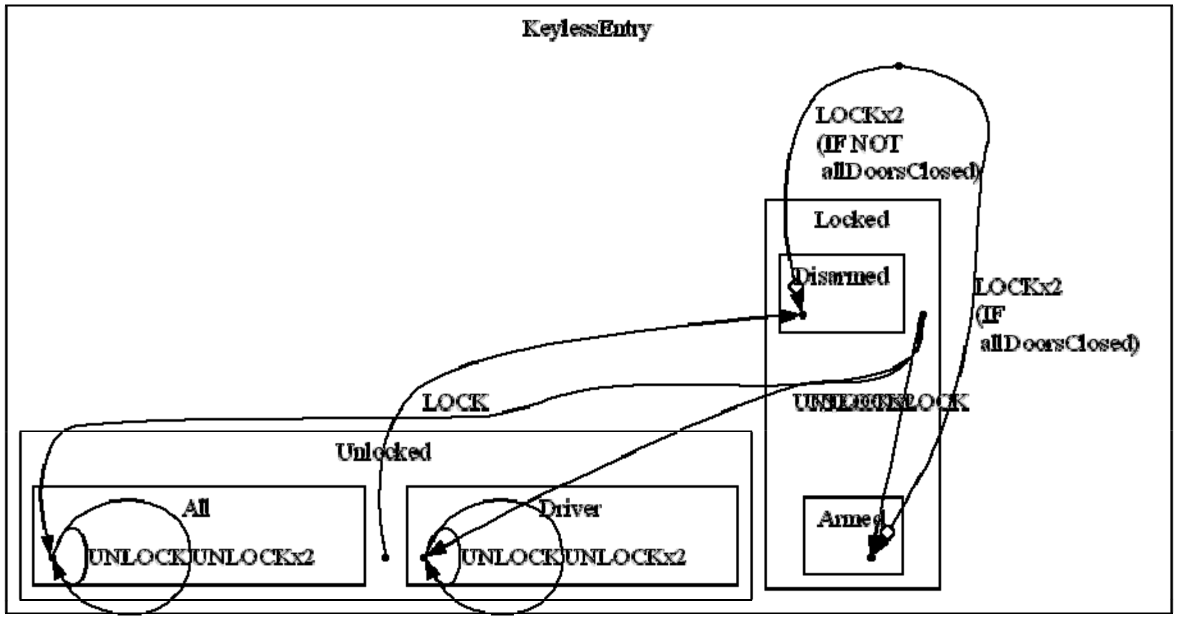
State	Validation Requirement
KeylessEntry.Locked.Armed	Alarm is armed
KeylessEntry.Unlocked	Alarm is disarmed
KeylessEntry.Locked.Disarmed	Alarm is disarmed
KeylessEntry.Locked	All doors are locked
KeylessEntry.Unlocked.All	All doors are unlocked
KeylessEntry.Unlocked.Driver	Driver door is unlocked
KeylessEntry.Unlocked.Driver	Passenger doors are locked



Guarded Transitions

Define some conditions as state variables instead of specific sub-states

Action	IfTrue	StartState	EndState
LOCK		KeylessEntry.Locked	KeylessEntry.Locked.Armed
LOCK		KeylessEntry.Unlocked	KeylessEntry.Locked.Disarmed
LOCKx2	allDoorsLocked	KeylessEntry	KeylessEntry.Locked.Armed
LOCKx2	!allDoorsLocked	KeylessEntry	KeylessEntry.Locked.Disarmed
UNLOCK		KeylessEntry.Locked	KeylessEntry.Unlocked.Driver
UNLOCK		KeylessEntry.Unlocked.*	KeylessEntry.Unlocked.*
UNLOCKx2		KeylessEntry.Locked	KeylessEntry.Unlocked.All
UNLOCKx2		KeylessEntry.Unlocked.*	KeylessEntry.Unlocked.*



Model-Based Test Engine (MBTE)

Automation framework that generates and executes tests based on a model of an application's behavior

- Can be built on top of existing automation scripting tools
- Combines good automation framework practices with Model-Based Testing
- Easier to create than complex Keyword-Driven Frameworks

MBTE Benefits

1. Simplified automation creation and maintenance
2. Simplified test result analysis
3. Automatic handling of most application changes and bugs
4. Generate and execute new tests … and find new bugs

Avoid the Test Automation Pitfalls

1. Simplified automation creation and maintenance

- Create new functions/methods for action execution, state validation, and results reporting
  - Reduce the required scripting vocabulary
  - Add features that aren't native to the tool
  - Integrate workarounds for tool issues
- Separate validations from actions
- Define tests with data instead of code
- Automatic validation detection

2. Simplified test result analysis

- Standardize results reporting
  - Don't rely on the tool's built-in reporting
  - Viewing results should not require an expensive tool
  - HTML, XML, Excel, Database
- Standardize results presentation
- Report enough information to trace errors to the application under test <u>and</u> the automation code

3. Automatic handling of most application changes and bugs

- Return a pass/fail status from every action and validation
- Build error handling into the framework instead of defining error handling for each test
- Use validations to identify test-stopping failures

4. Generate and execute new tests … and find new bugs

- Random action selection
  - Not all tests will be of value
    - Computers don't mind running tests all night
    - Computer hardware costs less than people
- Test data selection
  - Random by object class
  - From data tables
  - Orthogonal Arrays

Test Components

1. Action Tables
   - Define actions and the resulting state changes
2. Data-Driven Test Tables
   - Define test data to be used by the actions
3. State Tables
   - Define state validation requirements
4. Object Map
   - Links logical object names to the tool's method of describing UI objects
5. Call Script
   - Simple script that configures the test and starts the MBTE

Action Table Columns

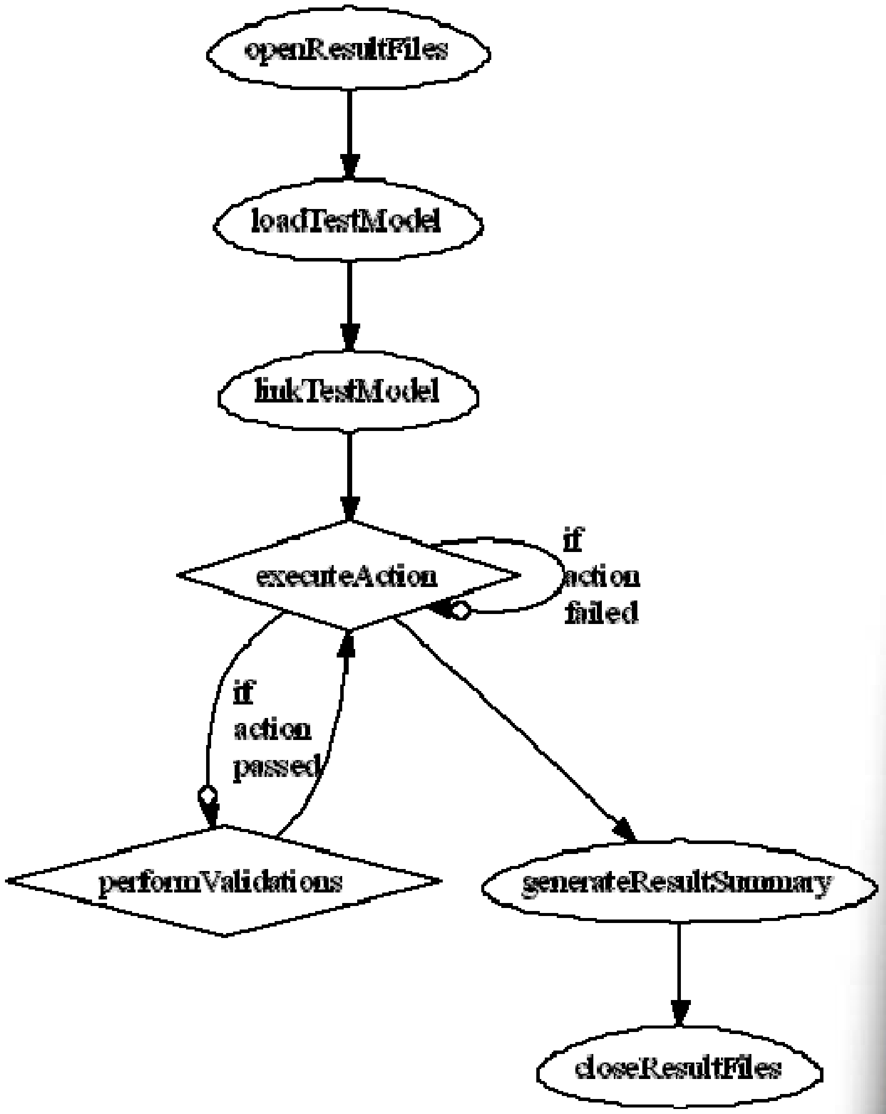
- <u>memberOf</u>: List of test sets to which the action belongs
- <u>testTitle</u>: Short title for the action
- <u>testDetail</u>: Detailed description of the action
- <u>ifTrue</u>: Expression defining data conditions required to execute the action
- <u>startState</u>: Name of the state from which the action is possible
- <u>endState</u>: Name of the expected state after executing the action
- <u>setupCode</u>: Code to execute prior to performing the action
- <u>testCode</u>: Code to execute to perform the action
  - actionWindow, actionObject, actionValue, actionSync

State Table Columns

- severity: numeric severity value given to the requirements
- expectedState: name of the state to which the requirement applies
- ifTrue: expression defining any data conditions that must be met for the requirement to apply
- expectedResult: description of the requirement
- testCode: code to execute to perform the validation
  - testWindow, testObject, testProperty, expectedCondition, expectedValue
- failState: state of the application if the validation fails

Basic MBTE Workflow

- Action execution
  - For a specified period of time
  - Test all test set actions
- Action selection
  - Random
  - Weighted-random
  - Untested first



MBTE Components

1. Main Script
2. Result Reporting
   result_open, result_message, result_action, result_validation, result_close
3. Action Execution
   object_set, action_first, action_next, action_do, ddt_action_do
4. State Validation
   test_window, test_object, test_table
5. Model-Processing
   1. Read model into memory: mbte_action_readTable, mbte_state_readTable
   2. Link actions and validations: mbte_link_actions, mbte_link_validations
   3. UML Image Generation: mbte_write_model
   4. Select Action: mbte_find_next_action
   5. Perform Action: mbte_perform_action
   6. Validate Results: mbte_validate_end_state

Model Processing

1. Read model from tables
   - Combine multiple tables into a single model
   - Generate action and validation code
2. Link actions and validations
   - Link actions to previous and next actions
   - Link state validations to end states
3. UML image generation
4. Select action
   - Random
   - Untested first with look-ahead
   - Weighted random
5. Perform action
   - Update model based on results (e.g., flag action as not available after repeated failures)
6. Validate results

Issues

- Requires a change in thinking
- No pre-trained staff available
  - No vendor training or certification for model-based testing
- Different metrics
  - How many test cases did a model-based test run?

What's Next?

- Create a MBTE for a load test tool
  - Multiple users simultaneously traversing state models
- Multiple active models
  - Start up a new model for threaded processes (e.g., web browser popup)
- Automated model generation?
- Your ideas?

Happy Modeling

"Essentially, all models are wrong, but some are useful."

"… the practical question is how wrong do they have to be to not be useful."

George Box and Norman Draper (1987)
Empirical Model-Building and Response Surfaces,
p 424 and 74

Ben@QualityFrog.com
Ben_Simo@StandardAndPoors.com