The next step in test automation: computer-generated tests

Antti Kervinen

Intel

antti.kervinen@intel.com

Software Testing Course
Tampere University of Technology
Oct 12th 2012
Welcome to the next level in test automation

Typical testing limitations

- Scripts test the same paths over and over again.
- Nobody tests most of the paths a user can take.
- Huge test suites require lots of maintenance.

Benefits of model-based testing

- Computer generates and executes tests:
  - short ones for quick checks,
  - long ones for testing a huge number of different paths and reliability.
- All tests are generated from a small number of models.
Welcome to the next level in test automation

This presentation will give you:

- basic knowledge on model-based testing.
- basic knowledge to try out the fMBT tool.
Contents

Part I  *Introduction to model-based testing (MBT)*
  - *How MBT differs from test cases and test scripts?*
  - *What are benefits of MBT?*
  - *When MBT is a good choice?*

Part II  *Hands–on: test generation from scratch*
  - *Test generation and execution explained*
  - *Creating a test for MPlayer*
  - *How to generate different tests?*
Traditionally test steps have **fixed order**:

1. Launch Camera application.
2. Capture image.
4. Stop video capture.

This does not depend on

- the **target** of the test: unit, integration, system test... or
- the **purpose** of the test: smoke, regression, reliability, performance test...

Now, let’s see how model-based testing differs from this.
There are no predefined test cases in model-based testing. They are generated automatically. This is possible when you have two things:

**Library** of test steps... and **conditions** when they’re enabled

- preview  
- captureImage  
- startVideoCapt  
- stopVideoCapt

- always  
- if not capturing  
- if not capturing  
- always

From these a model-based testing tool can automatically generate many different tests.
Difference between model-based and test case based testing

Two tests generated from steps and conditions on previous slide.

A reliability test:
- stopVideocapt
- captureImage
- stopVideocapt
- startVideocapt
- stopVideocapt
- preview
- stopVideocapt
- startVideocapt
- preview
- ...

A smoke test:
- startVideocapt
- stopVideocapt
- captureImage
- preview

Test generation parameters define what kind of test is wanted. This will be demonstrated in the next part of the presentation.
Difference between model-based and test case based testing

Now you have learned:

**Part I** *Introduction to model-based testing (MBT)*
- *How MBT differs from test cases and test scripts?*
- *What are benefits of MBT?*
- *When MBT is a good choice?*

**Part II** *Hands–on: test generation from scratch*
- *Test generation and execution explained*
- *Creating a test for MPlayer*
- *How to generate different tests?*

Let’s find out some benefits next.
Benefits of model-based testing

Benefits of model-based testing:

1. Increased test coverage.
2. Easier test case maintenance.

First, an example of increased coverage...
A reliability test:
- stopVideoCapt
- captureImage
- stopVideoCapt
- startVideoCapt
- stopVideoCapt
- preview
- stopVideoCapt
- startVideoCapt
- preview
- ...

Consider the reliability test on the left. If you needed to test these functions, would you have created test cases for
- stopping video capture without starting it first?
- testing video capture before and after using preview?
- creating a preview when video capture is running?

Most often people do not think of all these cases. Now they were covered automatically.
Benefits of model-based testing:

1. Increased test coverage.
2. Easier test case maintenance.

Next, two questions for audience on maintenance...
Question 1 (new feature): You are asked to test that preview, captureImage and video capturing work in portrait, landscape and auto orientation modes. How would you handle this, when you have...

...a dozen test cases for testing them and some of their interactions?

...library of test steps and conditions?

- preview
- captureImage
- startVideoCapt
- stopVideoCapt
- nextOrientationMode

- always
- if not capturing
- if not capturing
- always
- always

What else these steps can test?
Benefits of model-based testing

Question 2 (disabling tests). Due to changes on a platform, the preview feature will be broken on next week’s builds. You are asked to keep testing video and image capturing, and orientations. How would you handle this, when you have...

...a dozen test cases for testing preview, captureImage and video capturing plus the orientation testing modifications?

...library of test steps and conditions?

- preview
- captureImage
- startVideoCapt
- stopVideoCapt
- nextOrientationMode
- always
- never
- if not capturing
- always
- always
Benefits of model-based testing:

1. Increased test coverage.
2. Easier test case maintenance.

We have demonstrated the second benefit. As all test cases are generated, making changes – even big ones – is easy compared to fixing a large number of predefined test cases.
Benefits of model-based testing

Now you have learned:

**Part I** Introduction to model-based testing (MBT)
- How MBT differs from test cases and test scripts?
- What are benefits of MBT?
- When MBT is a good choice?

**Part II** Hands-on: test generation from scratch
- Test generation and execution explained
- Creating a test for MPlayer
- How to generate different tests?

Finally, let’s see where model-based testing is a good choice and where not.
Where model-based testing is a good choice

Model-based testing generates tests for you. **Anything that can be tested with automated test cases, can be tested using model-based testing, too.**

More important than where model-based testing can be applied is: Where model-based testing gives greatest benefits?

Let’s see how to recognize these cases...
Where model-based testing is a good choice

Model-based testing is beneficial, if

- things need to be tested in many situations / configurations

Does a call come through? How about when playing music? Watching videos? Capturing a video? Transferring a file using 3G, Wifi, Bluetooth and USB? And the same for a Skype call?

Just write a test step for each activity, and you will get the tests.
Where model-based testing is a good choice

Model-based testing is beneficial, if

- things need to be tested in many situations / configurations
- you need long tests with lots of variation

Take the previous test steps and make the preconditions liberal: music is played during file transfer, video capturing is started during music is played, etc. And calls are received in different combinations of activity.

Now you can generate tests where the device is used in very interesting ways. If wanted, tools can generate and run a single test for hours or days and keep varying different activity combinations all the time.
Where model-based testing is a good choice

Model-based testing is beneficial, if

- things need to be tested in many situations / configurations
- you need long tests with lots of variation
- many combinations or interleavings to be tested

Test a service with $n$ concurrent users. Is there an interleaving of user actions that renders the system unresponsive for any of the users?
Where model-based testing is a good choice

Model-based testing is beneficial, if

- things need to be tested in many situations / configurations
- you need long tests with lots of variation
- many combinations or interleavings to be tested
- you do monkey testing, fuzzing, . . .

Some model-based testing tools (like fMBT) allow inspecting the state of the system under test in preconditions of test steps. This enables, for instance, generating tests that look which buttons are on the display, choose a test step that clicks one of them, and then look again what could be the next test step. A sophisticated test generator allows testing different combinations more efficiently than a pure random “monkey”.

Antti Kervinen (Intel)
Where model-based testing is a good choice

The other way around, model-based testing is not beneficial, if...

- the system under test is stateless, and
- there are no different combinations (such as parameter values or environment configurations) that should be tested.

For a stateless systems it’s enough to test every input only once. The order in which test steps are executed and inputs sent does not matter. There is no reason to use model-based testing tools for generating many test step sequences in this case.
Summary on Part I

Now you have learned:

**Part I**  *Introduction to model-based testing (MBT)*
- *How MBT differs from test cases and test scripts?*
- *What are benefits of MBT?*
- *When MBT is a good choice?*

**Part II**  *Hands-on: test generation from scratch*
- *Test generation and execution explained*
- *Creating a test for MPlayer*
- *How to generate different tests?*

Next we’ll take a look at the fMBT tool.
Hands-on: test generation from scratch

In this part of the presentation:

- Test generation and execution explained
- Creating a test for MPlayer
- How to generate different tests?
Test generation and execution explained

fMBT test generation and execution:

1. Load test configuration, most importantly model, adapter and end conditions.
fMBT test generation and execution:

1. Load test configuration, most importantly model, adapter and end conditions.
2. Loop...
   - choose one of possible test steps
fMBT test generation and execution:

1. Load test configuration, most importantly model, adapter and end conditions.

2. Loop...
   - choose one of possible test steps
   - try executing it in an adapter
Test generation and execution explained

fMBT test generation and execution:

1. Load test configuration, most importantly model, adapter and end conditions.
2. Loop...
   - choose one of possible test steps
   - try executing it in an adapter
   - validate executed test step reported by the adapter
Test generation and execution explained

**fMBT test generation and execution:**

1. Load test configuration, most importantly model, adapter and end conditions.

2. Loop...
   - choose one of possible test steps
   - try executing it in an adapter
   - validate executed test step reported by the adapter
   - execute the test step in test model
fMBT test generation and execution:

1. Load test configuration, most importantly model, adapter and end conditions.
2. Loop...
   - choose one of possible test steps
   - try executing it in an adapter
   - validate executed test step reported by the adapter
   - execute the test step in test model
3. ...until any of end conditions is met.
fMBT supports two modelling languages: GT (graph transformations) and AAL (a pre/postcondition language with adapter blocks). In this example we’ll use AAL/Python, that is, AAL where the code is written in Python.

AAL syntax for test step definitions:

```plaintext
action "name of this test step" {
    guard() { code: return true iff test step can be executed }
    body() { code: update variables after successful execution }
    adapter() { code: interact with the system under test }
}
```
We will generate tests for MPlayer’s (http://www.mplayerhq.hu) slave mode. In the slave mode MPlayer can be controlled in many ways via standard input. The mode enables using MPlayer as a backend behind a GUI. We will test that pause, continue, step to next / previous file, and adding files to the play list works.
Testing MPlayer’s slave mode

Install git for next steps:
- `sudo apt-get install git`
- `yum install git`

Install fMBT from sources:
- Install dependencies listed in README: https://github.com/01org/fMBT, then
  - `git clone https://github.com/01org/fMBT.git`
  - `cd fMBT; ./autogen.sh; ./configure; make; sudo make install`

(Optional) Install MPlayer to run tests:
- `apt-get install mplayer` (Ubuntu, universe or multiverse)
- `yum install mplayer` (Fedora, rpmfusion)

Download the MPlayer test:
- `git clone https://github.com/askervin/fmbt-mplayertest.git`
- `cd fmbt-mplayertest`

Launch fMBT editor with mplayertest.aal and a test configuration:
- `fmbt-editor mplayertest.aal smoke.conf`
Testing MPlayer’s slave mode

There’re helpers for MPlayer launch, communication and state detection.

```python
aal "mplayerertest" {
    language: python {
        import pexpect
        import commands
        from time import sleep
    
    response_time = 1.0 # seconds, mplayer should respond to commands within this time

    def launchMplayer():
        global mplayer
        mplayer = pexpect.spawn("mplayer -idle -slave -quiet 1.mp3 2.mp3")
        sleep(response_time)

    def command(cmd):
        log("mplayer command: " + cmd)
        mplayer.write(cmd + "\n")
        sleep(response_time)
        log("mplayer response: " + mplayer.read_nonblocking(4096, 0))

    def openedMP3File():
        return int(commands.getoutput("basename $(ls -l /proc/" + str(mplayer.pid) + "/fd | awk '/mp3/{print $11}') .mp3"))

    def playing():
        cmd = "strace -p " + str(mplayer.pid) + " 2>&1 | head -n 25 | grep 'read'"
        status, output = commands.getstatusoutput(cmd)
        return 0 == status
    }
```
Then variable declarations and initialisations.

```plaintext
variables {
    state, song, songcount
}
initial_state {
    state = "playing" # either "playing" or "paused"
    song = 1  # number of the song being played
    songcount = 2  # number of songs in current playlist
    launchMplayer()
}
```

In AAL/Python variables is a comma-separated list of variables whose values can be changed by test steps. (In AAL/C++ it would be standard C/C++ variable declaration.)
Testing MPlayer’s slave mode

First test steps: pause and continue.

```plaintext
action "iPause" {
  guard() { return state == "playing" }
  body() { state = "paused" }
  adapter() {
    assert playing(), "song not played before pausing"
    assert openedMP3File() == song, "wrong song played"
    command("pause")
    assert not playing(), "pausing failed"
  }
}

action "iContinue" {
  guard() { return state == "paused" }
  body() { state = "playing" }
  adapter() {
    assert not playing(), "song not paused before continuing"
    command("pause")
    assert playing(), "continuing failed"
    assert openedMP3File() == song, "wrong song played"
  }
}
```

Generating a test from these two steps will not test much: just that played song can be paused and paused song played – over and over again.
Testing MPlayer’s slave mode

Adding next two test steps: stepping between next and previous songs

```plaintext
action "iNextSong" {
    guard() { return song < songcount }
    body() { song += 1 }
    adapter() { command("pausing_keep pt_step 1") }
}
action "iPrevSong" {
    guard() { return song > 1 }
    body() { song -= 1 }
    adapter() { command("pausing_keep pt_step -1") }
}
```

Test generation starts to pay off already. Stepping between songs 1.mp3 and 2.mp3 will be tested when a song is being paused and played. Furthermore, pause and continue will be tested before and after stepping to both directions. This will test that pause–next–prev–continue will play the paused song, for instance.
Testing MPlayer’s slave mode

And more test steps: add a song to current playlist and reset the playlist

```plaintext
action "iAddSong" {
    guard() { return songcount < 3 }
    body() { songcount += 1 }
    adapter() { command("pausing_keep loadfile " + str(songcount+1) + ".mp3") }
}
action "iNewPlaylist" {
    body() {
        songcount = 1
        song = 1
    }
    adapter() { command("pausing_keep loadfile 1.mp3") }
}
```

These six test steps will most likely win any human-designed test suite for the same functionality in terms of test coverage. They will test that added songs can be stepped into and back. They will test resetting a playlist when playing or paused on the first or the last song. And they will test continuing playback from the correct song in every case.

(Note: actions without guard() are always enabled.)
And even more: test “negative cases” on stepping songs

```plaintext
action "iNextSong - already last" {
    guard() { return song == songcount }
    adapter() { command("pausing_keep pt_step 1") }
}
action "iPrevSong - already first" {
    guard() { return song == 1 }
    adapter() { command("pausing_keep pt_step -1") }
}
```

Stepping beyond ends of playlists will be tested in paused and playing cases, and for playlists with one, two and three songs – and combinations of these. It will also be tested that making this error does not effect the functionality. That is, pausing, continuing, adding songs and correct stepping still works and plays correct songs.

(Note: actions without body() do not change values of variables, that is, the state of the model.)
Testing MPlayer’s slave mode

How to understand and validate what a model will test?

Visualisation on the right differentiates only states which differ by variables. This is the “state” point-of-view.
Testing MPlayer’s slave mode

And this shows only changes on “song” being played.
Testing MPlayer’s slave mode

And finally “songcount”.
Exercise:

Add test step `iRestart` for testing that MPlayer process terminates when it reads “quit” command in the slave mode. If successfully terminated, the test step should launch new instance of MPlayer. This allows continuing the test after the reset. If termination failed, test step should fail (raise an exception, for instance using `assert`).

Hint: `mplayer.isalive()` returns True if the process is running, otherwise False.
Testing MPlayer’s slave mode

Exercise:

Add test step iRestart for testing that MPlayer process terminates when it reads “quit” command in the slave mode. If successfully terminated, the test step should launch new instance of MPlayer. This allows continuing the test after the reset. If termination failed, test step should fail (raise an exception, for instance using assert).

```plaintext
action "iRestart" {
    body() {
        state = "playing"
        song = 1
        songcount = 2
    }
    adapter() {
        command("quit")
        sleep(2)
        assert not mplayer.isalive(), "mplayer did not quit"
        launchMplayer()
    }
}
```
Testing MPlayer’s slave mode

In fmbt-editor, try different test generation parameters, like perm(2), steps(100), in smoke.conf (tab F2):

```plaintext
model       = "aal_remote(remote_pyaal -l adapter.log mplayertest.aal)"
adapter     = "aal_remote(remote_pyaal -l adapter.log mplayertest.aal)"
heuristic   = "lookahead(3)"
coverage    = "perm(1)"

pass        = "coverage(1.0)"
pass        = "steps(80)"

on_pass     = "exit(0)"
on_fail      = "exit(1)"
on_inconc    = "exit(2)"
```

And see how they affect on generated test (tab F6) and coverage (tab F8).
fMBT editor only generates tests. You can run it as follows:

```
fmbt -l smoke.log smoke.conf
```

Inspect the log:

```
fmbt-log smoke.log
```

```
fmbt-log -f 'st ax tv' smoke.log
```

See some statistics:

```
fmbt-stats -f times smoke.log
```
Test generation parameters explained

- **model** – how to find names of test steps and conditions
- **adapter** – how to execute test steps. In case of AAL this can be the same as the model. Adapter configuration does not affect generated tests shown in fmbt-editor, it only affects executed tests.
- **coverage** – how to measure coverage — the greater the returned value the more is covered.
- **heuristic** – how to choose the next test step, for instance, random or lookahead(n).
- **verdict** = "condition" – when to stop test generation, and what is the verdict.

Next, how to generate different tests using coverages, heuristics and end conditions.
**Test generation parameters explained**

- **coverage** defaults to perm(2). It measures the percentage of covered permutations of any 2 test steps.
- Example: if test model contains test steps iPause, iContinue, iNextSong, iPrevSong, perm(2) measures the percentage of covered test step pairs:
  - iPause - iPause
  - iPause - iContinue
  - iPause - iNextSong
  - iPause - iPrevSong
  - iContinue - iPause
  ...
  - iPrevSong - iPrevSong
- Question for audience: What is needed for perm(3) to return 1.0?
- Other coverages: uwalks(from "iPause" to "iContinue") returns the number of unique minimal sequences that start with iPause and end with iContinue.
Test generation parameters explained

- **heuristic** defaults to random. It chooses the next test step with evenly distributed random choice among all enabled test steps.

- lookahead chooses randomly among all enabled test steps that give the largest increase on measured coverage. Does not simulate execution of test steps.

- lookahead\((n)\) simulates test generation \(n\) steps ahead from the current state of the model. It chooses the first test step of an \(n\)-step sequence that results in the best coverage.

- Question for audience: assume that test step iRemoveSongFile is enabled when playing() returns False, that is, MPlayer process is not currently playing any song. How this affects heuristics?

As the test step depends on the state of the real system under test, lookahead\((n)\) cannot simulate its execution. How would you fix the problem? When to use this kind of test steps?
Test generation parameters explained

- **end conditions** stop test generation. Examples:
  - `pass = "coverage(0.5)"
    test is passed when measured coverage reaches 50%.
  - `pass = "no_progress(10)"
    test is passed when measured coverage has not grown within last 10 steps.
  - `pass = "steps(100)"
    test is passed when 100 test steps have been executed.
  - `inconc = "duration(2 minutes)"
    the test is inconclusive if it has lasted two minutes or longer.
  - `fail = "deadlock"
    test fails if the test generation cannot be continued due to none of test steps being enabled.
Summary on Part II

- How to choose the next test step: heuristic makes the choice, often according to measured coverage.

- When test generation is stopped: end conditions define the verdict and when given. May depend on date/time, measured coverage and number of steps, for instance.
Question for audience: how would you compare the trouble of these two:

- adding iRestart test step for testing quit and restart in conjunction with pause/continue/next song/prev song...

- writing test cases for quit when playing, paused, after navigating songs (even with errors), after changing playlists... and the other way around.
Thank you!

Now you have learned:

**Part I** *Introduction to model-based testing (MBT)*
- *How MBT differs from test cases and test scripts?*
- *What are benefits of MBT?*
- *When MBT is a good choice?*

**Part II** *Hands-on: test generation from scratch*
- *Test generation and execution explained*
- *Creating a test for MPlayer*
- *How to generate different tests?*

Questions, comments?