PyModel
Model-based testing in Python

Jon Jacky

University of Washington
jon@u.washington.edu
http://staff.washington.edu/jon/pymodel/www/
Unit testing: code each test case, including an assertion that checks whether the test passed

Model-based testing: code a model that generates as many test cases as desired, and also acts as the oracle that checks whether any case passed

A model-based testing project is a programming project!

In PyModel the models are coded in Python. It is convenient when the implementation under test is also in Python (but this is not required).
What problem does model-based testing solve?

Testing behavior: ongoing activities that may exhibit history-dependence and nondeterminism.

Many variations (data values, interleavings, etc.) should be tested for each scenario (or use case).

So many test cases are needed that it is not feasible to code them all by hand.

Examples: communication protocols, web applications, control systems, user interfaces, ...
We need to test *behavior*: ongoing activities that may exhibit history dependence and nondeterminism.

We represent behavior with *traces*: sequences of *actions* with arguments. Specify a system by describing which traces are allowed, and which are forbidden.

**Example: alternating bit protocol**

```
Allowed
Send(0)  Allowed  Send(1)  Allowed  Send(1)
Ack(0)   Send(1)  Ack(1)   Send(1)
Send(1)  Ack(1)   Send(0)  Ack(1)
Ack(1)   Send(1)  Ack(1)   Ack(1)
Send(0)  Ack(0)   Send(0)  Ack(1)
```

```python
Forbidden
Send(0)  Ack(0)   Send(0)  Send(1)
Ack(0)   Send(0)  Ack(0)   Ack(1)
```
Finite State Machines (FSMs) can represent finite behaviors. Every path through the graph represents an allowed trace.

Allowed:
- Send(0)
- Ack(0)
- Send(1)
- Ack(1)

Forbidden:
- Send(0)
- Ack(0)
- Send(1)
- Ack(1)

Model-based testing in Python

Jon Jacky
FSMs are one kind of model in PyModel, coded as follows:

```
graph = ((0, (Send, (1,), None), 0),
         (0, (Ack, (1,), None), 0),
         (0, (Send, (0,), None), 1),
         (1, (Ack, (0,), None), 2),
         ... etc. ...,
         (4, (Send, (0,), None), 1))
```

The PyModel Graphics program `pmg` generates graphics from an FSM in this form.
The PyModel Tester **pmt** generates traces from a model. Each trace describes a test run, including the expected test results.

**Offline testing**: pmt saves the traces in a test suite.

**On-the-fly testing**: pmt executes the traces as they are generated.

C:`\Users\jon\Documents\mbt\samples\abp>pmt.py -n 10 ABP
Send(1,)
Send(1,)
Ack(1,)
Send(1,)
Ack(1,)
Send(0,)
Ack(1,)
Ack(0,)
Ack(0,)
Finished at step 10, reached accepting state
Model programs are another kind of model in PyModel. They can describe behaviors where the action arguments can have an “infinite” (very large) number of values.

A model program consists of state variables, action functions and enabling conditions.

```python
stack = list()  # State

def Push(x):  # Push is always enabled
    global stack
    stack.insert(0,x)

def Pop():     # Pop requires an enabling condition
    global stack
    result = stack[0]
    del stack[0]
    return result

def PopEnabled():  # Pop is enabled when the stack is not empty
    return stack
```
The PyModel Analyzer **pma** generates an FSM from a model program by a process called *Exploration.*
The PyModel Analyzer **pma** generates an FSM from a model program by a process called *Exploration*.
The PyModel Analyzer **pma** generates an FSM from a model program by a process called *Exploration*.

![State Machine Diagram](image)
The PyModel Analyzer **pma** generates an FSM from a model program by a process called *Exploration*. 

---

**Diagram:**

- State 0
  - Transition: Push(0) → State 1
  - Transition: Push(1) → State 2

- State 1
  - Transition: Push(0) → State 0
  - Transition: Push(1) → State 15

- State 2
  - Transition: Push(0) → State 0
  - Transition: Push(1) → State 16
The PyModel Analyzer **pma** generates an FSM from a model program by a process called *Exploration*.

etc ...
We must limit exploration of infinite programs. Here we define a finite *domain* to limit the width of the graph, and a *state filter* to limit its depth.

```
domains = { 'Push': {'x':[0,1]} }

def StateFilter():
    return len(stack) < 4
```
Test generation can select the next enabled action at random, or use an optional *strategy* to select an action that increases coverage according to some measure.

ActionNameCoverage and StateCoverage are included in PyModel. You can also code your own custom strategy.
We need *scenario control* to limit test runs to scenarios of interest.

PyModel uses *composition*, a versatile technique combines two or more models to form a new model, the *product*.

\[ M_1 \times M_2 = P \]

Usually we combine a *contract model program* (with action functions, etc.) with a *scenario machine*, an FSM.

\[ \text{Contract} \times \text{Scenario} = \text{Product} \]

Composition can also be used for validation, program structuring, etc. . . .
Composition synchronizes shared actions.
Composition synchronizes shared actions.

This usually has the effect of restricting behavior.
Composition synchronizes shared actions.

This usually has the effect of restricting behavior.
Composition interleaves unshared actions.

Composition

0

PowerOn(PowerOff())

1
Composition interleaves unshared actions.
Composition interleaves unshared actions.

This usually has the effect of adding behavior.
Composition with a scenario can help validate a model program.
Composition with a scenario can help validate a model program.

The product shows whether the model program can execute the complete scenario. Does the product reach an accepting state?
Composition with a scenario can help validate a model program.

The product shows whether the model program can execute the complete scenario. Does the product reach an accepting state?
In this example we compose the model program with a scenario machine to eliminate redundant startup and shutdown paths.
In this example we compose the model program with a scenario machine to eliminate redundant startup and shutdown paths.
In this example we compose the model program with a scenario machine to eliminate redundant startup and shutdown paths.

Now the product will only generate interesting traces.
Executing tests requires a harness (or adapter) to connect the model to the implementation. In PyModel a test harness is called a *stepper*. Its core `TestAction` function contains a branch for each action in the model.

```python
def TestAction(aname, args, modelResult):
    ...

    if aname == 'Initialize':
        session = dict()  # clear out cookies/session IDs from previous session

    elif aname == 'Login':
        user = users[args[0]]
        ...
        password = passwords[user] if args[1] == 'Correct' else wrongPassword
        postArgs = urllib.urlencode({'username':user, 'password':password})
        page = session[user].opener.open(webAppUrl).read()  # GET login page
        ...
        if result != modelResult:
            return 'received Login %s, expected %s' % (result, modelResult)

    elif aname == 'Logout':
        ...
```

Jon Jacky

Model-based testing in Python
On-the-fly testing generates test cases as the test run executes. It overcomes some disadvantages of offline test generation.

- No FSM is generated, needn’t finitize
- Test runs can be indefinitely long, nonrepeating

Especially

- Handles nondeterminism — responds to the nondeterministic choice that the implementation actually made!

This last requires an asynchronous stepper that distinguishes between controllable actions (functions the stepper can call) and observable actions (events the stepper can detect).
Implications

Model-based testing can encourage different approaches to testing.

- Encourages on-the-fly testing — but test runs may not be reproducible.
- Extends testing to noninvasive monitoring or run time verification — if the harness supports observable actions, the tester can check log files or monitor network traffic for conformance violations.
- Enables better integration of design analysis with testing — exploration is like model checking, can check for safety, liveness, and temporal properties.

A rational workflow might be to write the model before writing the implementation, analyze and tweak the design, then implement and test.
PyModel is an open-source model-based testing framework.

PyModel: Model-based testing in Python

Jonathan Jacky

PyModel is an open-source model-based testing framework in Python.

In unit testing, the programmer codes the test cases, and also codes assertions that check whether each test case passed. In model-based testing, the programmer codes a "model" that generates as many test cases as desired and also acts as the oracle that checks the cases.

Model-based testing is recommended where so many test cases are needed that it is not feasible to code them all by hand. This need arises when testing behaviors that exhibit history-dependence and nondeterminism, so that many variations (data values, interleavings, etc.) should be tested for each.

http://staff.washington.edu/jon/pymodel/www/