PyModel: Model-based testing in Python

Jonathan Jacky

Abstract—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 scenario (or use case). Examples include communication protocols, web applications, control systems, and user interfaces. PyModel is a model-based testing framework in Python. PyModel supports on-the-fly testing, which can generate indefinitely long nonrepeating tests as the test run executes. PyModel can focus test cases on scenarios of interest by composition, a versatile technique that combines models by synchronizing shared actions and interleaving unshared actions. PyModel can guide test coverage according to programmable strategies coded by the programmer.

Index Terms—testing, model-based testing, automated testing, executable specification, finite state machine, nondeterminism, exploration, offline testing, on-the-fly testing, scenario, composition

Introduction

Model-based testing automatically generates, executes, and checks any desired number of test cases, of any desired length or complexity, given only a fixed amount of programming effort. This contrasts with unit testing, where additional programming effort is needed to code each test case.

Model-based testing is intended to check *behavior*: ongoing activities that may exhibit history-dependence and nondeterminism. The correctness of behavior may depend on its entire history, not just its most recent action. This contrasts with typical unit testing, which checks particular *results*, such as the return value of a function, given some arguments.

It is advisable to check entire behaviors, not just particular results, when testing applications such as communication protocols, web services, embedded control systems, and user interfaces. Many different variations (data values, interleavings etc.) should be tested for each scenario (or use case). This is only feasible with some kind of automated test generation and checking.

Model-based testing is an automated testing technology that uses an executable specification called a *model program* as both the test case generator and the oracle that checks the results of each test case. The developer or test engineer must write a model program for each implementation program or system they wish to test. They must also write a *test harness* to connect the model program to the (generic) test runner.

The corresponding author is with University of Washington, e-mail: jon@uw.edu.

With model program and test harness in hand, developers or testers can use the tools of the model-based testing framework in various activities: Before generating tests from a model, it is helpful to use an *analyzer* to validate the model program, visualize its behaviors, and (optionally) perform safety and liveness analyses. An *offline test generator* generates test cases and expected test results from the model program, which can later be executed and checked by a *test runner* connected to the implementation through the test harness. This is a similar workflow to unit testing, except the test cases and expected results are generated automatically. In contrast, *on-the-fly testing* is quite different: the test runner generates the test case from the model as the test run is executing. On-the-fly testing can execute indefinitely long nonrepeating test runs, and can accommodate nondeterminism in the implementation or its environment.

To focus automated test generation on scenarios of interest, it is possible to code an optional *scenario machine*, a lightweight model that describes a particular scenario. The tools can combine this with the comprehensive *contract model program* using an operation called *composition*. It is also possible to code an optional *strategy* in order to improve test coverage according to some chosen measure. Some useful strategies are already provided.

Model-based testing supports close integration of design and analysis with testing. The analyzer is similar to a model checker; it can can check safety, liveness, and temporal properties. And, the *same models* are used for these analyses as for automated testing. Moreover, the models are written in the *same language* as the implementation.

PyModel is an open-source model-based testing framework for Python [PyModel11]. It provides the PyModel Analyzer `pma`, the PyModel Graphics program `pmg` for visualizing the analyzer output, and the PyModel Tester `pmt` for generating, executing, and checking tests, both offline and on-the-fly. It also includes several demonstration samples, each including a contract model program, scenario machines, and a test harness.

The PyModel framework is written in Python. The models and scenarios must be written in Python. It is often convenient, but not required, if the system under test is also written in Python, because it can be easier to write the test harness in that case.

Traces and Actions

We need to describe behavior. To show how, we discuss the Alternating Bit Protocol [ABP11], a simple example that exhibits history-dependence and nondeterminism. The protocol

is designed to send messages over an unreliable network. The sender keeps sending the same message, labeled with the same bit (1 or 0), until the receiver acknowledges successful receipt by sending back the same bit. The sender then complements the bit and sends a new message labeled with the new bit until it receives an acknowledgement with that new bit, and so on. When the connection starts up, both ends send bit 1. The sender labels the first real message with 0.

A sample of behavior is called a *trace*. A trace is a sequence of *actions*, where each action has a name and may have arguments (so actions resemble function calls). The alternating bit protocol has only two actions, named `Send` and `Ack`. Each action has one argument that can take on only two values, `0` or `1`. (We abstract away the message contents, which do not affect the protocol behavior.) Here are some traces that are allowed by the protocol, and others that are forbidden:

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

Traces like these might be collected by a test harness connected to the sender. The `Send` are *controllable actions* invoked by the sender while the `Ack` are *observable actions* that are observed by monitoring the network. (If the test harness were connected to the receiver instead, the `Send` would be the observable action and the `Ack` would be controllable.)

Finite Models

A model is an executable specification that can generate traces (to use as test cases) or check traces (to act as an oracle). To act as a specification, the model must be able to generate (or accept) any allowed trace and must not be able to generate any forbidden trace (it must reject any forbidden trace).

The alternating bit protocol is *finite* because there are only a finite number of actions (only a finite number of possible values for each action argument). Therefore this protocol can be modeled by a *finite state machine* (FSM), which can be represented by a graph where the edges represent actions and the nodes represent states (Figure 1). Every allowed trace can be obtained by traversing paths around this graph. In the figure, some of the nodes have doubled borders. These are the *accepting states* where traces are allowed to stop. A trace that stops in a non-accepting state is forbidden. If no accepting states are specified, all states are considered accepting states.

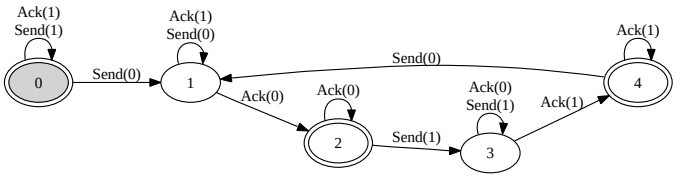

Figure 1: *Alternating bit protocol represented by a finite state machine (FSM)*

In PyModel, a finite state machine is represented by its graph: a tuple of tuples, where each tuple represents a state transition, the current state (a node), the action (an edge), and the next state (another node):

```
graph = ((0, (Send, (1,),), 0),
         (0, (Ack, (1,),), 0),
         (0, (Send, (0,),), 1),
         (1, (Ack, (0,),), 2),
         ... etc. ...
         (4, (Send, (0,),), 1))
```

The PyModel Graphics program `pmg` generated Figure 1 from this code.

Most interesting systems are infinite and cannot be described by finite state machines. In PyModel, finite state machines are most often used to describe *scenario machines* that are composed with infinite *contract model programs* to focus test case generation on scenarios of interest.

Infinite Models

Most interesting systems require infinite models. A system requires an infinite model when it has an infinite number of actions. This occurs whenever any of its action arguments are drawn from types that have an infinite number of values: numbers, strings, or compound types such as tuples.

Simple systems can be infinite. Consider a stack, a last-in first-out queue which provides a `Push` action that puts a value on top of the stack and a `Pop` action that removes the value from the top of the stack and returns it. Here are some allowed traces:

```
Push(1,)      Push(1,)      Push(1,)
Push(2,)      Pop(), 1      Push(2,)
Push(2,)      Push(2,)      Push(2,)
Push(1,)      Pop(), 2      Push(1,)
Pop(), 1      Push(1,)      Push(1,)
Pop(), 2      Pop(), 1      Push(1,)
Pop(), 2      Push(2,)      Push(2,)
Push(2,)      Pop(), 2      Push(2,)
Push(1.)      Push(1,)      Push(1,)
Push(1,)      Pop(), 1      Push(1,)
```

In PyModel, an infinite model is expressed by a Python module with an *action function* for each action and variables to represent the *state*, the information stored in the system. In this example, the state is a list that stores the stack contents in order. Constraints on the ordering of actions are expressed by providing each action with an optional *guard* or *enabling condition*: a Boolean function that is true for all combinations of arguments and state variables where the action is allowed to occur. In this example, `Push` is always enabled so no enabling function is needed; `Pop` is only enabled in states where the stack is not empty. Here is the model, as coded in the module `Stack`:

```
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 enabled when stack not empty
  return stack
```

Analysis

It can be helpful to visualize the behavior of a model program. The PyModel Graphics program `pmg` can generate a graph from finite state machine, as in Figure 1. The PyModel Analyzer `pma` generates a finite state machine from an infinite model program, by a process called *exploration* which is a kind of concrete state model-checking. In order to finitize the model program, it is necessary to limit the action arguments to finite *domains* and it may also be necessary to limit the state by *state filters*, Boolean functions which the state must satisfy. Exploration in effect performs exhaustive testing of the model program over these finite domains, generating all possible traces and representing them compactly as an FSM.

Here we define a domain that limits the arguments of `Push` to the domain `0, 1`; we also define a state filter that limits the stack to fewer than four elements:

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

def StateFilter():
  return len(stack) < 4
```

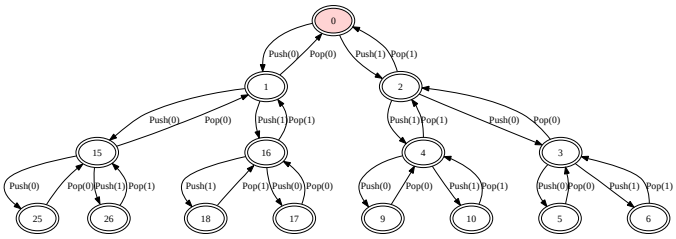

Figure 2: *FSM for finitized Stack model program, generated by exploration.*

Subject to these limitations, `pma` generates a finite state machine that is rendered by `pmg` (Figure 2).

Every trace allowed by the (finitized) model can be obtained by traversing paths around the graph. This is useful for validation: you can check whether the graph allows the expected behaviors.

Safety and Liveness

In addition to providing visualization, the analyzer can check other properties. *Safety analysis* checks whether anything bad can happen. You specify safety requirements by defining a *state invariant*, a Boolean function on state variables that is supposed to be satisfied in every state. The analyzer checks the invariant in every state reached during exploration and marks *unsafe states* where the invariant is violated. *Liveness analysis* checks whether something good will happen. You specify liveness requirements by defining an *accepting state condition*, a Boolean function on state variables that is supposed to be satisfied in the states where a trace ends. The analyzer checks the accepting state condition in every state and marks the terminal states (which have no outgoing actions) where the condition is violated; these are *dead states* from which an accepting state cannot be reached. Since exploration is exhaustive, these analyses are conclusive; they are machine-generated proofs that the safety and liveness properties hold (or not) for the model program over the given finite domains.

Test Harness

In order to execute tests, it is necessary to write a *test harness* that connects the model program to the test runner `pmt`. The test harness usually encapsulates the implementation details that are abstracted away from the model. It is often convenient, but not required, if the implementation under test is also written in Python, because it can be easier to write the test harness in that case.

Here is a fragment of the code from the harness for testing a web application. As it happens, the server code of the web application that we are testing here is in PHP, not Python, but this is not an inconvenience because the test harness acts as a remote web client, using the Python standard library module `urllib`, among others. The model includes `Initialize`, `Login`, and `Logout` actions, among others:

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

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

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

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

Offline Testing

Offline testing uses a similar workflow to unit testing, except the test cases and expected results are generated automatically from the model program.

Traces can be used as test cases. The PyModel Tester `pmt` can generate traces from a (finitized) model program; these include the expected return values from function calls, so they contain all the information needed for testing. Later, `pmt` can act as the test runner: it executes the generated tests (via the test harness) and checks that the return values from the implementation match the ones in the trace calculated by the model program.

On-the-fly Testing

In *On-the-fly testing* the test runner `pmt` generates the test case from the model as the test run is executing. On-the-fly testing

can execute indefinitely long nonrepeating test runs. On-the-fly testing is necessary to accommodate nondeterminism in the implementation or its environment.

Accommodating nondeterminism requires distinguishing between *controllable actions* (functions that the test runner can call via the test harness), and *observable actions* (events that the test harness can detect). For example, when testing the sender side of the alternating bit protocol, `Send` is controllable and `Ack` is observable. Handling observable actions may require asynchronous programming techniques in the test harness.

Strategies

During test generation, alternatives arise in every state where multiple actions are enabled (that is, where there are multiple outgoing edges in the graph of the FSM). Only one action can be chosen. The algorithm for choosing the action is called a *strategy*. In PyModel, the default strategy is random choice among the enabled actions. It is also possible to code an optional *strategy* in order to improve test coverage according to some chosen measure.

Some useful strategies are already provided. The `ActionNameCoverage` strategy chooses different actions, while the `StateCoverage` strategy attempts to reach unvisited states. Here are some test cases generated from the stack model using different strategies:

```
Random          Action name     State
(default)       coverage        coverage
--------        --------        --------
Push(1,)        Push(1,)        Push(1,)
Push(2,)        Pop(), 1        Push(2,)
Push(2,)        Push(2,)        Push(2,)
Push(1,)        Pop(), 2        Push(1,)
Pop(), 1        Push(1,)        Push(1,)
Pop(), 2        Pop(), 1        Push(1,)
Pop(), 2        Push(2,)        Push(2,)
Push(2,)        Pop(), 2        Push(2,)
Push(1.)        Push(1,)        Push(1,)
Push(1,)        Pop(), 1        Push(1,)
```

Composition

Composition is a versatile technique that combines models. PyModel uses it for scenario control, validation, and program structuring. All of the PyModel commands can accept a list of models to be composed in any context where they expect a model.

Composition combines two or more models to form a new model, the *product*. (In the following discussion and examples, just two models are composed.)

$$M_1 \times M_2 = P$$

When the product is explored, or is used to generate or check traces, PyModel in effect executes the composed models in parallel, synchronizing on shared actions and interleaving unshared actions. A shared action occurs in both models, an unshared action occurs in only one. A shared action must be simultaneously enabled in both models in order to execute in the product. This results in synchronizing the execution of the shared actions. This usually has the effect of limiting or restricting behavior, in effect filtering it (Figure 3). This is useful for both scenario control and validation, as we shall see.

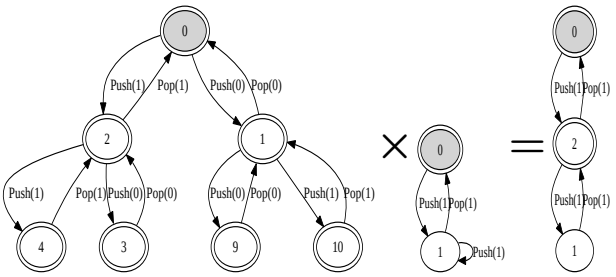

Figure 3: *Composition synchronizes on shared actions.*

An unshared action can execute in the product whenever it is enabled in its own model. This results in interleaving the execution of the unshared actions in the product. This usually has the effect of enlarging the behavior, in effect multiplying it (Figure 4). This can be useful as a structuring technique for building up complex models from simpler ones.

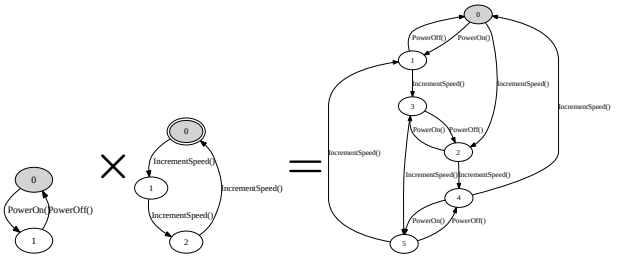

Figure 4: *Composition interleaves unshared actions.*

Notice that a state is an accepting state in the product if and only if it is an accepting state in both of the composed models.

Scenario Control

A difficulty with any automated testing method is generating too many tests. We need *scenario control* to limit test runs to scenarios of interest. We can achieve this by composing the comprehensive *contract model program*, usually a Python module with state variables etc., with a particular *scenario machine*, usually an FSM.

$$Contract \times Scenario = Product$$

In this example (Figure 5), the contract model program (on the far left) allows many redundant, uninteresting startup and shutdown paths. We would like to intensively test just the few interesting actions in this model. We create a scenario machine (on the near left) that specifies a single path through startup

and shutdown, and omits the interesting actions. When we compose the two models, the startup and shutdown actions are shared so the two models must synchronize, which forces the product to follow the sequences in the scenario. The interesting actions are unshared, so they are free to interleave, and the product can execute these as long as they are enabled. The product (on the right) will only generate traces that are interesting for this test purpose.

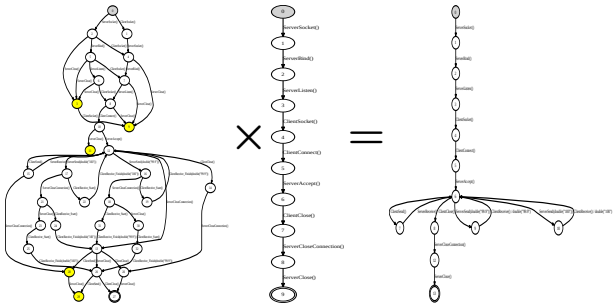

Figure 5: *Composition with a scenario can eliminate uninteresting paths from tests.*

Validation

A model program is just a program so it is necessary to *validate* it: to confirm that it expresses the intended behaviors. As already noted, simply inspecting the graphs generated by the analyzer can be helpful for this.

Composition also supports a more rigorous validation procedure analogous to unit testing. Composing a contract model program with a scenario machine results in a product that reaches an accepting state if and only if the model allows the behaviors described by the scenario, that is, if the model can execute the scenario. If the model cannot execute the scenario, the product will not reach an accepting state. Therefore, a collection of scenarios that are each known *a priori* to be allowed or forbidden can act as a unit test suite for a model program. Composing the model with each scenario in turn is, in effect, executing the unit test suite.

Figures 3 and 5 both show examples where the model program can execute the scenario. In Figure 6 we compose the stack model with a scenario that executes `Push(1)` followed by `Pop(),0`. This is forbidden, because pop should only return the value that was most recently pushed. As expected, we see that the product only contains the push action because it is unable to synchronize on the pop action, which is not enabled in the model. The product does not reach an accepting state, which shows that the model does not allow this scenario.

This technique can be use to check a model program for any property that can be expressed by a finite state machine, including any temporal logic formula. Exploration with composition is similar to model checking, and is a powerful complement to the state-based safety and liveness analyses described earlier.

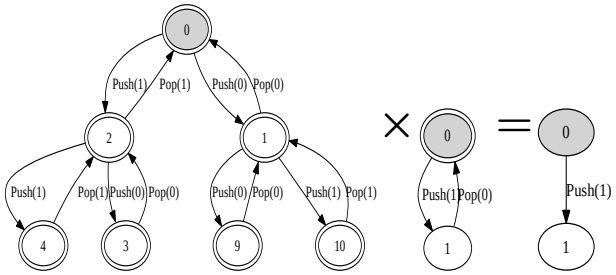

Figure 6: *Composition with a forbidden scenario cannot reach an accepting state.*

Conclusions

Model-based testing can encourage different approaches to testing. It encourages on-the-fly testing --- but in general, on-the-fly test runs are not reproducible, due to nondeterminism. It suggests extending testing to noninvasive monitoring or *run time verification* --- if the test harness supports observable actions, the test runner can check log files or monitor network traffic for conformance violations.

The most intruiging prospect might be better integration of design and analysis with testing. Exploration with composition is like model checking; it can can check for safety, liveness, and temporal properties. And, the *same models* are used for these analyses as for automated testing. Moreover, the models are written in the *same language* as the implementation, which could make them accessible to developers and test engineers, not just formal methods experts.

Model-based testing has been used on large projects in industry, but only *post-hoc*. Test engineers were given informal documentation and an implementation to test, and then reverse-engineered the models [Grieskamp08]. A more rational workflow might be to write the model *before* writing the implementation, analyze and tweak the design, then implement and test.

Related work

The techniques described in this paper can be expressed in any programming language. More detailed explanations and examples, using the NModel framework for C# [NModel11], appear in [Jacky08]. Another view of model-based testing appears in [Utting07]. Model checking is discussed in [Peled01].

References

[ABP11] Alternating Bit Protocol, Wikpedia, accessed June 2011. http://en.wikipedia.org/wiki/Alternating_bit_protocol

[Grieskamp08] W. Grieskamp, N. Kicillof, D. MacDonald, A. Nandan, K. Stobie, and F.L. Wurden. Model-based quality assurance of Windows protocol documentation. In: *ICST*, pages 502-506. IEEE Computer Society, 2008.

[Jacky08] Jonathan Jacky, Margus Veanes, Colin Campbell, and Wolfram Schulte. *Model-Based Software Testing and Analysis with C#*, Cambridge University Press, 2008.

[NModel11] NModel software, accessed June 2011. http://nmodel.codeplex.com/

[Peled01] Doron Peled. *Software Reliability Methods*, Springer, 2001.

[PyModel11] PyModel software, accessed June 2011. http://staff.washington.edu/jon/pymodel/www/

[Utting07] Mark Utting and Bruno Legeard. *Practical Model-Based Testing: a Tools Approach*, Morgan-Kaufmann, 2007.