Using Markov Chain Usage Models to Test Complex Systems

S. J. Prowell
The University of Tennessee
sprowell@cs.utk.edu

Abstract

Model-based testing using Markov chain usage models provides a powerful way to address testing concerns. Unfortunately, the use of Markov chain usage models on systems which have multiple streams of control, or which have many modeless dialogs, has required approaches which limit automated testing (strong abstractions) or make models difficult to analyze (notations hiding a state explosion). This paper presents a new approach which relies on applying concurrency operators to the test cases generated from simple Markov chain usage models to create sophisticated test cases. This approach leverages existing tools and notations.

1. Motivation

Effectively testing a system requires that one establish what to test (the selection criterion) and how much to test (the stopping criterion). Model-based testing using Markov chain usage models provides answers to these questions, and is a mature practice which has been used successfully on industrial projects [16, 3, 15]. The ability to analyze the Markov chain as a stochastic source provides help in choosing transition probabilities [14], determining when to stop testing [13, 8], and interpreting the results of testing [10, 9]. Computations for Markov chain usage models are documented [6], there is a standard notation for usage models [5], and tool support exists [7].

More recently, Markov chain usage models have been applied to networking components where there are multiple streams of inputs, and to large image processing systems with many concurrent modeless dialogs. Current approaches to the generation of usage models have relied on inventing strong abstractions (such as *none*, *some*, and *all*) which are difficult to translate into executable test cases for automated testing, or augmenting usage models with arbitrary variables and predicates, which makes them difficult to analyze using existing tools. An example of such a notation is EMML, which can succinctly describe very large usage models using usage variables and assertions about their values [4].

These practical concerns have motivated a new approach, in which test cases generated from Markov chain usage models are combined using concurrency operators similar to those used in concurrent regular expressions [2]. These allow one to use ordinary Markov chain usage models and existing tools to generate tests with multiple streams of control.

This paper presents a short introduction to Markov chain usage models, and then discusses the use of concurrency operators and their implementation in the Test Generation Language (TGL). A subset of which is already supported by tools [4] and in use on industrial problems.

2. Markov Chain Usage Models

Markov chain usage models are constructed to specify how a system to be tested is expected to be used once released into the field, and can be used to analyze expected use, generate tests, determine when to stop testing, and reason about the outcome of testing [16, 6, 10]. These models are directed graphs, in which states of use are connected by arcs labeled with usage events. A usage event is an external stimulus applied to the system under test, while different states of use are used to enable proper sequencing and relative likelihood of inputs. An example of a usage model for a simple telephone is shown in figure 1. In this model, a "use" (or a test case) of the phone begins in the [On Hook] state, and terminates when the [Exit] state is reached. Note that because of the manner in which the states are connected, a single test case consists of all events until the phone is first hung up.

The arcs of the usage model are annotated with their single-step transition probabilities. In this case uniform probabilities are chosen, but this need not be the case. In order to validate the usage model, one can analyze it to determine its long run properties, such as time spent in each state, overall percentage of a particular event, and probability of occurrence of an event in a single test case. See [7] for an example, and see [6] for more details.

Test cases can be generated from the model in a variety of ways.

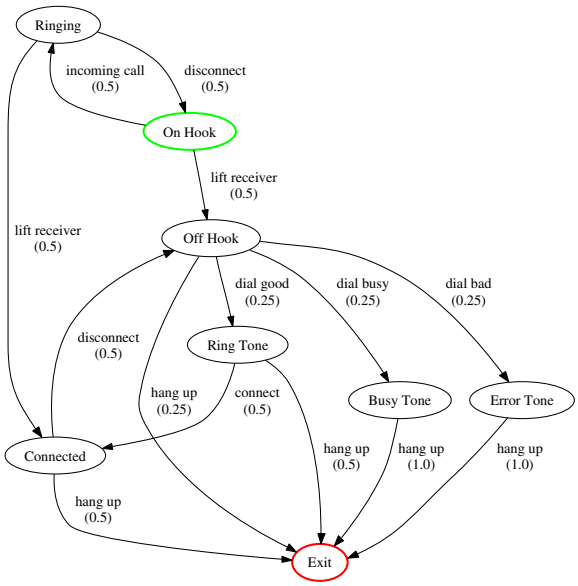

Figure 1. A Simple Telephone Model

- One can generate test cases by a random walk of the graph, starting in the source [On Hook] and ending in the sink [Exit]. This provides an unlimited source of test cases whose probability of generation is biased toward expected use and is suitable for statistical inference.

- Test cases can be generated which cover all arcs of the model in the fewest number of steps. This suite of test cases, called a *minimum coverage* test, can easily be generated using a postman algorithm. Running this collection of tests against the delivered system can help testers gain confidence that the delivered system matches the usage model structure.

- Weights can be associated with the arcs to generate test cases in order by weight. For example, if one believes the usage model to be representative of field use, one might be interested in seeing the ten most-likely uses. In this case the arc weights are the single-step probabilities, and the weight of a test case is the product of all its arc probabilities.

- Testers can generate test cases by hand to direct testing to particular features of the system. For example, the JUMBL toolkit provides a test case editor, and support for test cases written in XML [4, 7].

The usage model can be annotated such that test cases can be written as directly executable scripts. For example, one might annotate the model for automated testing of web applications [11] with C or Java language constructs for automated API testing [12]. One of the primary reasons to consider model-based testing is to enable automated testing of systems.

```
Phone 32: incoming call
Phone 65: incoming call
Phone 32: lift receiver
Phone 18: lift receiver
Phone 32: dial good
...
Phone 17: hang up
Phone 92: hang up
```

Figure 2. An Interleaved Test Case

3. Combining Tests

Consider again the usage model of figure 1. This model represents a single telephone, and test cases generated from it may be used to test a single telephone. Now consider a telephone switch, connected to one hundred such phones. In this case, one needs to combine many instances of this model, and distinguish among them. Since each phone can be in any of its seven states of use (ignoring the sink), such a model could require in 7^{100} states.

There is a better approach. One hundred test cases can be generated from the single phone model, and the events from these test cases randomly interleaved to for a new test case. Each of the original test cases is referred to as a *trajectory* in the new test case. Figure shows what such a test case might look like. In this case, the test will begin with all phones on hook, and end when all phones have been again been hung up.

Test Generation Language (TGL) is an XML [1] extension for combining test cases generated from Markov chain usage models. A full specification of TGL is beyond the scope of this document; see [4]. The root element of every TGL document is a `<TestGenerator>` element. Within the document two kinds of elements are present:

- Test sources, which may be collections of test cases (called *test records*), or references to usage models and appropriate automatic generation methods (random, minimum-coverage, or weighted).

- Test operators, which combine the test cases obtained from the test sources by concatenation, interleaving, or other methods.

The example given above of one hundred phones could be expressed in TGL as follows:

```
<?xml version="1.1"?>
```

```
<TestGenerator name="many_phones">
<!– Interleave 100 test cases. –>
<Tangle>
<GenTest num="100" model="phone"/>
</Tangle>
</TestGenerator>
```

This short TGL document contains one test source, `<GenTest>`. One hundred test cases are specified, the phone model is selected, and the default of random test case generation is used. One test operator, `<Tangle>`, is used to interleave these test cases to create a new test case. Each of the one hundred individual test cases is assigned a number, and becomes a single trajectory in the new test case. When automated test information is generated from this new test case, the trajectory number can be used to distinguish among the phones, as was shown in figure 2.

The `<Tangle>` operator takes an optional attribute, `channels`. This can be used to select the degree of multiplexing. For example, a better test case for the phone switch might allow many uses of a single phone.

```
<?xml version="1.1"?>
<TestGenerator name="many_phones">
<!– Interleave 1,000 test cases, with
one-hundred phones. –>
<Tangle channels="100">
<GenTest num="1000" model="phone"/>
</Tangle>
</TestGenerator>
```

By default the `<Tangle>` operator is assumed to have infinitely-many channels.

The idea of channels can be explained as follows. Suppose you have $n \geq 1$ test cases to interleave, and $1 \leq c \leq n$ channels. All usage events of test case m are appended, in order, to channel $m \mod c$, and this is done in the order the test cases are presented in the original collection. Figure 3shows an example for which $c = 4$ channels and $n = 7$ test cases. Because of this, the order of test cases in a single channel will be consistent with the order of test cases in the original collection. The interleaved test case is then generated as follows. Randomly select a channel, and remove the next usage event from the channel (from the top of the channel in figure 3). Append this usage event to the new test case, and repeat until all channels are empty.

Further improvements can be made. TGL allows specifying a range of values by enclosing the range in square brackets, as [10–100]. A random selection using a uniform distribution is made from the range. One could use such a range to specify the number of channels or the number of test cases.

It is likely that specific test setup and shutdown sequences are required for the switch, and usage models can be created to randomly generate such trajectories. These can be added to the start and end of the generated test case by "interleaving" the test cases with a single channel. Note that when there are more test cases than channels, the relative order of test cases enclosed in a `<Tangle>` operator is preserved.

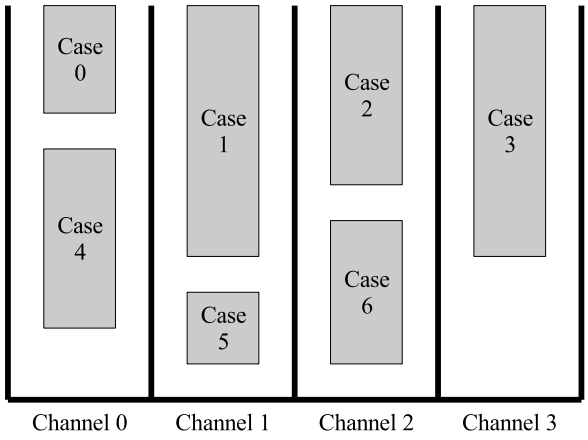

Figure 3. Allocating Test Cases to Channels

```
<?xml version="1.1"?>
<TestGenerator name="many_phones">
<Tangle channels="1">
<GenTest num="1" model="startup"/>
<Tangle channels="100">
<GenTest num="1000" model="phone"/>
</Tangle>
<GenTest num="1" model="shutdown"/>
</Tangle>
</TestGenerator>
```

This approach is common enough that a special operator, `<Cat>`, is defined for precisely this purpose.

TGL provides several operators for working with test sets. For example, `<Repeat>` can be used to indicate simple repetition (very much like a for loop) of the enclosed operators and test sources. The following fragment extracts a single test case from the ten most-likely test cases.

```
<Repeat num="[0-9]">
<Echo>Omitting a test case...</Echo>
<Discard>
<GenTest model="phone" method="weight"
id="1"/>
</Discard>
</Repeat>
<Echo>Generating a test case...</Echo>
<GenTest model="phone" method="weight"
id="1"/>
<Echo>Done!</Echo>
```

The above example illustrates several capabilities of TGL. The `<Echo>` operator is used to emit some text during interpretation of the TGL document. The `<Discard>` operator simply throws away the enclosed test cases (if any). In this case, the tests are generated in order by probability, starting with the highest-probability test case, and the `id` attribute is used to tie together the test generators, so that the interpreter knows not to re-start generation

with the highest-probability test case each time. By default `<GenTest>` generates a single test case. This short script works by randomly discarding from zero to nine test cases, and then generating a single test case.

The above example probably seems cumbersome, and TGL provides the much more succinct `<Random>` operator to randomly select from a collection of test cases. The following example also extracts a single test case from the ten most-likely.

```
<Random>
<GenTest model="phone" method="weight"
num="10"/>
</Random>
```

The following TGL document describes a fairly sophisticated test of the phone switch. The `<Repeat>` operator combined with a random number for the `<Tangle>` operator allows up to half the phones to be idle during each pass.

```
<?xml version="1.1"?>
<TestGenerator name="many_phones">
<Repeat num="[100-200]">
<Tangle channels="[50-100]">
<GenTest model="phone"
num="[100-1000]"/>
</Tangle>
</Repeat>
</TestGenerator>
```

4. Synchronization

A problem with the prior approach is that it does not account for synchronization among test cases. In our simple example of a phone switch, every "incoming call" event should match a "dial good" event on another phone. In other words, the test cases need to synchronize on these events. A model of how this can be done is provided by the synchronous composition operator [2]. Synchronous composition is not yet implemented in the JUMBL; this section describes a proposed implementation of this operation.

Test cases which require synchronization are combined with the `<Synchronize>` operator. This operator takes two or more collections of test cases, and attempts to combine them synchronously. This is done by placing each test collection in its own channel (thus if there are three enclosed collections, there are three channels), and drawing events from channels at random, as was done for the `<Tangle>` operator. As test cases are placed in the channels, the intersection of the usage events is taken, and the resulting set becomes the "synchronous" events. When a synchronous event appears on a channel that channel is blocked (the synchronous event cannot be drawn from the channel) until the same event appears on all other channels. At this point all synchronization events are drawn from the channels, in random order. Should a channel become empty while other channels are blocked, the synchronization fails, and this is reported to the user. Should all channels become blocked, the system is deadlocked, and this is also reported to the user.

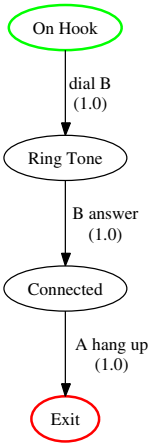

Figure 4. phoneA Model

The phone problem can now be addressed by splitting the phone usage model into two new models: phoneA (figure 4) and phoneB (figure 5), where phoneA represents the caller, and phoneB represents the callee. The shared events are "dial B", "B answer", and "A hang up". Note that these events, though the same in both models, could have different labels for automated testing; using the same name on an arc doesn't mean the same automated test information must be generated. These would then be combined with the following TGL snippet.

```
<Synchronize>
<GenTest model="phoneA"/>
<GenTest model="phoneB"/>
</Synchronize>
```

While this may seem simplistic, remember that it can now be inserted into a `<Repeat>` operator, and the result combined with other synchronized phone pairs using a `<Tangle>` operator. This can rapidly generate sophisticated test cases.

Since the automatic discovery of the synchronized events for test sets could potentially lead to unintuitive events, it sometimes makes sense to explicitly specify the synchronous events. This is done by including `<Event>` elements inside the `<Synchronize>` operator, as follows.

```
<Synchronize>
<Event>dialB</Event>
<Event>B answer</Event>
<Event>A hang up</Event>
<GenTest model="phoneA"/>
<GenTest model="phoneB"/>
</Synchronize>
```

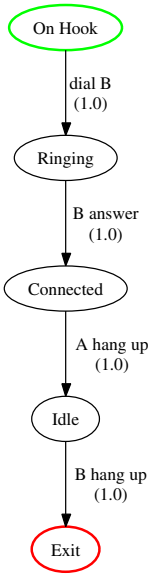

Figure 5. phoneB Model

The primary advantage of trying to synchronize test cases instead of creating a model which explicitly captures the synchronization is that one avoids the resulting state explosion. Further, this approach can use existing test records, which may have been generated manually using a test case editor. Since the usage models are not modified in any way, a statistical analysis of the usage models is still representative of the tests generated from each model. The primary disadvantage of synchronizing test cases is that, since the test cases can be generated randomly, synchronization may fail. This can typically be avoided by careful structuring of the individual models from which the tests are generated. Another disadvantage is that no overall statistics, such as the probability of deadlock, are provided. Model checkers may be used in this case, or one may simply generate many tests to obtain a sample estimate of the desired property.

5. Conclusions

TGL offers a way to combine ordinary existing Markov chain usage models in novel ways to generate tests for systems with multiple independent streams without a state explosion. The notation has proved to be easy to use in practice. Since the operators are applied to test cases, and not the usage models, generation of the new state space is never required, and analysis of the underlying models is preserved. Tool support for the interleave operation is already in place, and support for limited synchronization is planned.

References

[1] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML) 1.1. W3C recommendation, World-Wide Web Consortium, April 2004. http://www.w3.org/TR/2004/REC-xml11-20040204/.

[2] Vijay K. Garg and M. T. Ragunath. Concurrent regular expressions and their relationship to Petri nets. *Theoretical Computer Science*, 96(2):285–304, 1992.

[3] David P. Kelly and Robert Oshana. Improving software quality using statistical testing techniques. *Information and Software Technology*, 42(12):801–807, 2000.

[4] Software Quality Research Laboratory. JUMBL 4 user's guide. Technical report, The University of Tennessee, Knoxville, TN, November 2002.

[5] Stacy J. Prowell. TML: A description language for Markov chain usage models. *Information and Software Technology*, 42(12):835–844, September 2000.

[6] Stacy J. Prowell. Computations for Markov chain usage models. Computer Science Technical Report UT-CS-03-505, The University of Tennessee, Knoxville, TN, 2003.

[7] Stacy J. Prowell. JUMBL: A tool for model-based statistical testing. In *Proceedings of the 36th Annual Hawaii International Conference on System Sciences (HICSS'03)*. IEEE Computer Society Press, January 2003.

[8] Stacy J. Prowell. A stopping criterion for statistical testing. In *Proceedings of the 37th Hawaii International Conference on Systems Sciences (HICSS'37)*, Kona, HI, January 2004.

[9] Stacy J. Prowell and Jesse H. Poore. Computing system reliability using markov chain usage models. *Journal of Systems and Software*, 2004. to appear: special issue on the practical application of statistics in software engineering.

[10] Kirk D. Sayre. *Improved Techniques for Software Testing Based on Markov Chain Usage Models*. PhD thesis, The University of Tennessee, Knoxville, TN, December 1999.

[11] Kirk D. Sayre. Testing dynamic web applications with usage models. In *International Conference on Software Testing, Analysis and Review (STAR West)*. Software Quality Engineering, October 2003.

[12] Kirk D. Sayre. Automated API testing: A model-based approach. In *International Conference on Software Testing Analysis and Review (STAR East)*. Software Quality Engineering, May 2004.

[13] Kirk D. Sayre and Jesse H. Poore. Stopping criteria for statistical testing. *Information and Software Technology*, 42(12):851–857, September 2000.

[14] Gwendolyn H. Walton and Jesse H. Poore. Generating transition probabilities to support model-based software testing. *Software—Practice and Experience*, 30(10):1095–1106, August 2000.

[15] Gwendolyn H. Walton, Jesse H. Poore, and Carmen J. Trammell. Statistical testing of software based on a usage model. *Software—Practice and Experience*, 25(1):97–108, January

1995.

[16] James A. Whittaker and Michael G. Thomason. A Markov chain model for statistical software testing. *IEEE Transactions on Software Engineering*, 20(10):812–824, October 1994.