A TML Tutorial

S. J. Prowell
Software Quality Research Laboratory

October 27, 2000

About this Document

This document provides an overview of TML. More detailed information about TML can be found at the TML web site[1]. This document refers to TML 2.0, only. This document does not discuss the use of the JUMBL[2] or other utilities for compiling and manipulating TML.

Contents

[1] http://www.cs.utk.edu/sqrl/esp/tml.html
[2] http://www.cs.utk.edu/sqrl/esp/jumbl.html

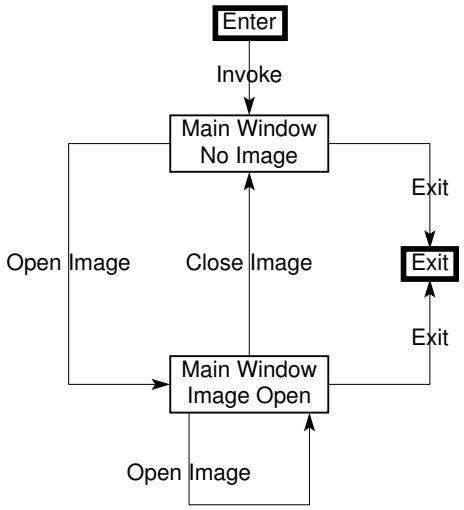

Figure 1: The “ImageViewer” Usage Model

1 What is TML?

The Model Language (TML) is a notation for describing Markov chain usage models. You can write models in TML, then use other tools to manipulate and analyze the models. TML supports the use of included models and constraints so you can describe your models in a higher-level fashion than with other tools.

2 How do I describe a model in TML?

Fig. 1 shows the graph of a simple usage model for an image viewer. Images can be opened for viewing or the user can exit. The nodes in the graph, such as Enter, represent states of use, and the edges, such as Invoke, represent usage events. In TML, the names of states are enclosed in square brackets and the names of usage events are enclosed in double quotation marks. For each state, the list of event / next state pairs is given following the state name. The model of Fig. 1 is equivalent to the following TML.

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
    "Invoke"      [Main Window, No Image]

[Main Window, No Image]
    "Open Image"  [Main Window, Image Open]
    "Exit"        [Exit]
```

```
[Main Window, Image Open]
    "Open Image"  [Main Window, Image Open]
    "Close Image" [Main Window, No Image]
    "Exit"        [Exit]
end
```

Models in TML are introduced by the `model` keyword and terminated by the `end` keyword. A model name must follow the `model` keyword, and may consist of upper- and lower-case letters, digits, and the underscore. Model names are case-sensitive. One model should be defined per file, and the file name should be the same as the model name, with the suffix "`.tml`" appended. Thus the above model name should be in a file named `ImageViewer.tml`.

TML can include C++ style comments. There are two forms of comments: single-line comments, which are introduced by `//` and terminated by the end of line; and multi-line comments which are introduced by `/*` and terminated by `*/`.

The general form for each state definition of a model is:

```
[state]
    "event 1"  [state 1]
    "event 2"  [state 2]
    ...
    "event n"  [state n]
```

All outgoing arcs from a given state must have distinct usage event names, and all state names for a given model must be unique.

Every model has two special states: [Enter] and [Exit]. The [Enter] state is the start state for the model; the [Exit] state is the final state of the model, and thus cannot have any outgoing arcs (else it would not be the *final* state). Since [Exit] cannot have any outgoing arcs, it need not be defined explicitly in the model.

TML ignores whitespace except for separating tokens; the above model could have been entered on a single line (except for the comment). The convention is to use a blank line between state definitions, indent arc definitions, and start each arc definition on a separate line.

3 How do I set probabilities on the arcs?

You can set probabilities for the arcs of a model by enclosing the probability value within `($` ... `$)` delimiters immediately before the event name. The `($` ... `$)` delimiters and enclosed text are called a *constraint*. The following is one possible set of probabilities for the previous model.

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
  ($1.00$) "Invoke"      [Main Window, No Image]
```

```
[Main Window, No Image]
  ($0.80$) "Open Image"  [Main Window, Image Open]
  ($0.20$) "Exit"        [Exit]

[Main Window, Image Open]
  ($0.50$) "Open Image"  [Main Window, Image Open]
  ($0.25$) "Close Image" [Main Window, No Image]
  ($0.25$) "Exit"        [Exit]
end
```

This explicitly sets the probability of each arc. You can instead use relative frequencies. For example, if no image is open, opening an image is about four times as likely as exiting. If an image is open, opening an image is twice as likely as exiting, and closing the image has the same likelihood as exiting. This can be expressed directly as:

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
  ($1$) "Invoke"      [Main Window, No Image]

[Main Window, No Image]
  ($4$) "Open Image"  [Main Window, Image Open]
  ($1$) "Exit"        [Exit]

[Main Window, Image Open]
  ($2$) "Open Image"  [Main Window, Image Open]
  ($1$) "Close Image" [Main Window, No Image]
  ($1$) "Exit"        [Exit]
end
```

Hint: You can use arithmetic expressions such as `($1-0.4*2$)`. Such expressions are evaluated in the usual algebraic order (the value of this expression is 0.2). You may also include arbitrary whitespace in values, as well as comments. The following is equivalent to the expression just given:

```
($ // Multiply 0.4 by 2, and subtract
   // the result from 1.
   1 -
     0.4 * 2
$)
```

4 What happens if I don't specify a value for an arc?

If you don't specify a value for an arc from state [X], one of four things happens:

1. All unspecified arcs are assumed to have value zero.

2. A default value is used for all unspecified arcs.

3. The total value for all specified arcs is subtracted from some target value, and the remainder is allocated evenly across the unspecified arcs.

4. The arc values for the unspecified arcs are set such that the state entropy is maximized.

The values of the outgoing arcs may then be scaled so that they sum to a given value (i.e., normalized). You control what is done using the `assume()`, `fill()`, `emax`, and `normalize()` directives. These must appear within a `($` ... `$)` pair before a state (to apply them to the outgoing arcs from that state) or model (to apply them to all arcs in the model).

The `assume()` directive instructs TML to use the expression specified within the parentheses as the value of any arc whose value is not explicitly given. The following is equivalent to the previous model.

```
// Usage model for a simple image viewer.
($ assume(1) // Assume a value of 1.
$)
model ImageViewer
[Enter]
        "Invoke"      [Main Window, No Image]

[Main Window, No Image]
  ($4$) "Open Image"  [Main Window, Image Open]
        "Exit"        [Exit]

[Main Window, Image Open]
  ($2$) "Open Image"  [Main Window, Image Open]
        "Close Image" [Main Window, No Image]
        "Exit"        [Exit]
end
```

Likewise, the `fill()` directive instructs TML to subtract the sum of all known arcs from the argument to `fill()`, and then distribute the difference evenly across all unspecified arcs. The following is equivalent to the previous model.

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
  ($1.0$) "Invoke"      [Main Window, No Image]

($ fill(1) $)
[Main Window, No Image]
  ($0.8$) "Open Image"  [Main Window, Image Open]
          "Exit"        [Exit]
```

```
($ fill(1) $)
[Main Window, Image Open]
  ($0.5$) "Open Image"  [Main Window, Image Open]
          "Close Image" [Main Window, No Image]
          "Exit"        [Exit]
end
```

The `emax` directive instructs TML to find values for the unknown arcs which maximize the single-step uncertainty for each state[3], and the `normalize()` directive instructs TML to scale the values of all arcs so that they sum to the given argument. The following is another version of the previous model, in which the arc values are scaled to sum to 100 (i.e., to be percentages):

```
// Usage model for a simple image viewer.
($ normalize(100)  // Arc values are percentages.
   emax            // Maximize entropy.
$)
model ImageViewer
[Enter]
       // This arc gets the value 100.
       "Invoke"      [Main Window, No Image]

[Main Window, No Image]
       // These arcs get the values 80 and 20.
  ($4$) "Open Image"  [Main Window, Image Open]
  ($1$) "Exit"        [Exit]

[Main Window, Image Open]
       // These arcs get approximately the
        // values 43.6, 34.6, and 21.8.
  ($2$) "Open Image"  [Main Window, Image Open]
       "Close Image" [Main Window, No Image]
  ($1$) "Exit"        [Exit]
end
```

If you do not specify any constraint before the model declaration, `($ emax $)` is assumed.

Hint: The most common approaches are to say `assume(1)` at the top of the file, then use values below one (such as 0.5 or 0.1) for events which are less likely than the average, and values above one (such as 2 or 10) for events which are more likely than the average. Alternately, if you don't know very much about usage, a conservative approach is to specify the relative frequencies for the arcs you do know, then use `emax` to choose the rest.

[3] If no arcs are specified, this is the uniform distribution.

5 Can I use variables?

You can declare and use variables within `($` ... `$)` pairs using a very simple syntax: *name* = *expression*. The expression may mention other variables and may use arithmetic operators. Variables are assigned and evaluated in the order they are encountered, and you may use variables as the arguments to directives.

Variables in TML are statically-scoped: variables declared in a `($` ... `$)` pair before the `model` keyword are available throughout the model (they have *model scope*); variables declared in a `($` ... `$)` pair before a state are available for all arcs from the state (they have *state scope*). The following is yet another way to specify the image viewer model.

```
// Usage model for a simple image viewer.
($ A=1 big=2 assume(A) $)
model ImageViewer
[Enter]
          "Invoke"      [Main Window, No Image]

[Main Window, No Image]
  ($4*A$) "Open Image"  [Main Window, Image Open]
          "Exit"        [Exit]

[Main Window, Image Open]
  ($big$) "Open Image"  [Main Window, Image Open]
          "Close Image" [Main Window, No Image]
          "Exit"        [Exit]
end
```

Hint: You can assign values to the variables `h` and `l`, and then use these in your model when you want to indicate an arc's probability is higher or lower than the average, respectively. If you decide to do this, be sure to `assume()` some intermediate value, such as one. For example, you might use:

```
($ avg=1 l=avg/2 h=avg*2 assume(avg) $)
```

6 Can I use more than one distribution?

Multiple distributions can be used on a model. Each distribution is distinguished by its *key*, which is an identifier consisting of letters, digits, and the underscore. To associate a key with a `($` ... `$)` pair, place the key followed by a colon `(:)` before the `($` ... `$)` pair, as in `my_key:($ emax $)`. If there is no key in front of a `($` ... `$)` pair, then the `($` ... `$)` pair is part of the *default* distribution, which is always present in a model.

The following expands the image viewer model to have three distributions. The default distribution is the same one used previously. The uniform distribution assigns equal probabilities to all arcs. The long distribution significantly increases the likelihood of opening an image.

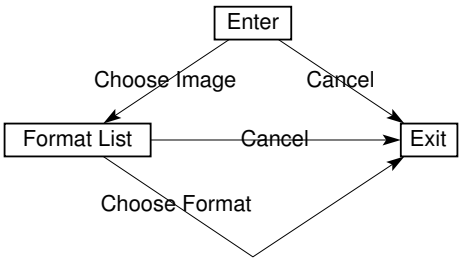

Figure 2: The Import Model

```
// Usage model for a simple image viewer.
        ($ assume(1) $)
uniform:($ assume(1) $)
long:   ($ assume(1) $)
model ImageViewer
[Enter]
        "Invoke"      [Main Window, No Image]

[Main Window, No Image]
  long:($8$)
  ($4$) "Open Image"  [Main Window, Image Open]
        "Exit"        [Exit]

[Main Window, Image Open]
  long:($4$)
  ($2$) "Open Image"  [Main Window, Image Open]
        "Close Image" [Main Window, No Image]
        "Exit"        [Exit]
end
```

Even though all three distributions use the same declaration at the model level, they use different values on the arcs.

7 Can I use one model in another?

You can include one model in another model by reference in much the same way subroutines are used in programming languages. This allows you to break a complicated model up into simpler models. For example, suppose the simple image viewer is extended to allow importing images of other formats. Importing an image is a two-step process. First, the image is selected. Second, the image type is specified. The user can cancel the operation at either step, and returns to the previous state. A model for import is given in Fig. 2.

Included models in TML are just other models, specified using the `model` keyword as before. As with any model, there must be an [Enter] state and an [Exit] state. The following is the TML for the import model.

```
// The image import model.
model Import
[Enter]
    "Cancel"        [Exit]
    "Choose Image"  [Format List]

[Format List]
    "Cancel"        [Exit]
    "Choose Format" [Exit]
end
```

This model needs to be inserted in two places in the original model. One can import an image whether or not there is currently an image open, so the import model should be included from two states: [Main Window, No Image] and [Main Window, Image Open]. To reference an included model, simply give a usage event, the included model name, and the state to which to go on exit from the included model. The import model is added at [Main Window, Image Open] as follows.

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
    "Invoke"       [Main Window, No Image]

[Main Window, No Image]
    "Open Image"   [Main Window, Image Open]
    "Exit"         [Exit]

[Main Window, Image Open]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
                       [Main Window, Image Open]
    "Close Image"  [Main Window, No Image]
    "Exit"         [Exit]
end
```

Hint: TML locates models using the assumption that the model name is the same as the file name. If you get errors indicating that a model could not be found, make sure the included model has been compiled, has the correct name, and is in either the current directory or the search path[4].

[4] See the documentation for the tools you are using to find out how model are located.

8 How can I go to different states based on what happens in an included model?

In the last section the import model must also be inserted at the [Main Window, No Image] state. If the user exits the import model by choosing Cancel, then the next state should be [Main Window, No Image]. If the user exits the import model by choosing an image format, then the next state should be [Main Window, Image Open].

The appropriate next state can be specified using the `select` keyword, then giving event / next state pairs, and then using the `end` keyword. The import model is added at [Main Window, No Image] as follows:

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
    "Invoke"       [Main Window, No Image]

[Main Window, No Image]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
       select
         "Cancel"        [Main Window, No Image]
         "Choose Format" [Main Window, Image Open]
       end
    "Exit"         [Exit]

[Main Window, Image Open]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
                     [Main Window, Image Open]
    "Close Image"  [Main Window, No Image]
    "Exit"         [Exit]
end
```

You can only use the last event of the included model in a `select` ... `end` block.

When you use a selector, you must include every arc label which exits the included model. For example, if “Cancel” were omitted from the above selector, the selector would be incorrect. In some cases you only want to trap a few exit events, and then have all other exit events go to the same state. This can be done with a *default* selector. To do this, use the keyword `default` followed by the state name. The following is equivalent to the above model:

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
    "Invoke"       [Main Window, No Image]
```

```
[Main Window, No Image]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
      select
        "Choose Format" [Main Window, Image Open]
         default          [Main Window, No Image]
      end
    "Exit"         [Exit]

[Main Window, Image Open]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
                     [Main Window, Image Open]
    "Close Image"  [Main Window, No Image]
    "Exit"         [Exit]
end
```

9 Can I start in a different state of the included model?

So far one always enters the included model at the model source. This behavior can be modified using an input *trajectory*. An input trajectory is a sequence of stimuli, applied starting at the source, and ending at some state in the model. The state where the new trajectory ends is the source for the purpose of the inclusion. Thus one has the general rule that *information is passed to and from included models in the form of stimuli*. This use of stimuli is intended to make model use independent of changes in the states (splitting and merging states, for instance).

To specify an input trajectory, append each stimulus to the model name with a period, and remember to enclose the stimulus name in quotation marks. Trajectories can be arbitrarily long, but *cannot* exit the included model. Suppose you want to allow changing the format of an image. The import model can be used for this, but it must be started as if the "Choose Image" stimulus has already happened. The change format option is added at [Main Window, Image Open] as follows:

```
// Usage model for a simple image viewer.
model ImageViewer
[Enter]
    "Invoke"       [Main Window, No Image]

[Main Window, No Image]
    "Open Image"   [Main Window, Image Open]
    "Import Image" Import
      select
        "Cancel"        [Main Window, No Image]
        "Choose Format" [Main Window, Image Open]
      end
```

```
    "Exit"          [Exit]

[Main Window, Image Open]
    "Open Image"    [Main Window, Image Open]
    "Import Image" Import
                     [Main Window, Image Open]
    "Change Format" Import."Choose Image"
                       [Main Window, Image Open]
    "Close Image"   [Main Window, No Image]
    "Exit"          [Exit]
end
```

Hint: Trajectories must always start from the model source. The following trajectories are *not* legal.

```
Import."Format List"    // Not a stimulus name.
Import."Choose Format"  // Must start at source.
```

10 Can I include test automation information?

Arbitrary data can be attached to a model, state, or arc. This is done by specifying the information either inside a `{$ ... $}` block, or following an `|$` up to and including the end of line. Such information is called a *label*, and they are placed immediately after the name of the thing to which they are attached.

Every label must have an associated key, similar to they keys used for multiple probability distributions. The keys consist of letters, numbers, and the underscore. To associate a key with a label, place the key followed by a colon (`:`) in front of the label. The following uses labels to attach instructions to human testers to the model, to the [Exit] state, and to each arc in the model:

```
// The image import model.
model Import
i2t:|$When starting a test, make sure the
    |$select image is displayed to choose
    |$an image.
[Enter]
    "Cancel"
    i2t:|$Click the Cancel button on the
        |$select image dialog.
    [Exit]

    "Choose Image"
    i2t:|$Click on an image, then click
        |$the Select button on the
        |$select image dialog.
```

```
    [Format List]

[Format List]
    "Cancel"
    i2t:|$Click the Cancel button on the
        |$choose format dialog.
    [Exit]

    "Choose Format"
    i2t:|$Click on a format, then click
        |$the Select button on the
        |$choose format dialog.
    [Exit]

[Exit]
i2t:|$Verify that the chosen image is
    |$imported in the chosen format, and
    |$the import dialog is dismissed.
end
```

If a test case were generated from the model, one could write the `i2t` labels as each model, state, or arc were encountered, giving a script which could be read by a human tester to execute the test.

Instead of simple text for human readers, one could attach information to be read by automated testing tools. For example, one might use a C++ language API for each of the events:

```
// The image import model.
model Import
cpp:|$#import "iostream.h"
    |$#import "import.h"
    |$// Test the import model.
    |$int main(void) {
    |$  cout << "Starting test of import."
    |$       << endl;
    |$  int step=1;
[Enter]
    "Cancel"
    cpp:|$  click_cancel();
        |$  step++;
    [Exit]

    "Choose Image"
    cpp:|$  int img=rand()*get_list_count();
        |$  select_item(img);
        |$  click_select();
```

```
        |$  step++;
    [Format List]

[Format List]
    "Cancel"
    cpp:|$  click_cancel();
        |$  step++;
    [Exit]

    "Choose Format"
    cpp:|$  int fmt=rand()*get_list_count();
        |$  select_item(fmt);
        |$  click_select();
        |$  step++;
    [Exit]

[Exit]
cpp:|$  cout << "Test complete after "
    |$       << step << " steps." << endl;
    |$  return 0;
    |$}
end
```

One can thus use labels to connect models to automated testing libraries and tools, and to both execute an event and check the results.

All the labels shown so far use the `|$` form, which includes the end of line marker. If you want to avoid this, use a `{$` ... `$}` pair. Only the enclosed characters are included. Any character can be included in a label. To use special characters, you can use character escapes. The following is a complete list of character escapes available in TML labels.

Escape	Meaning
`\n`	Newline (ASCII 10).
`\r`	Carriage return (ASCII 13).
`\t`	Tab (ASCII 4).
`\"`	Double quotation mark (useful in arc names).
`\]`	Literal closing square bracket (useful in state names).
`\\`	Literal backslash.
`\`*o*	ASCII *o*, where *o* is up to three octal digits. For example, `\015` is the same as `\r`.
`\x`*hh*	ASCII *hh*, where *hh* is two hexadecimal digits. For example, `\0d` is the same as `\r`.
`\u`*hhhh*	Unicode *hhhh*, where *hhhh* is four hexadecimal digits. For example, `\u000d` is the same as `\r`.

In addition to character escapes, tools which operate on labels may expand the

following variables on output (this is not, strictly-speaking, part of the TML language; consult the tool documentation for more information).

Variable	Expansion
`$M`	The current model name.
`$N`	The current node (state or model reference) name.
`$A`	The current arc name.
`$s`	The current step number in the test case.
`$t`	The current trajectory number in the test case.
`$$`	A literal dollar sign ($).
`$[`*n*`]`	A random integer x such that $0 \leq x \leq n$.

Note here that `$t` does not refer to input trajectories, but to multiple-trajectory test cases supported by the JUMBL.

11 Where do I go next?

You should read up on Markov chain usage models. Visit the Software Quality Research Laboratory publications page[5] for recent papers. There is on-line TML documentation which is much more comprehensive than this short tutorial[6]. The JUMBL is a tool which supports the use of TML[7].

[5] http://www.cs.utk.edu/sqrl/publications.html
[6] http://www.cs.utk.edu/sqrl/esp/tml/index.html
[7] http://www.cs.utk.edu/sqrl/esp/jumbl.html