JUMBL 4 User's Guide

Software Quality Research Laboratory

14th November 2002

This document is Copyright © 2002 by the Software Quality Research Laboratory. All trademarks and servicemarks herein mentioned are the property of their respective owners.

```
$Revision: 1.13 $
$Date: 2002/11/15 03:24:34 $
```

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Welcome to the Java Usage Model Builder Library (JUMBL), a collection of tools to support statistical testing of software based on Markov chain usage models, a form of software testing that has gained wide acceptance in industry. The JUMBL lets test engineers

- construct usage models from components using numerical constraints,
- generate tests in various ways,
- convert tests to executable data to support test execution, and
- assess testing by providing test measures including expected system reliability in the field.

There is considerable literature describing statistical testing. Consult the references for more information.

1.1 License

The JUMBL is developed and distributed by the Experimentation, Simulation, and Prototyping (ESP) Project of the Software Quality Research Laboratory (SQRL) under a special license agreement. Contact the SQRL for more information: help@sqrl.cs.utk.edu.

1.2 Support

The JUMBL provides on-line help for all commands. Additionally, the SQRL maintains a web site with information about the JUMBL and resources such as this user's guide, tutorials, FAQs, and information about future versions. Visit

```
http://www.cs.utk.edu/sqrl/esp/index.html
```

for more information.

There is a mailing list for users of the JUMBL. Visit

```
http://lists.cs.utk.edu/mailman/listinfo/jumbl
```

to subscribe or view the archive. General support questions should be directed to help@sqrl.cs.utk.edu.

1.3 About this Document

This document describes the use of the JUMBL, version 4. After installation, you can type "jumbl About" and press Enter to see what version of the JUMBL you are using.

Chapter 2 covers installation and configuration of the JUMBL and how to get on-line help. Chapter 3 describes the statistical testing process and how the JUMBL supports each phase. Subsequent chapters describe use of the JUMBL in each phase. Finally, there are appendices providing references to the MML, TCML, and EMML file formats.

The JUMBL provides a command-line interface, so it is necessary to show interaction at the command line. The prompt is shown as a dollar sign ($). Lines to be typed by the user are underlined, and lines printed by the computer are shown in regular code font. For example:

```
$ jumbl GenTest simple.tml
Creating new test record: simple.str
Solving: model simple.
Generating tests: model simple.
Generated test: simple_1
```

Chapter 2

The JUMBL

The JUMBL can be used on any platform which supports a Java 1.4 or later virtual machine, and the file formats are platform independent, making the JUMBL very flexible. Special support is available for Windows, UNIX, and Linux installations, and those are described in this document.

Prior to installing the JUMBL on any platform, a Java 1.4 or later virtual machine must be installed. The JUMBL is usually tested with two different virtual machines: one from Sun Microsystems, and one from IBM. Unfortunately, at the time of writing IBM does not have a Java 1.4 virtual machine ready. When it becomes available, the JUMBL will be tested with it. These virtual machines can be obtained free of charge from the following web sites:

Sun: `http://java.sun.com/`
IBM: `http://www.ibm.com/developer/java/`

Follow the instructions provided with the Java distribution to install the appropriate virtual machine for your platform.

2.1 Installing the JUMBL Under Windows

The JUMBL for Windows machines (Windows 95 / 98 / Me / NT / 2000 / XP) consists of two files: `jumbl_rt.jar` and `jumbl.exe`. These files should be placed in a directory in the path, such as `C:\WINDOWS` or `C:\WINNT`. Once this is done, the JUMBL is installed.

If you wish, you may install the JUMBL in a different location. To do this, put the two files in a directory and then make sure that directory is in the `PATH`. To view the `PATH`, open the Control Panel, open the System item, change to the Advanced tab and click Environment Variables. Make sure the directory in which the JUMBL is installed is in the PATH (path elements are separated by semicolons). Next, create a new environment variable `JUMBLPATH` and set it to the directory in which you installed the JUMBL. If this is not done, the jar file might not be found (or worse, the wrong jar file might be found).

To test your installation of the JUMBL, open a command line window, type “jumbl” at the prompt, and press Enter. You should see output such as the following:

```
$ jumbl
JUMBL Wrapper Options:

  --mem=[num] Increase the memory limit for the virtual machine to
              [num] megabytes.

  --time      Time the execution of the specified action.

JUMBL: Specify command.
The following commands are understood:
     About
     Analyze
     Check
     CraftTest
     Flatten
     GenTest
     ManageTest
     Options
     Prune
     RecordResults
     Write

This software is Copyright (c) 2002 by the Software Quality Research
Laboratory, all rights reserved. Visit http://www.cs.utk.edu/sqrl/
```

If you see the above message, then all is well. For some versions of the JUMBL you will additionally see a special collection of switches which are useful for debugging installation problems.

If the JUMBL cannot locate the `jumbl_rt.jar` file, you may see the following message:

```
ERROR:
Unable to locate the runtime JAR file for the JUMBL.
This is the file jumbl_rt.jar, and it should be in
one of the following directories:
C:\WINDOWS, C:\sqrl, C:\jumbl, C:\WINNT
```

If you see this message you should make sure that the `jumbl_rt.jar` file is located in `C:\WINDOWS` or `C:\WINNT`, or that it is located in the directory pointed to by the environment variable `JUMBLPATH`.

If you receive a message that Java cannot be found, make sure you have correctly installed the Java distribution. It may be necessary to add the `bin` directory under your Java installation to the `PATH` environment variable (see above).

2.2 Installing the JUMBL Under UNIX and Linux

The JUMBL for UNIX and Linux machines (tested with Solaris 2.7 and 2.8, and with Red Hat 7.3) consists of two files: `jumbl_rt.jar` and `jumbl`. These files should be placed in a directory in the path, such as `/usr/local/bin`. Once this is done, the JUMBL is installed.

If you wish, you may install the JUMBL in a different location. To do this, put the two files in a directory and then make sure that directory is in the `PATH`. The `PATH` environment variable is typically set from a user's `.profile` or `.cshrc`, and may be set by the system administrator in a global `profile` or `cshrc`. Contact your system administrator for specifics.

To test your installation of the JUMBL, open a command line window, type "jumbl" at the prompt, and press Enter. You should see output such as the following:

```
$ jumbl
JUMBL Wrapper Options:

  --mem=[num] Increase the memory limit for the virtual machine to
              [num] megabytes.

  --time      Time the execution of the specified action.

JUMBL: Specify command.
The following commands are understood:
     About
     Analyze
     Check
     CraftTest
     Flatten
     GenTest
     ManageTest
     Options
     Prune
     RecordResults
     Write

This software is Copyright (c) 2002 by the Software Quality Research
Laboratory, all rights reserved. Visit http://www.cs.utk.edu/sqrl/
```

If you see the above message, then all is well. For some versions of the JUMBL you will additionally see a special collection of switches which are useful for debugging installation problems.

If the JUMBL cannot locate the `jumbl_rt.jar` file, you may see the following message:

```
ERROR:
Unable to locate the runtime JAR file for the JUMBL.
```

```
This is the file jumbl_rt.jar, and it should be in
the directory:
/usr/local/bin
```

If you see this message you should make sure that the `jumbl_rt.jar` file is located in the specified directory, as this is where the JUMBL will look for it. This is the directory where the `jumbl` script is located.

If you receive a message that Java cannot be found, make sure you have correctly installed the Java distribution. It is necessary to add the `bin` directory under your Java installation to the `PATH` environment variable (see above).

If you receive a message such as the following:

```
jumbl: No such file or directory
```

then you may need to modify the first line of the `jumbl` file. This line points to the `sh` (or `bash`) shell interpreter, which is typically located in `/bin`. The first line of the file specifies the location. For example, if the `sh` executable is located in `/usr/bin` on your system, you would change the first line to:

```
#!/usr/bin/sh
```

Once this is done you should be able to execute the JUMBL.

If you receive a message such as the following:

```
jumbl: Permission denied
```

you should set the permissions on the `jumbl` file. The file should have world read and execute permissions, which can be set by:

```
$ chmod 755 jumbl
```

The jumbl_rt.jar should have world read permissions, which can be set by:

```
$ chmod 644 jumbl_rt.jar
```

Once this is done you should be able to execute the JUMBL.

2.3 Configuring the JUMBL

The JUMBL has a number of configuration options, which it stores in a configuration file. If this file is not present, you will receive a warning when you use any JUMBL command. To create the configuration file (and suppress the warning) you should run the JUMBL options editor:

```
$ jumbl Options
```

This will bring up a GUI displaying several tabs and a list of options on each tab. Do not edit anything just yet. Click Exit and you will be prompted to save your changes. Click Yes, and a configuration file will be created for you.

It will seldom be necessary to modify any configuration options (the default values should be sufficient for most purposes). If you want to temporarily set a configuration option, you can do so from the command line. Just include the setting on the command line in square brackets.[1] The new setting will be used for that command only. The configuration name is composed of the tab name, a period, and the option name, and is case-sensitive. For example, to temporarily suppress multiple arcs when writing GML, you would use:

```
$ jumbl write -tGML [GML.multiple=false] simple.tml
```

This has the same effect as unchecking the "multiple" option on the GML tab, running the command, and then setting the "multiple" option back to its prior setting.

2.4 Using On-line Help

All the information present in the JUMBL command-line guide is available from the JUMBL itself. To get a list of the JUMBL commands, just type "jumbl" and press Enter. To see help for a specific command, type "jumbl" followed by the command name (commands are case-insensitive) and either -h or –help and press Enter. For example:

```
$ jumbl check -h
Usage: Check [models...]

Check the structure of the specified models. This detects states
which cannot be reached from the source and states from which the
sink cannot be reached. It also reports some statistics about the
model.

  --gui .................Use a GUI to select the files to check and
                         also to display the results.
  -h or
  --help ................Describe this command.
  -O [path] or
  --object_path=[path] ...Specify the object search path.

This software is Copyright (c) 2002 by the Software Quality Research
Laboratory, all rights reserved. Visit http://www.cs.utk.edu/sqrl/
```

There is also some support for obtaining help when an error or warning is generated. A URL is provided which points to the SQRL web site. Because of time constraints, this

[1]It may be necessary to enclose the entire setting in quotation marks to protect it from some versions of UNIX shells, such as CSH / TCSH on Solaris: `"[GML.multiple=false]"`.

documentation is being developed Just In Time. If you encounter a link which reports that no information is available, contact the SQRL lab (see Section 1.2) to report this.

2.5 Interacting With the JUMBL

The JUMBL toolkit provides a collection of commands. To use a command you type "jumbl" at the prompt, followed by the command name. Command names are not case-sensitive, and you need only type the first few characters of the command name (but you must type enough to distinguish among commands) so the following three lines are identical.

```
$ jumbl Write
$ jumbl wriTE
$ jumbl w
```

After the command is specified, any arguments and switches are specified. Switches are processed first, no matter where they appear on the command line. Arguments are processed next, in the order they occur.

There are two kinds of switches. Long switches start with two dashes and are one or more words, such as `--suffix` or `--object-path`. If a long switch takes an argument, an equal sign follows the switch, and the argument value follows the equal sign with no intervening spaces, such as `--suffix=bat`. Short switches start with a single dash and consist of a single character, such as `-s` or `-O`. If a short switch takes an argument it may be given immediately after the switch, with or without intervening spaces, such as `-s bat`. The following two command lines perform the same task.

```
$ jumbl Write --suffix=foo --key=highw simple.tml
$ jumbl w simple.tml -s foo -khighw
```

Care has been taken to keep switches consistent across JUMBL commands. Some common switches are `--suffix` to specify an output file name suffix, `--key` to specify a distribution or label key, and `--type` to specify an output type.

Chapter 3

Overview

In statistical testing, one treats the software testing problem as a statistical experiment. A sample (test cases to execute) is taken, and performance on the sample is used to predict performance in the field. The JUMBL supports this process, from generating a sample to analyzing the results of testing.

3.1 The Statistical Testing Process

The statistical testing process consists of the following phases:

1. Usage Modeling, in which usage models are constructed to represent the population of all tests. The JUMBL allows users to use constraints to specify probabilities, and to construct models in parts, and combine the parts into large models. JUMBL commands relevant to this phase are `Write`, `Check`, `Flatten`, and `Prune`.

2. Model Analysis and Validation, in which models are analyzed to determine their properties. These properties are compared against known or assumed properties of use. The JUMBL generates a comprehensive report of model and use statistics. The JUMBL command relevant to this phase is `Analyze`.

3. Test Planning, in which test cases are generated and test automation is planned. The JUMBL allows users to generate tests in a number of ways, and provides general support for test automation. The JUMBL commands relevant to this phase are `GenTest`, `CraftTest`, `Write`, and `ManageTest`.

4. Testing, in which tests are executed against the system and the results are recorded. The JUMBL allows users to record the results of test execution directly in the test case files. The JUMBL command relevant to this phase is `RecordResults`.

5. Product and Process Measurement, in which the results of testing are analyzed and reliability measures are reported. The JUMBL generates a comprehensive report on the testing experience. The JUMBL command relevant to this phase is `Analyze`.

3.2 Summary of the JUMBL Commands

Usage Modeling

- `Write` converts from one format to others.
- `Check` verifies that a usage model has the proper structure.
- `Flatten` takes a model which references other models, and instantiates these referenced models to create a single, large model.
- `Prune` removes unreachable and trap states from a model.

Model Analysis and Validation

- `Analyze` generates a comprehensive report of model and use statistics.

Test Planning

- `GenTest` creates random test cases from a model; creates a collection of test cases which cover a model (visit all arcs) in the fewest steps and with the lowest cost; or creates a collection of tests cases based on their weight, and can generate the highest probability test cases (those expected to occur most frequently in use).
- `CraftTest` is a facility for creating test cases by hand, or for editing existing test cases.
- `Write` converts tests from one format to others.
- `ManageTest` allows working with test records.

Testing

- `RecordResults` is used to record the results of executing a test case in the test case file.

Product and Process Measurement

- `Analyze` generates a comprehensive test report including reliabilities and test sufficiency measures.

3.3 Planning for Testing

Statistical testing requires significant planning, especially if automated testing is desired. This section describes one test planning approach.

First, identify what is to be tested. It may be a complete system, some aspect of a system, or a subsystem. This identification allows construction of a test boundary, which may or may not be the same as the system boundary used for system specification. The *test boundary* is the list of interfaces controlled during the test. For each controlled interface it must be possible to generate all inputs (in order to perform the test in a repeatable manner) and observe all outputs (in order to verify correct behavior). If this cannot be done, the interface is not testable, and one must either re-design the test, re-design the interface, or use another means of insuring correctness, such as functional verification.

Second, identify the classes of users, uses, and environments of interest for testing. Because statistical testing is based on expected use, these overall factors which influence use are identified.

- User: Any entity outside the test boundary which can generate inputs to an interface controlled during test. Users may be human users, or other hardware and software systems. Classes of user might include experts, experienced users, and novices. Different users use the system in different ways.
- Use: The motivation for using the system. One might use a spreadsheet in one manner to fill out an expense report, and in a very different manner to simulate a chemical reaction. The reason why one is using the system alters the way in which the system is used.
- Environment: The platform on which the system will be deployed. For example, the JUMBL can be used on Windows machines as well as UNIX machines, with some differences in the installation and configuration.

Once the user, use, and environment classes are identified, one constructs combinations of the three which make sense for testing. For example, one may assume that only expert users are going to be using the spreadsheet for simulating a chemical reaction. Each combination of user, use, and environment classes constitutes one *stratum* for testing. For each identified stratum, the percentage of overall test effort and the method of testing must be described.

3.4 Markov Chain Usage Models

Markov chain usage models have a simple structure. They are deterministic finite-state machines with probabilities on the arcs. The sum of the probabilities of the outgoing arcs from any state must sum to one. States of a Markov chain usage model are sometimes called *states of use*. The arcs in the model are labeled with test events called *usage events*. The model thus represents the population of all tests.

Every Markov chain usage model has two special states: the *source* and the *sink*. The source state represents the initial state of the usage model, and the sink represents the final state of the usage model. There should be no outgoing arcs from the sink. A *test* may be defined as any path starting in the model source, then traversing arcs in the model until the model sink is reached. The source and sink should both be *verifiable*. That is, testers should be able to ascertain that the system to be tested is truly in the source (initial) or sink (final) state. A test cannot start until the system is correctly in the source state, and if the system does not correctly reach the sink state, the test fails.

Models can be constructed in a number of ways. One approach is *bounded enumeration*, in which one starts with the source state and considers every usage event which can be applied at that point (often only starting the software or turning on the system). Each usage event labels an outgoing arc. The arc terminates at a new state if one of the following is true:

- After the usage event, different usage events are possible than were before the usage event. For example, initially a system may have no open files and very few options are available. After opening a file, however, many editing options become possible.
- After the usage event, different usage events are likely than were before the usage event. If the probabilities of the events change after the event, then one has a new state. For example, after printing a document one has the same options, but is typically less likely to print again.
- It is convenient to have a new state. States in a usage model are, essentially, placeholders which allow usage events to be sequenced. It is perfectly acceptable to create "bookkeeping" states or other convenient states in the model. For example, one may be interested in what happens after opening a file from the local disk or from a web site, even thought the same usage events are possible and likely after opening the file. In this case, the "opened file" state may be split into "opened from disk" and "opened from web."

If none of these conditions holds, the arc terminates at a previously-visited state. For each new state created, repeat the process. Remember that one is exploring the *usage* states of the system, not the system's *behavioral* states.

One usage model may reference other usage models, allowing hierarchical modeling, composition of models, and the creation of model component libraries (see Section 4.2). There is special language associated with these. If model B is referenced from model A, then model B is a *referenced* model. It is useful to decide what to do next in model A based on what happened in model B. For this purpose one may use an exit *selector*, which determines the next state of A based on the last event which occurred in B. Finally, it may be useful to start in B at some state other than the source. For this purpose one uses an input *trajectory*. The use of trajectories and input selectors is described in Section 4.8 and also in the TML documentation available on the web at:

`http://www.cs.utk.edu/sqrl/esp/tml.html`

There is considerable documentation available about constructing and analyzing Markov chain usage models. See the references for more information.

Chapter 4

Constructing Usage Models

The JUMBL supports many different ways of constructing usage models, including writing or generating TML, generating MML, drawing the model in Graphlet, and building the model in a spreadsheet. These approaches are each considered in this chapter.

4.1 Working With Different Formats

The JUMBL's native format is Saved Model (SM) files. Certain operations, such as generating tests, will create or replace an existing SM file.

It is easy to convert from one format to another. The JUMBL automatically detects the format of input files, and the `Write` command can be used to write these files in any supported format. Table 4.1 lists the formats supported by the JUMBL.

The `Write` command converts one format to another. To use it, specify the target format name with `-t`. To obtain a list of the available output formats, use the `--list` switch. The following will convert all TML files in the current directory to the SM format (the default format).

```
$ jumbl Write *.tml
Writing: simple.tml as SM.
Writing: ync.tml as SM.
Writing: ync_test.tml as SM.
```

When converting files from one format to another, one must be careful not to overwrite the original source files. This can be avoided by using the `-s` or `--suffix` switch. The specified filename suffix is appended to the model name and to the filename prior to the file extension. The following will convert all the GML files in the current directory to TML files, giving the file names the suffix "foo" to distinguish them from other files. Thus a file called `test.gml` would become `testfoo.tml`.

```
$ jumbl Write *.gml -tTML -sfoo
Writing: simple.tml as GML.
```

```
Writing: ync.tml as GML.
Writing: ync_test.tml as GML.
```

You may develop your models in any format understood by the JUMBL. The suggested formats are TML, GML, CSV, or MML. Subsequent sections in this chapter discuss creating models in SM, TML, GML, and CSV format. For MML, see Appendix A. The final sections of the chapter discuss checking model structure and flattening models which reference other models.

4.2 SM

You cannot create models directly in SM; it is a special compressed format used internally by the JUMBL. You can, however, convert other formats to SM and convert SM to other formats.

When the JUMBL must locate a model as part of performing some command (like flattening a model, generating test cases, or analyzing a test record), it searches for an SM file. If an SM file cannot be found, the JUMBL will issue a warning or error. Compile the source file to an SM file to solve this problem.

The JUMBL locates SM files by searching the object path. This can be set in two ways:

- Set the configuration option ObjectManager.path.
- Use the `--object-path` or `-O` switches with any JUMBL command to specify the object search path for the duration of the command.

The object search path defaults to the current working directory, which is sufficient for most purposes. If you have a global repository of usage model components, you may want to set the object search path to include the repository. Multiple directories in the object path are separated by colons (:) on UNIX and Linux, and by semicolons (;) on Windows; this is the same as the PATH variables on these platforms.

For example, to specify that the JUMBL should first search the current directory, and then search a library, one might use the following on Windows:

```
[ObjectManager.path=.;C:\models]
```

Likewise, one might use the following on UNIX or Linux:

```
[ObjectManager.path=.:/usr/share/models]
```

See Section 2.3 for more information about setting configuration options.

Table 4.1: Model Formats Supported by the JUMBL

Format	JUMBL Name	Input	Output	Supported Information	Special
The Model Language	TML	Yes	Yes	All	
Saved Model	SM	Yes	Yes	All	Default format for JUMBL.
Model Markup Language	MML	Yes	Yes	All	MML is an XML extension.
Extended Model Markup Language	EMML	Yes	Yes	All except model references.	EMML is an XML extension.
DOT	DOT	No	Yes	All except constraints, labels, and trajectories.	DOT is the format used by the AT&T Graphviz tools.
Graph Modeling Langauge	GML	Yes	Yes	All except trajectories.	Only useful if a GML editor is available. Some GML editors may not preserve counts, constraints, and labels; Graphlet does.
Comma Separated Value	CSV	Yes	Yes	All except referenced models and labels.	Only useful if a spreadsheet supporting CSV is available, such as Excel or KSpread.
Hypertext Markup Language	HTML	No	Yes	All	Creates a hyperlinked web among referenced models.

```
($ assume(1) $)
model ync_test
[Enter] "start test" [No Project]

[No Project]
    "Open Project"  [Project, No Change]
    "Exit"          [Exit]
    "New Project"   [Project, Change]

[Project, No Change]
    "Close Project" [No Project]
    "Exit"          [Exit]
    "Work"          [Project, Change]

[Project, Change]
    "Close Project" ync
    select "Yes"    [No Project]
           "No"     [No Project]
           "Cancel" [Project, Change]
    end
    "Work"          [Project, Change]
    "Exit" ync
    select "Yes"    [Exit]
           "No"     [Exit]
           "Cancel" [Project, Change]
    end
    "Save"          [Project, No Change]
end
```

Figure 4.1: An Example Model

4.3 TML

By far the most common means to create usage models is to write them in The Model Language (TML). This is a very simple language created specially for describing Markov chain usage models, and supports hierarchical modeling, constraints, and automated test information. An example model is shown in Figure 4.1.

TML files are text, and can be edited in any text editor, such as Notepad and Wordpad on Windows machines. A TML file has the extension .tml, and should contain one model whose name should match the file name without the .tml extension. So the model Simple would be found in the file Simple.tml.

In TML, if there is no explicit source (specified with the "source" keyword), but there is a state named [Enter], then the state [Enter] becomes the source. Likewise, if there is no explicit sink (specified with the "sink" keyword), but there is a state named [Exit], then the state [Exit] becomes the sink.

Most JUMBL commands can be run directly on TML files, but because the SM format can be read much faster it is customary to first "compile" the TML using `Write`. For example, the model of Figure 4.1 should be stored in a file named `ync_test.tml`, and would be compiled to a SM file with:

```
$ jumbl Write ync_test.tml
Writing: ync_test.tml as SM.
```

The SM file could then be "decompiled" to TML as follows:

```
$ jumbl Write ync_test.sm -tTML -s_foo
Writing: ync_test.sm as TML.
```

This creates a file named `ync_test_foo.tml`. Note that the `-t` is used to specify the type of file to write, and the `-s` is used to prevent overwriting the original `ync_test.tml`.

4.4 GML

Graph Modeling Language (GML) is a language for expressing graphs, and is supported by several graphical tools, including the following:

- Graphlet is a graph layout and editing tool. This utility is used at SQRL, and is the most supported. Graphlet runs on Windows (except XP), UNIX, and Linux. Visit:

  http://www.infosun.fmi.uni-passau.de/Graphlet/

- The yed editor of yWorks is a commercial graph layout and editing tool. It is written in Java, and runs on any Java 1.2+ platform. Visit:

  http://www.yworks.de/

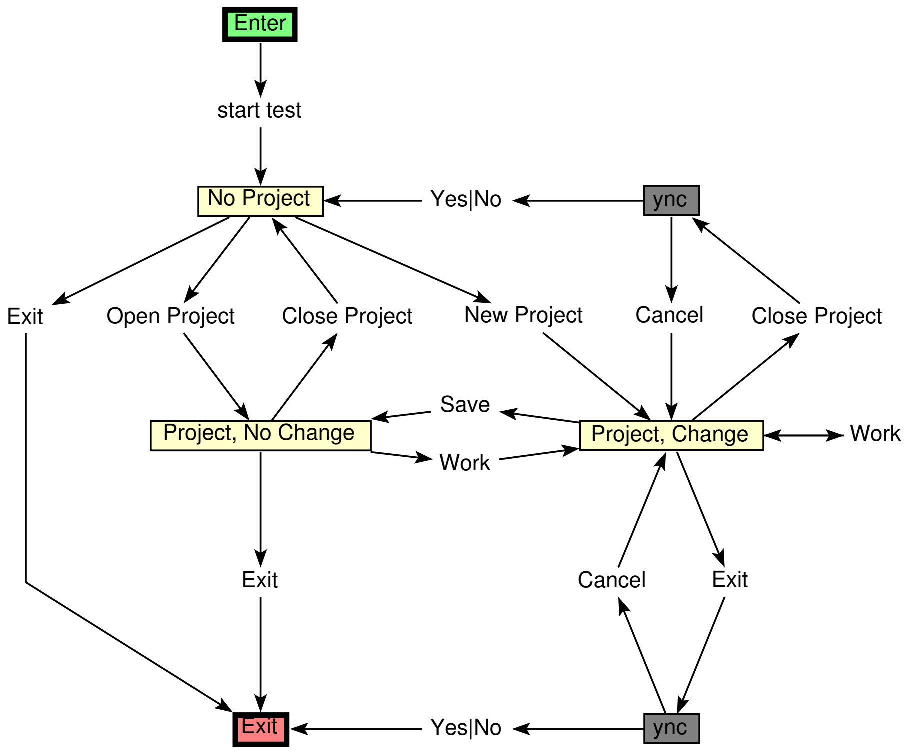


Figure 4.2: An Example Model

An example model graph is shown in Figure 4.2. When creating your model in a graph editor, you must observe the following rules so that JUMBL can correctly interpret what you create.

- **There must be at most one model source, and it must be green.** A node is considered green if its fill color at least 50% (128 decimal or 80 hex) green, and its red and blue components are below 25% (64 decimal or 40 hex).
- **There must be at most one model sink, and it must be red.** A node is considered red if its fill color at least 50% (128 decimal or 80 hex) red, and its green and blue components are below 25% (64 decimal or 40 hex).
- **Referenced models should be 50% dark.** A node is considered 50% dark if its fill color components are no more than 50% (128 decimal or 80 hex).
- **You cannot have self-loops ("mouse ears") on referenced models.** This leads to infinite selectors in TML.
- **Always draw arcs from the source to the target.** If you simply reverse the arrowheads (you can do this in some tools) the parser will not catch it. This may be fixed in a future version.
- **Unlabeled arcs are assumed to be labeled with the name of the target node.** An unlabeled arc to [Exit] is assumed to be labeled "Exit." To avoid this assumption, explicitly label the arc.
- **Input trajectories are not supported.** Trajectories are a little-used feature of TML, and are not currently supported for GML input to the JUMBL.
- **Indicate default selections by labeling the arc with an asterisk ("*") or the text "(default)," with the latter choice including the parentheses.**
- **If your graph is unnamed, then the base filename is used as the model name.** Some graph editors allow you to label your graph. If this is done, JUMBL will use the graph label as the model name.
- **One graph per file.** Some tools may allow you to construct and save multiple graphs in a single file. This is not recommended.
- **If you split arcs, the intermediate node's outline must be 75% bright.** Arcs may be split, meaning that the arc is replaced with a node which has the arc label, and two additional arcs. This improves automated graph layout in many tools. A node is considered 75% bright if all its outline (not fill) color components are at least 75% (192 decimal or C0 hex).
- **Separate multiple events on an edge with the vertical bar.** You can represent multiple arcs with a single edge by separating the choices with a vertical bar.
- **Remember: The conversion process is lossy in some graph editors.** If you convert from TML to GML and back, you may lose constraints and labels unless you are using Graphlet with multiple arcs.

Even though this may seem like a lot of rules, it is actually very easy to construct a model in a graph editor. Place the nodes, connect them with edges, label the edges and nodes, and you are essentially done. Color the source green, the sink red, and if you have any referenced models, make them dark gray.

There are options you can pass to `Write` which control how the GML output is generated. These may be set on the command line, or via the GML tab of the JUMBL `Options` editor. See Section 2.3.

- [GML.multiple=*boolean*]
  If true, one edge is written for every arc between two nodes. If false, only one edge is created and is labeled with multiple event names. Setting this to false prevents preserving labels and constraints in the output, but can make the graph much less cluttered. If you are trying to layout the graph for printing, it is good to set this option to false. By default this is true.

- [GML.split=*boolean*]
  If true, each edge is split and an intermediate node is inserted. If false, this is not done. Setting this to true makes the graph much larger (and thus makes its layout slower), but can significantly improve the look of the final graph. By default this is true.

- [GML.edgeLabels=*boolean*]
  If true, edge labels are written. If false, edge labels are suppressed. For large graphs, setting this to false and also setting GML.multiple to false will generate the state connectivity graph, giving a high-level view of the model structure. Note that information (edge labels) is lost if you set this to false. By default it is true.

The example model of Figure 4.2 might be stored in a file named `ync_test.gml`. This file could then be "compiled" to an SM file using `Write` as follows:

```
$ jumbl Write ync_test.gml
Writing: ync_test.gml as SM.
```

Alternately, this file could be converted to TML (for editing labels and constraints) using:

```
$ jumbl Write ync_test.gml -tTML
Writing: ync_test.gml as TML.
```

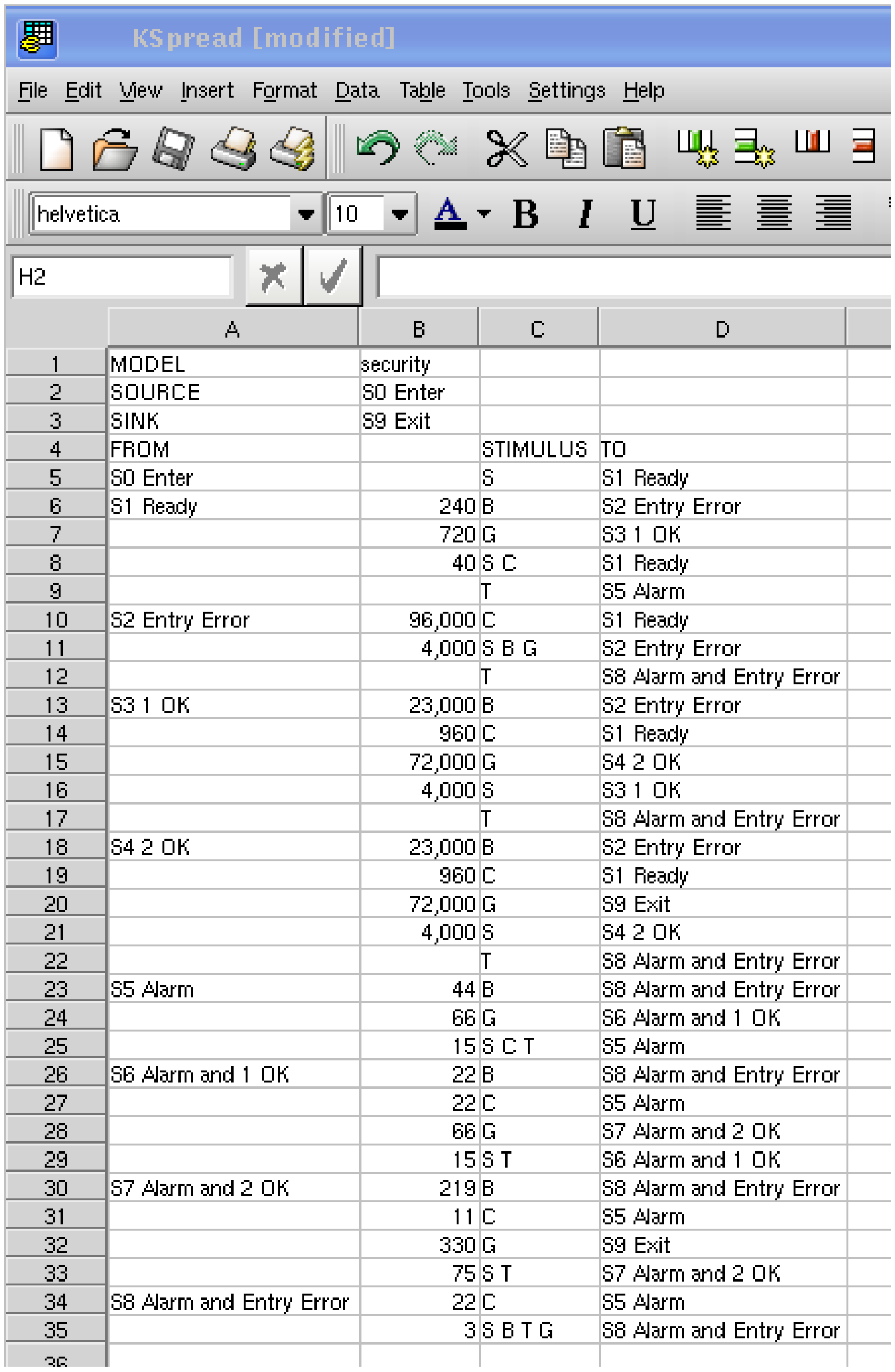
Note that this will overwrite any existing `ync_test.tml` file. Finally, the edited TML file could be converted back to GML using:

```
$ jumbl Write ync_test.tml -tGML
Writing: ync_test.tml as GML.
```

4.5 CSV

Comma-separated value (CSV) is a common interchange format for databases and spreadsheets. It is supported by most spreadsheets, including Excel on Windows, and KSpread on Linux. It is not very elaborate, and places some constraints on what can and cannot be present in state and arc labels. An example model is shown in Figure 4.3. When creating your model in a spreadsheet for export as CSV, you must observe the following rules so that JUMBL can correctly interpret what you create.

- **Only one model per spreadsheet.** At most one model may be specified in a CSV file.
- **Only model data in the spreadsheet.** JUMBL tries to interpret all data in the spreadsheet as model data. There is currently no provision for comments and non-model data.
- **Column A is significant.** The first column (column A in most spreadsheets) holds keywords and state names. These must go in the first column. Keywords are not case-sensitive, but state names are.
- **Optionally specify the model name.** If you want to specify the model name, put the keyword “model” in column A and the model name in column B. If no model name is specified, the JUMBL will use the file’s base name (the name without any path or extension).
- **Optionally specify the model source.** If you want to specify the model source, put the keyword “source” in column A and the source state name in column B. If no source is specified, none is assumed.
- **Optionally specify the model sink.** If you want to specify the model sink, put the keyword “sink” in column A and the sink state name in column B. If no sink is specified, none is assumed.
- **Give a header row.** Every model must have a header row, which specifies the contents of the columns. The header row must have the keyword “from” in column A, and must be placed after any model, source, or sink declarations. Two subsequent cells of the header row must contain the keywords “to” and “stimulus.” Other non-empty cells specify constraint names. A single empty cell may be present, which designates the corresponding column as the default constraints. The keywords “from,” “to,” and “stimulus” are not case-sensitive.
- **All subsequent rows of the spreadsheet are arcs.** Every row of the spreadsheet subsequent to the header row must specify an arc of the model. The from state name must go in the first cell of the row (the “from” column) unless it is the same from state as the prior arc, in which case the first cell may optionally be empty. The to state name must go in the “to” column. The stimulus label must go in the “stimulus” column. Constraint columns may hold numeric constraints for the arcs.



	A	B	C	D
1	MODEL	security		
2	SOURCE	S0 Enter		
3	SINK	S9 Exit		
4	FROM		STIMULUS	TO
5	S0 Enter		S	S1 Ready
6	S1 Ready	240	B	S2 Entry Error
7		720	G	S3 1 OK
8		40	S C	S1 Ready
9			T	S5 Alarm
10	S2 Entry Error	96,000	C	S1 Ready
11		4,000	S B G	S2 Entry Error
12			T	S8 Alarm and Entry Error
13	S3 1 OK	23,000	B	S2 Entry Error
14		960	C	S1 Ready
15		72,000	G	S4 2 OK
16		4,000	S	S3 1 OK
17			T	S8 Alarm and Entry Error
18	S4 2 OK	23,000	B	S2 Entry Error
19		960	C	S1 Ready
20		72,000	G	S9 Exit
21		4,000	S	S4 2 OK
22			T	S8 Alarm and Entry Error
23	S5 Alarm	44	B	S8 Alarm and Entry Error
24		66	G	S6 Alarm and 1 OK
25		15	S C T	S5 Alarm
26	S6 Alarm and 1 OK	22	B	S8 Alarm and Entry Error
27		22	C	S5 Alarm
28		66	G	S7 Alarm and 2 OK
29		15	S T	S6 Alarm and 1 OK
30	S7 Alarm and 2 OK	219	B	S8 Alarm and Entry Error
31		11	C	S5 Alarm
32		330	G	S9 Exit
33		75	S T	S7 Alarm and 2 OK
34	S8 Alarm and Entry Error	22	C	S5 Alarm
35		3	S B T G	S8 Alarm and Entry Error

Figure 4.3: An Example Model

- **Model references are not supported.** Models constructed in CSV may not reference other models.

The model of Figure 4.3 should be stored in a file named `ync_test.csv`. This file could then be "compiled" to an SM file using `Write` as follows:

```
$ jumbl Write ync_test.csv
Writing: ync_test.csv as SM.
```

Alternately, this file could be converted to TML (for editing labels) using:

```
$ jumbl Write ync_test.csv -tTML
Writing: ync_test.csv as TML.
```

Note that this will overwrite any existing `ync_test.tml` file. Finally, the TML file can be converted to CSV (resulting in loss of labels) using:

```
$ jumbl Write ync_test.tml -tCSV
Writing: ync_test.tml as CSV.
```

When the JUMBL writes a model as CSV, it first tries to flatten any model references by collapsing them (see Section 4.8). Thus the referenced models should be available as SM files. See Section 4.2 for information about how the JUMBL locates referenced models.

4.6 Checking Model Structure

A model's structure is correct if there is a single source, single sink, and for every state in the model there is a path from the source to the sink which visits the state. States for which there is no path from the source to the state are *unreachable*, and states for which there is no path from the state to the sink are *trap* states.

A model's structure can be checked using the `Check` command. For example, consider the model of Figure 4.4 and examine the italicized portion. State names are case-sensitive, so the state name shown does not match that referenced in the arcs. The model can be checked as follows:

```
$ jumbl Check bad.tml
Checking: model bad.
========================================
ERROR:  Unreachable node: [Project, No change]
ERROR:  Trap node: [Project, No Change]
----------------------------------------
          State count: 6
Model reference count: 2
            Arc count: 17
========================================
($ assume(1) $)
model bad
[Enter] "start test" [No Project]

[No Project]
    "Open Project"  [Project, No Change]
    "Exit"          [Exit]
    "New Project"   [Project, Change]

[Project, No change]
    "Close Project" [No Project]
    "Exit"          [Exit]
    "Work"          [Project, Change]

[Project, Change]
    "Close Project" ync
    select "Yes"    [No Project]
           "No"     [No Project]
           "Cancel" [Project, Change]
    end
    "Work"          [Project, Change]
    "Exit" ync
    select "Yes"    [Exit]
           "No"     [Exit]
           "Cancel" [Project, Change]
    end
    "Save"          [Project, No Change]
end
```

Figure 4.4: An Model With Errors

The JUMBL reports that the state [Project, No change] cannot be reached, and that the sink cannot be reached from state [Project, No Change]. Also, some overall model statistics are reported. There are six states in the model; the four explicit states and the additional implicit states [Exit] and [Project, No Change]. There are two model references, and 17 arcs. After correcting the error, only the statistics are reported by `Check`.

When using model references, `Check` also verifies input trajectories and selectors, and makes sure each referenced model can be loaded. `Check` also resolves all constraints and verifies that a state is not made a trap state (all outgoing arcs have probability zero) by the constraints.

4.7 Pruning Models

The `Prune` command removes any states which are unreachable from the source or from which the sink cannot be reached. To prune the ync_test model use:

```
$ jumbl Prune ync_test.sm
Pruning: model ync_test
```

If you want to see if there are any unreachable states before you prune, use `Check`:

```
$ jumbl Check ync_test.sm
Checking: model ync_test.
========================================
----------------------------------------
          State count: 7
Model reference count: 0
            Arc count: 17
========================================
```

4.8 Flattening Models

If you have built a model out of component models then you may wish to “flatten” your model. This will reduce it to just states and arcs, with all model references either removed or instantiated. This must be done prior to generating tests (see Chapter 6).

There are two ways to flatten a model:

- **Flatten by collapsing** converts every model reference into a normal state, with the selector determining the outgoing arcs. Referenced models do not have to be present for this to work.

- **Flatten by instantiating** replaces every model reference with the complete referenced model, and is the usual (and default) way in which one flattens a model. All referenced models must be present as SM files for this to work.

To flatten a model, use the `Flatten` command. By default, this command flattens by instantiating, meaning that all referenced models must be available as SM files prior to invoking the command (see Section 4.2).

Recall the model of Figure 4.1, and assume the referenced model ync is given by Figure 4.5. If `ync_test.tml` and `ync.tml` are available in the current directory, you could flatten ync_test by instantiation as follows:

```
($ assume(1) $)
model ync
[Enter]
    "Yes"    [Exit]
    "No"     [Exit]
    "Cancel" [Exit]
end
```

Figure 4.5: Referenced Model

```
$ jumbl Write ync.tml
Writing: ync.tml as SM.
$ jumbl Flatten ync_test.tml
Flattening: model ync_test.
```

This generates a file `ync_test.sm` which contains the flattened model. This can be converted to TML with:

```
$ jumbl Write -tTML ync_test.sm -s_flat
Writing: ync_test.sm as TML.
```

The `-s` avoids overwriting the original TML file. The result of this is shown in Figure 4.6. There are a few things to note about the flattened model.

1. Every reference to a model is separately instantiated. This is a consequence of the first-order Markov property and the use of exit selectors.

2. The states created by flattening are named with the referenced model name, a dot, and the state name. Ties are broken by suffixing an integer.

3. Constraints from the referenced models are preserved in the flattened model.

The example does not show an input trajectory, but if one were used there might be states which were unreachable in the flattened model. These would be states which were “skipped over” by the input trajectory, and should be removed from the flattened model.

To flatten by collapsing, use the `--collapse` switch with the `Flatten` command. For example:

```
//TML
/*
 * ync_test_flat
 * This file has been generated by TML.
 */
($assume(1)$)
model ync_test_flat
    source [Enter]
        "start test"        [No Project]

    sink [Exit]

    [No Project]
        "Exit"              [Exit]
        "New Project"       [Project, Change]
        "Open Project"      [Project, No Change]

    [Project, Change]
        "Close Project"     [ync.Enter]
        "Exit"              [ync.Enter_1]
        "Save"              [Project, No Change]
        "Work"              [Project, Change]

    [Project, No Change]
        "Close Project"     [No Project]
        "Exit"              [Exit]
        "Work"              [Project, Change]

    ($assume(1)$)
    [ync.Enter]
        "Cancel"            [Project, Change]
        "No"                [No Project]
        "Yes"               [No Project]

    ($assume(1)$)
    [ync.Enter_1]
        "Cancel"            [Project, Change]
        "No"                [Exit]
        "Yes"               [Exit]

end // of model ync_test_flat
```

Figure 4.6: Flattened Model

```
$ jumbl Flatten ync_test.tml --collapse
Flattening: model ync_test.
```

This creates `ync_test.sm`. Again, use `Write` to see the result, shown in Figure 4.7.

```
$ jumbl Write -tTML ync_test.sm -s_flat
Writing: ync_test.sm as TML.
```

```
//TML
/*
 * ync_test_flat
 * This file has been generated by TML.
 */
($assume(1)$)
model ync_test_flat
    source [Enter]
        "start test"        [No Project]

    sink [Exit]

    [No Project]
        "Exit"              [Exit]
        "New Project"       [Project, Change]
        "Open Project"      [Project, No Change]

    [Project, Change]
        "Close Project"     [ync_2]
        "Exit"              [ync]
        "Save"              [Project, No Change]
        "Work"              [Project, Change]

    [Project, No Change]
        "Close Project"     [No Project]
        "Exit"              [Exit]
        "Work"              [Project, Change]

    [ync]
        "Cancel"            [Project, Change]
        "No"                [Exit]
        "Yes"               [Exit]

    [ync_2]
        "Cancel"            [Project, Change]
        "No"                [No Project]
        "Yes"               [No Project]

end // of model ync_test_flat
```

Figure 4.7: Model Flattened by Collapsing

Chapter 5

Analyzing Usage Models

The analysis of a usage model reveals information about the time spent in each state, the number of times a stimulus will occur in testing, and how long it takes to run a test. The JUMBL creates a comprehensive analytical report of model and use statistics which can be used to validate model correctness, plan for testing, investigate characteristics of expected use, and compare different modeling approaches.

The same command, `Analyze`, is used for analyzing models and analyzing test records. While most of this chapter pertains to analyzing models, some parts are generalized and discuss analyzing test records, as well.

5.1 Using Distributions

A usage model may have one or more distributions associated with it. A *distribution* associates numbers with models, states, and / or arcs. The most common use of distributions is to specify probabilities for arcs in a usage model, but distributions can also specify costs and weights for test generation. Distributions are usually created by using constraints in TML; see the documentation on TML for more information.

There is always an unnamed distribution on a model, called the *default* distribution. This is the distribution which is used by the JUMBL by default for generating tests, analyzing models, etc. Other distributions are identified by a *key*, which is a sequence of alphanumerics and underscores, with no spaces.

A model may have any number of distributions; for example, the model of Figure 5.1 has two distributions. These distributions are defined by the statements at the top of the model. The first distribution has no key and is the default distribution. This distribution is defined by the statement `($ assume(1) $)`, which directs the JUMBL to assume a constraint of one for all arcs. The second distribution has the key `highw` and is defined by the line `highw:($ fill(100) h=80 $)`, which directs the JUMBL to make outgoing arcs sum to 100 and also defines the constant `h` for use later. In this case the default distribution makes all outgoing arcs from a state equally-likely, and the `highw` distribution makes the likelihood of the event “Work” 80% from states where it is possible. Note the italicized portion; here the constant `h` is used to modify the probability of an exit arc for the `highw` distribution.

```
($ assume(1) $)
highw:($ fill(100) h=80 $)
model ync_test
[Enter] "start test" [No Project]

[No Project]
    "Open Project"  [Project, No Change]
    "Exit"          [Exit]
    "New Project"   [Project, Change]

[Project, No Change]
    "Close Project" [No Project]
    "Exit"          [Exit]
  highw:($h$)
    "Work"          [Project, Change]

[Project, Change]
    "Close Project" ync
    select "Yes"    [No Project]
           "No"     [No Project]
           "Cancel" [Project, Change]
    end
  highw:($h$)
    "Work"          [Project, Change]
    "Exit" ync
    select "Yes"    [Exit]
           "No"     [Exit]
           "Cancel" [Project, Change]
    end
    "Save"          [Project, No Change]
end
```

Figure 5.1: An Example Model With Two Distributions

5.2 Analyzing a Model

The JUMBL can analyze a model written in any of the recognized input formats except EMML (EMML can be converted to another format, such as TML, using `Write`). It generally only makes sense to analyze models in the TML, SM, and MML formats, since those can contain constraints. (GML may also contain constraints, but there is currently no way to directly enter these constraints in the graph editor. The model must first be converted to TML or some other format.) To analyze a model, use the `Analyze` command (note the spelling):

```
$ jumbl Analyze ync_test.tml
Solving: model ync_test.
Analyze: model ync_test.
```

This command will produce an analysis report in HTML with the name `ync_test_ma.html`, which may be opened for viewing in a web browser, and is described in Section 5.3.

If no distribution is specified, the default distribution is analyzed. For models with more than one distribution, the distribution to analyze can be specified with `--key` or `-k`, and the `Analyze` command supports the `-s` and `--suffix` switches described in Chapter 4, with a default suffix of `_ma` for "model analysis." For example, to generate an analysis against the `highw` distribution, use:

```
$ jumbl Analyze --key=highw --suffix=highw ync_test.tml
Solving: model ync_test.
Analyze: model ync_test.
```

This will produce a file named `ync_testhighw.html`.

When a model contains model references, as does the model of Figure 5.1, these are treated as states for the purpose of analysis (this is similar to flattening by collapsing; see Section 4.8).

5.3 Interpreting Analytical Results

Analysis reports describe the statistical properties of the usage model as a stochastic process. This section gives an overview of the analysis report, and describes some of the most common statistics. A complete guide to the statistics produced by the JUMBL is available on the web:

```
http://www.cs.utk.edu/sqrl/esp/jumbl.html
```

The names of the analytical results are hyperlinked to the statistics guide on the web, for convenience.

The analysis report is divided into four sections, which provide increasing granularity:

- The **model** section reports statistics which relate to the model as a whole. These include the test case length and the size of the model.
- The **nodes** section reports statistics which relate to each node, including the expected number of times the node will be visited in a test case.
- The **stimuli** section reports statistics which relate to each stimulus (a unique event labeling an arc of the model). These statistics are presented since a single stimulus may label multiple arcs in a model, such as "work" does in the model of Figure 5.1. These statistics include the expected number of times the stimulus will occur in a test case.

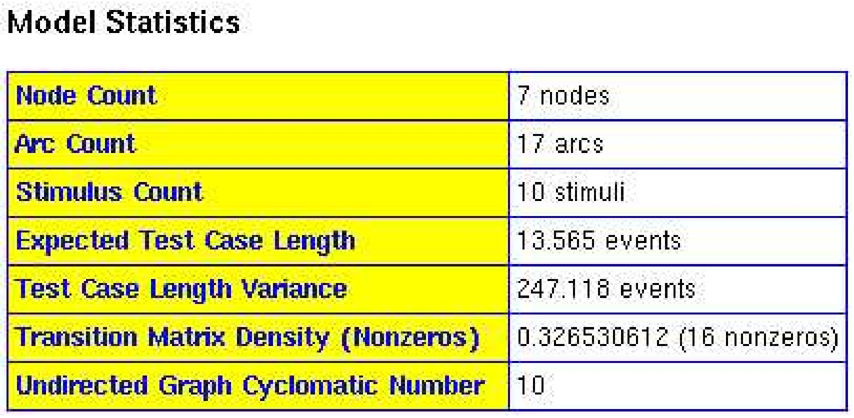
- The **arcs** section reports statistics which relate to each arc of the model. These include the probability of visiting a given arc in a test case, and the expected number of times the arc will occur in a test case.

These statistics can be used to validate the model, by showing that the model matches what is known or anticipated about system use.

Model Statistics

The model statistics include the number of nodes (both states and model references), number of stimuli (unique arc labels), and arcs. Of particular interest is the expected test case length. This is the average number of events which will be in a test case for a large number of test cases. See Figure 5.2 for an example.

For the model of Figure 5.1, there are five states (including the implicit state [Exit]) and two model references, for a total of seven nodes. There are 17 arcs, and ten stimuli. Under the `highw` distribution a test case can be expected to contain just over 13 events, but the variance on this is quite high (247), so individual test cases may be much longer.



Model Statistics

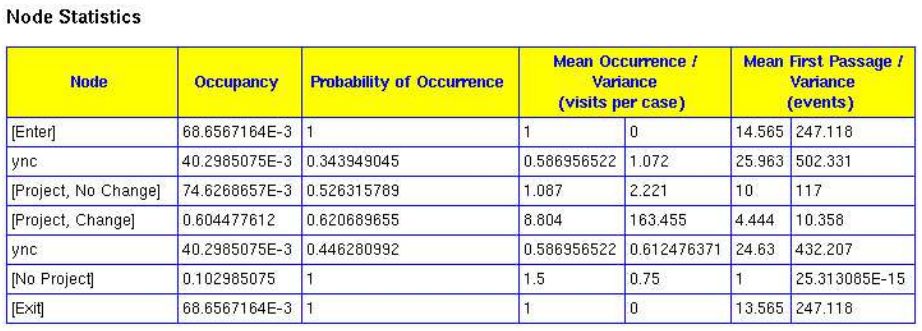
Node Count	7 nodes
Arc Count	17 arcs
Stimulus Count	10 stimuli
Expected Test Case Length	13.565 events
Test Case Length Variance	247.118 events
Transition Matrix Density (Nonzeros)	0.326530612 (16 nonzeros)
Undirected Graph Cyclomatic Number	10

Figure 5.2: Model Statistics

Node Statistics

The nodes of the model (states and model references) are listed down the left-hand column of the table, with each node's statistics listed across the row. Of particular interest is the node's occupancy, which is the fraction of time one will spend in the node (out of all nodes) for a large number of tests. Also of interest is the probability that a particular node occurs in a test case. See Figure 5.3 for an example.

For the model of Figure 5.1, the highest occupancy is 0.60 for [Project, Change]. This indicates that about 60% of the time spent in testing will be spent in this state. This fits the intuition that one is typically working on (changing) a project. The probability of occurrence of [Project, Change] is 0.62, so if one generates a large number of test cases [Project, Change] will only occur in about 62% of these. Since one would almost always want to edit a project, this indicates that arcs to [Project, Change] should have higher probabilities.



Node Statistics

Node	Occupancy	Probability of Occurrence	Mean Occurrence / Variance (visits per case)		Mean First Passage / Variance (events)	
[Enter]	68.6567164E-3	1	1	0	14.565	247.118
ync	40.2985075E-3	0.343949045	0.586956522	1.072	25.963	502.331
[Project, No Change]	74.6268657E-3	0.526315789	1.087	2.221	10	117
[Project, Change]	0.604477612	0.620689655	8.804	163.455	4.444	10.358
ync	40.2985075E-3	0.446280992	0.586956522	0.612476371	24.63	432.207
[No Project]	0.102985075	1	1.5	0.75	1	25.313085E-15
[Exit]	68.6567164E-3	1	1	0	13.565	247.118

Figure 5.3: Node Statistics

Stimulus Statistics

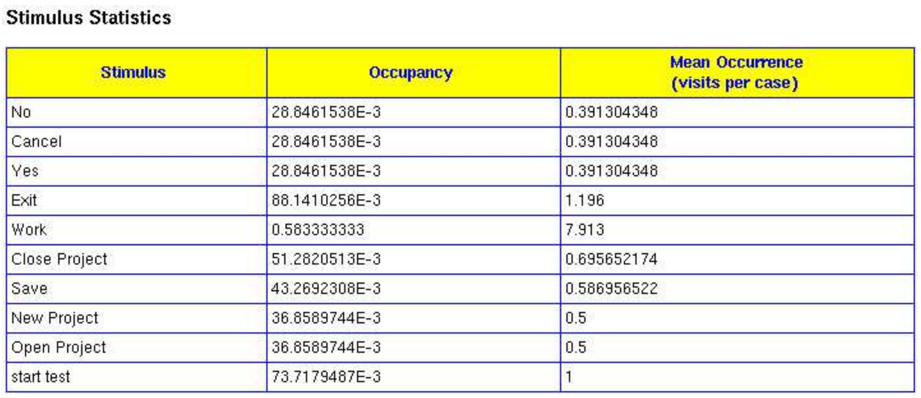
The stimuli of the model (unique arc labels) are listed down the left-hand column of the table, with each stimulus' statistics listed across the row. Of particular interest is the occupancy which is the fraction of time one spends on that stimulus (out of all stimuli) for a large number of tests. See Figure 5.4 for an example.

For the model of Figure 5.1, the highest occupancy is for "Work," at 0.58. This means that approximately 58% of the time one is doing work. This seems a bit low, and perhaps constraints should be modified to direct more of the model's flow to the work arcs. This can be done by increasing the "Work" arc probabilities, or increasing the probabilities of arcs which reach the nodes from which "Work" is possible.

Arc Statistics

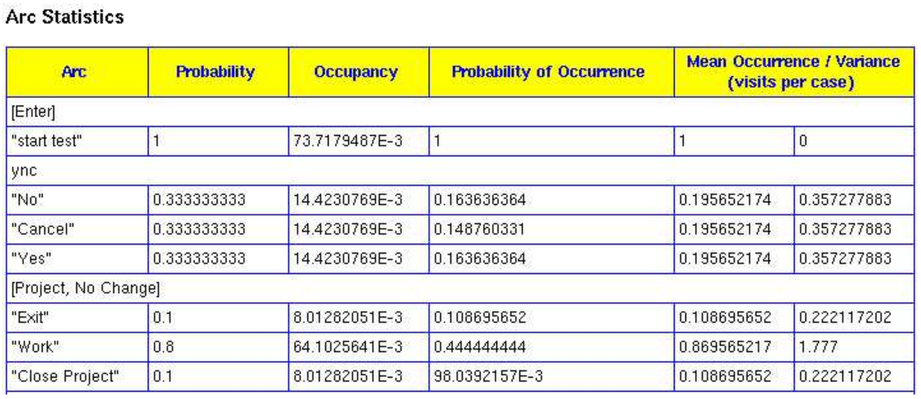
The entire model is reproduced in this section. For each arc of the model statistics are listed across the table. The probability of choosing a particular arc given the current node (called the "single-step" probabilities) is given next, as determined from the constraints. As with nodes and stimuli, the relative occupancy is given. Also, the probability that an arc is visited in a particular test case is given (not to be confused with the single-step probabilities). See Figure 5.5for an example.

For the model of Figure 5.1 one can compare the two "Work" arcs. Even though both have the same single-step probability (80%) they have different probabilities of occurrence in a test case. "Work" from the [Project, No Change] state occurs in 64% of test cases, whereas "Work" from the [Project, Change] state occurs in 52% of test cases. This is due to the fact that one must visit [Project, No Change] prior to [Project, Change].


Stimulus Statistics

Stimulus	Occupancy	Mean Occurrence (visits per case)
No	28.8461538E-3	0.391304348
Cancel	28.8461538E-3	0.391304348
Yes	28.8461538E-3	0.391304348
Exit	88.1410256E-3	1.196
Work	0.583333333	7.913
Close Project	51.2820513E-3	0.695652174
Save	43.2692308E-3	0.586956522
New Project	36.8589744E-3	0.5
Open Project	36.8589744E-3	0.5
start test	73.7179487E-3	1

Figure 5.4: Stimulus Statistics


Arc Statistics

Arc	Probability	Occupancy	Probability of Occurrence	Mean Occurrence / Variance (visits per case)	
[Enter]					
"start test"	1	73.7179487E-3	1	1	0
ync					
"No"	0.333333333	14.4230769E-3	0.163636364	0.195652174	0.357277883
"Cancel"	0.333333333	14.4230769E-3	0.148760331	0.195652174	0.357277883
"Yes"	0.333333333	14.4230769E-3	0.163636364	0.195652174	0.357277883
[Project, No Change]					
"Exit"	0.1	8.01282051E-3	0.108695652	0.108695652	0.222117202
"Work"	0.8	64.1025641E-3	0.444444444	0.869565217	1.777
"Close Project"	0.1	8.01282051E-3	98.0392157E-3	0.108695652	0.222117202

Figure 5.5: Arc Statistics

Sorting

The analytical results are naturally sorted in alphabetical order by state and arc name. The following options are available to control this behavior.

- [ModelAnalysis.sortBy=*type*]
  If *type* is name, then sort alphabetically by name (default behavior). If *type* is occupancy, then sort the results in order by long-run occupancy. If *type* is expectation, then sort the results in order by expected number of occurrences. Arcs are always sorted first by the source state using the defined sort order, then alphabetically by arc name.

- [ModelAnalysis.highestFirst=*bool*]
  If true, then sort numbers such that the highest appears first (this is the default). If false, sort numbers so that the lowest appears first. This does not affect the alphabetical sorting.

- [ModelAnalysis.nodeLimit=*n*]
  Limit the display of the nodes to just the first *n* nodes. If this is zero, display all nodes.

- [ModelAnalysis.stimulusLimit=*n*]
  Limit the display of the stimuli to just the first *n* stimuli. If this is zero, display all stimuli.

- [ModelAnalysis.arcLimit=*n*]
  Limit the display of the arcs to just the arcs of the first *n* nodes. If this is zero, display arcs for all nodes.

5.4 Using Analysis Engines

There are many ways to generate analytical results. One can use "analytical" methods, which typically involve solving a linear system and can take significant time and memory, but which produce very precise and accurate results. One can use "approximation" methods, which are typically iterative and generate more precise results the longer they are allowed to run. One can use "simulation" techniques which produce accurate results with lower precision. There are trade-offs among these approaches.

- For large models analytical methods may be very time-consuming and most results uninteresting. For example, if there are thousands nodes the individual node occupancies are probably very low, and iterative methods are probably a better approach.

- For models with very long test case lengths, simulation methods (which must generate many test cases) may be more costly than analytical techniques.

Because one can use model references, models should seldom grow to more than a few hundred states, and analytical methods are sufficient for these models. When models get larger, one is typically more interested in test case length, and this can be obtained quickly by iterative methods.

The JUMBL provides different analytical engines which obtain results using combinations of the above methods. An *analytical engine* computes some or all of the analytical results. Results which are not computed by a particular engine are omitted from the report. To see what analytical engines are available, use the `--list` switch.

5.5 Performance

For models which are not small (800 states or fewer) it may be necessary to increase the memory available to the Java virtual machine (JVM) in order to perform a model or test analysis. Maximum memory for the JVM can be specified using the `--mem` switch and giving a number of megabytes. To increase the maximum memory available to the JVM to 128 Mb, one would use:

```
$ jumbl Analyze ync_test.tml --mem=128
Analyzing: model ync_test.
```

The `--mem` switch increases the *maximum* amount of memory the JVM may use, which may be larger than the physical memory on a machine (in this case the JVM will use hard drive space, called *swap* space). The upper limit on the memory which may be requested via `--mem` is dependent on the operating system, specific JVM, and configuration of the computer's hardware and will differ from machine to machine.

If the `--mem` switch does not work for your installation, consult the documentation for your JVM for the correct switch. You can pass command line switches and arguments directly to the JVM via the JUMBL's `--J` switch.

If you typically work with very large models, you may want to permanently increase the memory available to the JVM. This can be done by setting the environment variable `JUMBLMAXMEM` to the desired number of megabytes. (This setting can always be overridden by the `--mem` switch.)

Chapter 6

Generating Tests

The JUMBL makes it easy to generate tests from a usage model. These tests can be written in many forms, supporting import into other tools (TCML) and automated execution of tests.

6.1 Working With Test Records

A test case is any path in a usage model which starts in the source, and ends in the sink. A test case is thus a sequence of usage events. Many test cases may be run during the course of testing, and these test cases may be generated in different ways (random, weighted, coverage), come from different distributions, come from different models, or come from combinations of these. The JUMBL organizes tests into Saved Test Records (STR files), herein called simply "test records."

A test record is an ordered sequence of test cases. When new test cases are generated, they are automatically added to a test record, and test records can be analyzed to determine expected reliability. Test cases stored in a test record can be exported for automated testing.

One works with test records in the JUMBL via the `ManageTest` command. The `ManageTest` command is then followed by a test management command. The following commands are currently supported:

- `Add` is used to add test cases to a test record. The test cases are added at the end of the test record. This command can be used to merge test records and to create a new test record from parts of other test records.

- `Delete` removes test cases from a test record. The test cases are simply discarded.

- `Export` writes individual test cases in a test record to separate files. This command can be used to write test cases in executable form for automated testing.

- `Insert` is used to add test cases to a test record, but unlike `Add`, the test cases are added starting at a given index. This command can be used to re-order test cases in a test record.
- `List` displays a directory of the test record's contents.

Like ordinary JUMBL commands, `ManageTest` commands are case-insensitive and can be abbreviated. The following three commands are exactly the same:

```
$ jumbl ManageTest List MyTests.str
$ jumbl managetest list MyTests.str
$ jumbl m l MyTests.str
```

6.1.1 The List Command

The simplest test management command is `List`, which just displays a directory of the content of a test record. For example, the following lists the test cases present in MyTests.str:

```
$ jumbl ManageTest List MyTests.str
 index name                 model      key        method     events E F
------ -------------------- ---------- ---------- ---------- ------ - -
     1 example_min_1        example               coverage       36
     2 example_min_2        example               coverage        5
     3 example_1:1          example               random          3 E
     4 example_2:1          example               random         21 E
     5 example_3            example               random         11 E
     6 example_4            example               random         38 E F
     7 example_5            example               random         30 E
     8 example_1:2          example    highp      random         42
     9 example_2:2          example    highp      random          7
------ -------------------- ---------- ---------- ---------- ------ - -
     9 test cases           example    *many*     *many*        193
```

There are several pieces of information in the listing.

- Each test case is given a unique index, starting from one. Test cases may be referenced by their index.
- Each test case has a name, which is typically assigned based on the model, generation method, and order of generation. If two test cases have the same name, a colon and an integer are appended (as with `example_1:1` and `example_1:2`). Test cases may be referenced by name.
- The model from which each test case is generated is listed (in this case all test cases come from the same model). Tests in a single test record may come from different models.
- The key used when generating the test case is listed, and a blank means the default key was used. For random generation, this is the probability distribution used. For weighted distribution, this is the weight distribution. For minimum coverage, this is the cost distribution (if any).

- The method used to generate the test case is listed. This will be one of: *random* (random sampling, discussed in Section 6.4), *coverage* (part of a minimum-cost coverage set, discussed in Section 6.3), *weight* (created based on a weight function, discussed in Section 6.5), or *crafted* (created using the test case editor discussed in Section 6.6).
- The number of events in the test case (the test case length) is listed.
- If the test case has been executed, an E appears after the number of events.
- If the test case has failures recorded in it, an F appears at the end of the line.

The last line of the listing is a summary of the test cases shown. In this example, there are nine test cases with 193 events, total. The test cases come from many distributions and are generated by many methods, but they all come from the same model.

Only a single test record can be specified when using the `List` command.

6.1.2 Using Selectors

One may refer to test cases in a test record either by index (starting from one), or by name. Such a reference is called a *selector*. Selectors follow the test record name and start with an "at" symbol (@). Following the at symbol may be an index, a name, or a range (using either indices or names), and selectors may be combined. Be careful that there are *no spaces* between the test record name and the selector, or within the selector itself. The order in which selectors appear is significant.

The following example lists the first three test cases in the usual order, then lists the last three test cases in reverse order.

```
$ jumbl ManageTest List example.str@1-example_1:1@9-7
 index name                 model      key        method     events E F
------ -------------------- ---------- ---------- ---------- ------ - -
     1 example_min_1        example               coverage       36
     2 example_min_2        example               coverage        5
     3 example_1:1          example               random          3 E
     9 example_2:2          example    highp      random          7
     8 example_1:2          example    highp      random         42
     7 example_5            example               random         30 E
------ -------------------- ---------- ---------- ---------- ------ - -
     6 test cases           example    *many*     *many*        123
```

Note that the indices do not change; they refer to the absolute position of the test case in the test record. The summary line reflects only those test cases which have been selected. In this case, six test cases have been selected, with 123 events, total.

A single test case may be selected multiple times, and one may use the special index "e" to refer to the last test case. This example selects the last test case in a test record twice.

```
$ jumbl ManageTest List MyTests.str@e@e
 index name                 model      key        method     events E F
------ -------------------- ---------- ---------- ---------- ------ - -
     9 example_2:2          example    highp      random          7
     9 example_2:2          example    highp      random          7
------ -------------------- ---------- ---------- ---------- ------ - -
     2 test cases           example    highp      random         14
```

Selectors may be used when adding and inserting test cases from one test record into another, when deleting test cases from a test record, and when extracting test cases from a test record. If you are not sure about a selector, you can try it with the `List` command to make sure it is correct.

There is one final type of selector. One may follow the at symbol with a slash (/) and a regular expression (no spaces).[1] This selects those test cases whose names match the regular expression. This is very useful, since test case names typically come from the model name and generation method. For example, the following would list all test cases generated as part of a minimum coverage set.

```
$ jumbl ManageTest List MyTests.str@/.*_min.*
 index name                 model      key        method     events E F
------ -------------------- ---------- ---------- ---------- ------ - -
     1 example_min_1        example               coverage       36
     2 example_min_2        example               coverage        5
------ -------------------- ---------- ---------- ---------- ------ - -
     2 test cases           example               coverage       41
```

If you use a regular expression selector, it must be the only selector present. The particulars of the regular expression syntax are beyond the scope of this document, but these regular expressions are similar to those provided by Perl and other tools. A good text on constructing and using regular expressions is [11].

6.1.3 The Add and Insert Commands

There are two commands for adding test cases to test records: `Add` and `Insert`. The `Add` command adds test cases at the end of the test record, while the `Insert` command can add test cases anywhere in the test record. One can generate new test cases directly into a test record (see Section 6.2), so the `Add` command is not needed for that purpose. One typically uses the `Add` and `Insert` commands to re-order test cases in a test record, or to create new test records from existing test records, or to add manually created test cases to a test record (see Section 6.6).

To add test cases to a test record, give the `Add` command followed by the target test record. Then list any test cases to add. The following creates a new test case consisting of the first four test cases from MyTests.str:

```
$ jumbl ManageTest Add New.str MyTests.str@1-4
Creating new test record: New.str.
Added example_min_1 to record at position 1.
Added example_min_2 to record at position 2.
Added example_1 to record at position 3.
Added example_2 to record at position 4.
```

[1] It may occasionally be necessary to enclose the entire reference in single quotation marks to protect it from some versions of the UNIX shell: `'MyTests.str@/.*_min.*'`.

The following merges the complete New.str with the rest of the test cases from MyTests.str, and creates a new test record called Full.str (which will actually be the same as MyTests.str):

```
$ jumbl ManageTest Add Full.str New.str MyTests.str@5-e
Creating new test record: Full.str.
Added example_min_1 to record at position 1.
Added example_min_2 to record at position 2.
Added example_1 to record at position 3.
Added example_2 to record at position 4.
Added example_3 to record at position 5.
Added example_4 to record at position 6.
Added example_5 to record at position 7.
Added example_1 to record at position 8.
Added example_2 to record at position 9.
```

The `Insert` command works much the same way, except that it can add test cases starting at any index from one up to one past the last index in use (inserting at the end). To use the `Insert` command, specify the target test record, then the index at which to start inserting, and then give the test cases to insert. The following adds all the test cases in More.str to Full.str, starting at index three:

```
$ jumbl ManageTest Insert Full.str 3 More.str
Added demo_wt_1 to record at position 3.
Added demo_wt_2 to record at position 4.
Added demo_wt_3 to record at position 5.
Added demo_wt_4 to record at position 6.
Added demo_wt_5 to record at position 7.
```

6.1.4 The Delete Command

The `Delete` command removes test cases from a test record. To use it, give the test record after the `Delete` command, and specify the test cases to remove with a selector. The following removes the five test cases from Full.str which were added at the end of the last section (Section 6.1.3).

```
$ jumbl ManageTest Delete Full.str@/demo.*
Removed 5 test cases.
```

Only a single test record can be specified with the `Delete` command.

6.1.5 The Export Command

The `Export` command writes test cases from a test record as single files. As with usage models, test cases can be written in different formats. At the time of writing the following formats are supported:

- Saved Test Case (STC) files have the extension .stc, and are required for working with the test case editor (see Section 6.6).
- TestCase format is a special way of writing tests which can use information from the model file to create executable tests. This format can also create human-readable .txt files from test cases, and this is the default behavior of `Export`. See Figure 6.1 for an example of the output.

```
# ========================================
# Trajectory: 0
# Model:      ync_test
# Key:
# Method:     random
#
# Events:     3
# Including failure information.
# ========================================
# Step: 1, Trajectory: 0
[Enter]."start test"
# Step: 2, Trajectory: 0
[No Project]."Open Project"
# Step: 3, Trajectory: 0
[Project, No Change]."Exit"
```

Figure 6.1: A Test Case

The `Export` command can write tests in either format. To use it, specify the target format with `--type`. By default `Export` creates human-readable .txt files. The following will convert the first four test cases in MyTests.str to .txt files.

```
$ jumbl ManageTest Export MyTests.str@1-4
Writing example_min_1.txt.
Writing example_min_2.txt.
Writing example_1.txt.
Writing example_2.txt.
```

As with the `Write` command, a filename suffix can be specified with the `-s` or `--suffix` switch. The specified filename suffix is appended to the filename prior to the file extension. Further, the extension may be changed with the `-e` or `--extension` switch (include the period in the extension). The following writes the first four test cases to four other files with the extension .doc.

```
$ jumbl ManageTest Export MyTests.str@1-4 -sfoo -e.doc
Writing example_min_1foo.doc.
Writing example_min_2foo.doc.
Writing example_1foo.doc.
Writing example_2foo.doc.
```

The `Export` command can also prepare test cases for automated execution. See Section 6.9 for more information.

6.2 Test Generation

The JUMBL's `GenTest` command supports four automated test generation methods. Generating the minimum-length coverage set for a model is described in Section 6.3. Generating tests randomly based on a probability distribution is described in Section 6.4. Generating tests in order by weight is described in Section 6.5. Using TGL files is described in Section 6.7.

Test cases generated by `GenTest` are named according to the pattern [*model*][*suffix*][*number*]. For example, if the suffix is `_min_` and the model is ync_test, then the test cases will be named `ync_test_min_1`, `ync_test_min_2`, etc. Each generation method has a different default suffix, but the suffix may be changed with the `-s` or `--suffix` switch.

Test cases generated by `GenTest` are placed in test records (see Section 6.1). When test cases are generated directly from a model, they are placed in a test record with the same name as the model. When test cases are generated from a TGL file, they are placed in the test record named in the TGL file. This behavior can be overridden by specifying a target test record with the `-r` or `--record` switch. Test cases are always added to the end of the test record by `GenTest`.

6.3 Generating the Coverage Set

Test cases generated from the model must match the functionality of the software; if the test case says to select a particular menu item, that menu item must exist and be available. One can gain confidence that the usage model and the delivered software match one another by generating test cases which visit every arc in the test case. This ensures that every state / next event pair has been executed at least once.

The JUMBL can generate a collection of test cases which visit every arc in a model with the minimum number of test steps. This is done using the `GenTest` command with the `-m` or `--min` switch:

```
$ jumbl GenTest -m ync_test.tml
Solving: model ync_test.
Generating coverage: model ync_test.
Generated test: ync_test_min_1
Generated test: ync_test_min_2
Generated test: ync_test_min_3
Generated test: ync_test_min_4
```

This command will generate one or more test cases which, combined, cover all arcs of the model with the fewest number of test events. The default test case suffix for when generating the minimum coverage set is `_min_`, and the test cases resulting from the example just given will be named `ync_test_min_1`, `ync_test_min_2`, `ync_test_min_3`, and `ync_test_min_4`.

Suppose that there are two arcs from [A] to [B], the first with label “x” and the second with label “y.” During testing performing event “x” requires approximately one second, whereas performing “y” may require several minutes. When generating the coverage set it may be necessary to move from [A] to [B] several times. If the algorithm chooses “x” all but one time, then the test is short. If the algorithm chooses “y” all but one time, then the test may be very long.

The JUMBL provides a means to bias the choices made when generating the coverage set by associating a cost distribution with the arcs. (See Section 5.1 for information about distributions). Consider the model snippet in Figure 6.2. A special distribution `cost` is declared. The declaration `cost:($assume(1)$)` gives each arc unit cost by default. From the state shown, choosing “No” means that changes are discarded. Choosing “Yes” means that changes are to be saved, and the user must then enter a filename and verify that changes are correctly saved. This is about ten times costlier than not saving changes, so the cost for this arc is set to ten. Now when the coverage set is generated, the “Yes” arc will be visited once (for coverage), but the “No” arc will be used all other times for the transition from [ync.Enter] to [No Project].

```
cost:($assume(1)$)
model ync_test_flat
    [ync.Enter]
        "No"          [No Project]
      cost:($10$)
        "Yes"         [No Project]
        "Cancel"      [Project, Change]
...
end
```

Figure 6.2: Using Costs

To tell `GenTest` what distribution to use for costs, specify the distribution key with `--key` or `-k`. The following will generate the coverage set using the `cost` distribution.

```
$ jumbl GenTest --min --key=cost ync_test.tml
Solving: model ync_test.
Generating coverage: model ync_test.
Generated test: ync_test_min_1
Generated test: ync_test_min_2
Generated test: ync_test_min_3
Generated test: ync_test_min_4
```

6.4 Generating Random Tests

Random test generation is a simple way to get a large number of tests. Tests can be generated randomly, based on the probabilities in the model. This is done via the `GenTest` command:

```
$ jumbl GenTest ync_test.tml
Solving: model ync_test.
Generating tests: model ync_test.
Generated test: ync_test_1.
```

By default, the `GenTest` command generates a single test case. The default suffix for random test cases is a single underscore (_), and for the example just given, the test case will be named `ync_test_1`.

Multiple tests can be generated by using the `--num` or `-n` switches. The following will generate 1,500 test cases from the model ync_test.

```
$ jumbl GenTest -n 1500 ync_test.tml
Solving: model ync_test.
Generating tests: model ync_test.
Generated test: ync_test_1.
...
Generated test: ync_test_1500.
```

These tests will have the names `ync_test_1`, ..., `ync_test_1500`.

A model may have more than one distribution on it (see Section 5.1), and tests can be generated based on any distribution in the model. If no distribution is specified, the default distribution is used. Note that the default distribution is always present, so test cases can always be randomly generated. To specify a distribution other than the default, use the `--key` or `-k` switches.

```
$ jumbl GenTest -n 100 ync_test.tml -k highw -s _highw_
Solving: model ync_test.
Generating tests: model ync_test.
Generated test: ync_test_highw_1.
...
Generated test: ync_test_highw_100.
```

6.5 Generating Weighted Tests

Each test case may have a weight associated with it. This weight can be one of two forms:

- The probability of the test case being generated. This is the product of the probabilities of the arcs traversed in the test case.
- The sum of weights on the arcs traversed in the test case.

Test cases can be generated in order by weight using the `GenTest` command with the `-w` or `--weight` switch:

```
$ jumbl GenTest -w ync_test.tml
Solving: model ync_test.
Generating 1 importance sampled tests: model ync_test.
Generated test: ync_test_wt_1
```

By default `GenTest` generates weighted tests in order by decreasing probability, with the highest probability test cases first. Note that, for most models, there is no lowest-probability test case because there is no longest test case, so this ordering makes sense.

By default, the `GenTest` command generates a single test case. The default suffix for weighted test cases is (`_wt_`), and for the example just given, the test case will be named `ync_test_wt_1`.

Multiple tests can be generated by using the `--num` or `-n` switches. The following will generate the highest-probability 1,500 test cases from the model ync_test.

```
$ jumbl GenTest --weight -n 1500 ync_test.tml
Solving: model ync_test.
Generating 1500 importance sampled tests: model ync_test.
Generated test: ync_test_1.
...
Generated test: ync_test_1500.
```

These tests will have the names `ync_test_wt_1`, ..., `ync_test_wt_1500`.

A model may have more than one distribution on it (see Section 5.1), and tests can be generated based on any distribution in the model. If no distribution is specified, the default distribution is used. Note that the default distribution is always present, so test cases can always be generated by weight. To specify a distribution other than the default, use the `--key` or `-k` switches.

```
$ jumbl GenTest --weight -n 100 ync_test.tml -k highw -s _highw_
Solving: model ync_test.
Generating 100 importance sampled tests: model ync_test.
Generated test: ync_test_highw_1.
...
Generated test: ync_test_highw_100.
```

Whenever test cases are generated by probability, the outgoing arc values are automatically normalized so they sum to one.

As mentioned previously, it is also possible to generate in order by the sum of weights. Under this approach, a distribution assigns each arc a nonnegative weight, and test cases are generated in order starting with the lowest sum (from least costly to most costly). Note that, for most models, there is no highest sum because there is no longest test case, so this ordering makes sense. To use the sum of weights instead of the product of normalized probabilities, use the `--sum` switch.

```
$ jumbl GenTest --weight --sum -n 10 ync_test.tml -s _sum_
Solving: model ync_test.
Generating 10 importance sampled tests: model ync_test.
Generated test: ync_test_sum_1.
...
Generated test: ync_test_sum_10.
```

Whenever test cases are generated in order by weight, if an arc has a negative or zero weight then the arc is ignored. This is consistent with the idea that probability zero arcs are intended to be "disabled." Zero weights would otherwise present a problem for minimizing the sum.

6.6 Generating Tests Manually

While it is possible to generate test cases automatically using `GenTest`, it is sometimes desirable to run a specific test. The JUMBL provides the command `CraftTest` to allow users to manually create tests from a model, or to edit existing tests. Unlike the other commands described, `CraftTest` is a GUI application. To start `CraftTest`, use:

```
$ jumbl CraftTest
```

This will open the test case editor window, shown in Figure 6.3. The window is divided into two sections. On the left will appear the events of the test case, in order. On the right will appear the list of arcs from the currently selected node. There is a collection of buttons at the bottom of the window. These are enabled when their use is appropriate. Finally, there is a Test Case menu at the top of the window. From this menu users can create new test cases, open existing test cases, save test cases, and exit the editor.

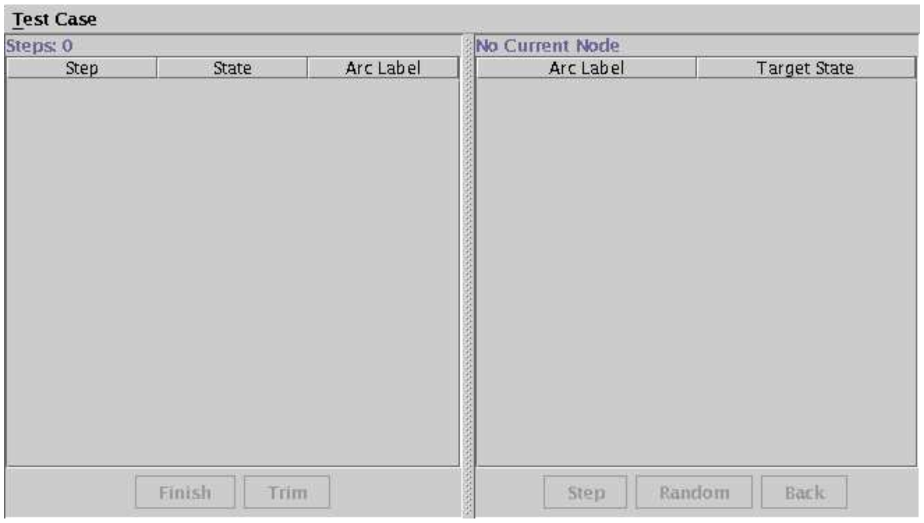
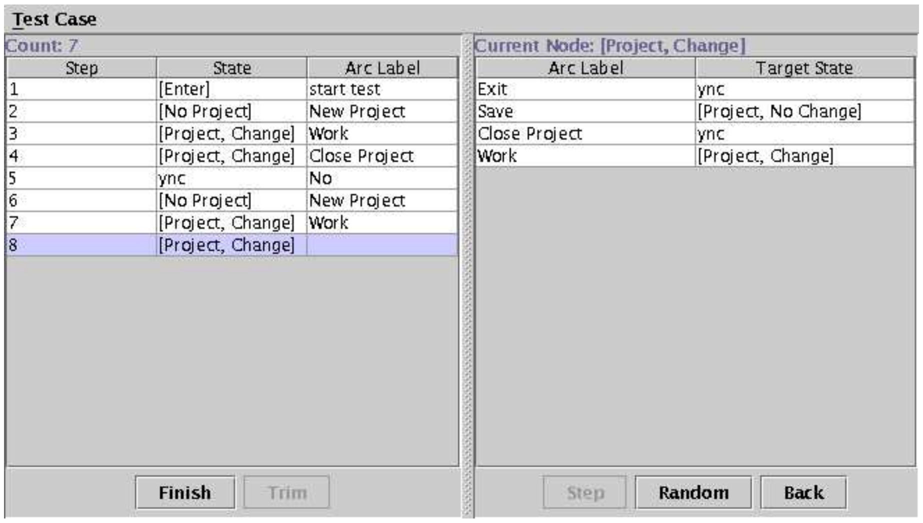


Figure 6.3: The CraftTest Window

Opening or Creating a Test Case

If you want to create a new test case from an existing model, select New From Model... from the Test Case menu and then select the model from which you want to generate a test case. If you want to edit an existing test case, select Open Test Case... from the Test Case menu and then select the test case you want to edit. Note that `CraftTest` must always locate and load the SM file for the model associated with a test case, so the model must be in the object search path (see Section 4.2). If the model cannot be located, you will get an error message. Further, the test case you want to edit cannot be in a test record; it must be exported to an STC file (see Section 6.1.5 for how this is done).

Once a new test case has been created, or an existing test case has been loaded, the display will look similar to Figure 6.4. In this case the test case is not yet complete (the sink has not been reached). There are seven steps currently in the test case. Note that there is no arc label in the eighth row; the user is ready to enter the eighth step. The state for the highlighted row is [Project, Change], and its arcs are listed on the right.



Count: 7

Step	State	Arc Label
1	[Enter]	start test
2	[No Project]	New Project
3	[Project, Change]	Work
4	[Project, Change]	Close Project
5	ync	No
6	[No Project]	New Project
7	[Project, Change]	Work
8	[Project, Change]	

Current Node: [Project, Change]

Arc Label	Target State
Exit	ync
Save	[Project, No Change]
Close Project	ync
Work	[Project, Change]

Figure 6.4: An Open Test Case

Editing a Test Case

To add a particular step to a test case which is not complete, select the last row of the test case on the left. The list of outgoing arcs will appear on the right. Select one of the arcs, and click Step at the bottom of the window. The step will be added to the test case.

To add a random step (using the default distribution) to a test case which is not complete, just click Random. The step is added to the end of the test case.

To remove the last step from a test case, click Back.

To remove several steps from a test case, highlight a row on the left, and click Trim. All steps from the selected step to the end of the test case will be removed.

Finally, to finish a test case by taking random steps (using the default distribution) just click Finish.

Saving a Test Case

Only complete test cases may be saved. Once the test case is complete, the Save Test Case... item of the Test Case menu will be enabled. Choose Save Test Case... from the Test Case menu and enter the test case name in the file dialog. The test case will be saved in the desired format.

STC is the default format for test cases. The appropriate file extension will be added when the test case is saved. If you wish to add this test case to a test record, you may do so with the `Add` or `Import` commands (see Section 6.1.3).

6.7 Using Test Generation Files

The JUMBL provides a powerful means to generate sophisticated test cases. This approach uses a special XML extension called Test Generation Language (TGL). By writing TGL files and using them with the `GenTest` command, one can automate the production of very sophisticated test cases.

6.7.1 Motivation: Tangling Test Cases

Suppose a system is composed of multiple independent components, such as multiple phones connected to a phone switch, or multiple modeless dialogs. Building a single model to track the state of each independent actor would result in a very large (and difficult to maintain) model. A better approach is to build several models, and then merge or "tangle" the tests from these models.

Consider the model of Figure 6.5. Suppose we want to test a system with one hundred different phones. We could create a single test which "multiplexes" the phones as follows. One hundred test cases could be generated using `GenTest`, and then we could randomly interleave the events from these test cases, creating a new "tangled" test case which tested more than one phone at a time. Figure 6.6 shows an example of what such a test case might look like. Each of the original test cases becomes a "trajectory" in the new test case. In the case shown all the original test cases came from the same model and distribution, but this is not necessary. The header of the test case lists the model and distribution key for each trajectory. Then the events are listed. Note that events from the different trajectories have been interleaved. In this case the test runs until all phones have been hung up.

6.7.2 Overview

A TGL file is a text file containing valid XML, a language similar to HTML. Consult one of the many references on XML if you are unfamiliar with the format.

The top-level element of any TGL file must be `<TestGenerator>`, with a name attribute specifying the name of the test generator. If the name attribute is omitted,

```
// A single use of the phone is all events from
// lifting the receiver until the receiver is
// replaced.
model phone
source [On Hook]
    "lift receiver"    [Off Hook]
    "incoming call"    [Ringing]
[Off Hook]
    "hang up"          [Exit]
    "dial busy"        [Busy Tone]
    "dial bad"         [Error Tone]
    "dial good"        [Ring Tone]
[Busy Tone]
    "hang up"          [Exit]
[Error Tone]
    "hang up"          [Exit]
[Ring Tone]
    "hang up"          [Exit]
    "connect"          [Connected]
[Connected]
    "hang up"          [Exit]
    "disconnect"       [Off Hook]
[Ringing]
    "lift receiver"    [Connected]
    "disconnect"       [On Hook]
end
```

Figure 6.5: Simple Model for a Phone

```
# ========================================
# Trajectory: 99
# Model:      phone
# Key:
#
# Trajectory: 98
# Model:      phone
# Key:
#
...
#
# Trajectory: 1
# Model:      phone
# Key:
#
# Trajectory: 0
# Model:      phone
# Key:
#
# Events:     389
# Including failure information.
# ========================================
# Step: 1, Trajectory: 62
[On Hook]."incoming call"
# Step: 2, Trajectory: 65
[On Hook]."incoming call"
# Step: 3, Trajectory: 92
[On Hook]."lift receiver"
# Step: 4, Trajectory: 90
[On Hook]."incoming call"
# Step: 5, Trajectory: 68
[On Hook]."lift receiver"
# Step: 6, Trajectory: 22
[On Hook]."incoming call"
...
# Step: 385, Trajectory: 0
[Busy Tone]."hang up"
# Step: 386, Trajectory: 91
[Off Hook]."dial bad"
# Step: 387, Trajectory: 20
[Connected]."hang up"
# Step: 388, Trajectory: 91
[Error Tone]."hang up"
# Step: 389, Trajectory: 43
[Error Tone]."hang up"
```

Figure 6.6: An Interleaved Test Case

```
<?xml version="1.0"?>
<TestGenerator name="many_phones">
    <!-- Tangle all enclosed test cases with infinite
         channels.
    -->
    <Tangle>
        <!-- Generate one hundred random test cases. -->
        <GenTest num="100" model="phone"/>
    </Tangle>
</TestGenerator>
```

Figure 6.7: A TGL File

"unnamed" is assumed. Inside the `<TestGenerator>` element can be any number of test sources and operators. These are discussed in the following sections. An example TGL file is shown in Figure 6.7. This file creates one hundred random test cases from the phone model, and then interleaves them as described in 6.7.1.

A TGL file may describe how to produce a single, complex test case (as in Figure 6.7), or it may describe how to produce many tests. Tests may be generated, or read in from test records. They may be tangled, concatenated, or discarded. Test cases may be shuffled and chosen at random from a collection. Portions of the TGL file may iterate some number of times. Tests are created or read using test sources (see Section 6.7.3), and combined using operators (see Section 6.7.4).

In many cases where numbers are required, a range may be used. This is specified by using square brackets and giving two integers separated by a hyphen. For example, if zero to 100 test cases are desired, one might use the following:

```
<!-- Generate up to 100 test cases.  Possibly generate
     no test cases at all.
-->
<GenTest model="phone" num="[0-100]"/>
```

Test cases are generated from a TGL file using the `GenTest` command. Give the name of the TGL file after the command. If desired, you can specify a number of times to execute the file with the `-n` or `--num` switch. This has the same effect as wrapping the entire file in a `<Repeat>` operator (see Section 6.7.4). In particular, weighted generators are *not* reset between runs.

Test cases are placed (by default) in a test record specified by the name attribute of the top-level `<TestGenerator>` element, but this can be overridden by the `-r` or `--record` switch. If the TGL file in Figure 6.7 were placed in many_phones.tgl, then the following would use it to generate five test cases, storing them in the Big.str test record:

```
$ jumbl GenTest -n 5 -r Big.str many_phones.tgl
```

```
Creating new test record: Big.str.
Running generator: many_phones.tgl.
Generated: phone_1
Running generator: many_phones.tgl.
Generated: phone_101
Running generator: many_phones.tgl.
Generated: phone_201
Running generator: many_phones.tgl.
Generated: phone_301
Running generator: many_phones.tgl.
Generated: phone_401
```

6.7.3 Using Test Sources

There are two test sources available through TGL: `<GenTest>`, which functions very much like the `GenTest` command, and `<TestRecord>`, which allows reading test cases from a test record.

The `<GenTest>` element may not contain any other elements, and takes several attributes which control its operation. It is replaced with the generated tests, if any.

- `model="`*M*`"`
  Required. The name of the model from which test cases are to be generated. This must be present, and the referenced model must be compiled to an SM file (see Section 4.2) in the object search path.

- `num="`*N*`"`
  Default is one. A random range is allowed here. Specify that *N* test cases are to be generated. This may only be used when the generation method is weighted or random.

- `method="`*M*`"`
  Default is random. Specify the method of generation. *M* must be one of the strings random, weight, or min, for random generation, generation in order by weight, and generating the minimum-cost coverage set, respectively. For details on these different methods, see Sections 6.3, 6.4, and 6.5.

- `sum`
  Default is false. If this attribute is present (it may be given the value "true," if desired) and the generation method is weighted generation, then minimize the sum instead of maximizing the product (see Section 6.5). If omitted, the product is maximized.

- `suffix="`*S*`"`
  Default depends on method. Specify a suffix for test case names. The default suffix is determined by the generation method chosen. For random generation, it is the empty string. For weighted generation, it is `_wt`. For the minimum-cost coverage set, it is `_min`. Test cases are named by taking the base name, appending a suffix, and then appending an increasing number.

- `name="`*N*`"`
  Default is model name. If this attribute is present, it specifies that *N* will be the base name for test cases generated. Test cases are named by taking the base name, appending a suffix, and then appending an increasing number.

- `first="`*F*`"`
  Default is one. Specify the index of the first test case created. Test cases are named by taking the base name, appending a suffix, and then appending the index.

- `key="`*K*`"`
  Default is either none, or the default distribution, depending on the method. Specify a probability or cost distribution. For random generation, this specifies the probability distribution to use. For weighted and minimum-cost coverage, this specifies the cost distribution to use.

- `id="`*X*`"`
  No default. Used to tie together weighted generators. If not present, every instance of a weighted generator is treated as different. That is, if you generate the first ten test cases from a model in one place, and then use a different generator to generate the first ten test cases from the same model later, you'll get the same ten test cases each time. To avoid this, specify the same id for both. Then the second generator will resume where the first left off, and generate the *next* ten test cases.

The following are some examples.

```
<!-- Generate one to ten test cases, inclusive. -->
<GenTest model="phone" num="[1-10]"/>

<!-- Generate the minimum coverage set. -->
<GenTest model="phone" method="min"/>

<!-- Generate the first twenty test cases in order,
     by probability.
-->
<GenTest model="phone" method="weight" num="10" id="1"/>
<GenTest model="phone" method="weight" num="10" id="1"/>
```

The `<TestRecord>` element is replaced with test cases taken from a test record. The test record is specified with the file attribute. By default all test cases in the test record are used. If this is not desired, a selector attribute may be used whose value must be a valid selector (including the at symbol). See Section 6.1.2 for more information on selectors. The following are some examples.

```
<!-- Get the first five test cases from the test record,
     followed by the last test case in the record,
     followed by all the minimum-coverage scripts.
```

```
-->
<TestRecord file="MyTests.str" select="@1-5@e"/>
<TestRecord file="MyTests.str" select="@/.*_min.*"/>
```

6.7.4 Using Operators

There are several operators: `<Echo>`, `<Tangle>`, `<Cat>`, `<Random>`, `<Shuffle>`, `<Discard>`, and `<Repeat>`. All of these except `<Echo>` may enclose other operators and test case sources.

The most basic operator is `<Echo>`. It has no attributes and simply prints the enclosed text. This is useful for seeing what is happening as the TGL is used to construct test cases.

```
<Echo>About to generate some tests.</Echo>
<GenTest model="phone" num="10"/>
<Echo>Finished generating ten random tests.</Echo>
```

The `<Tangle>` operator supports precisely the kind of operation described in Section 6.7.1. The events of all enclosed test cases are randomly interleaved to form a single test case, which replaces the `<Tangle>` element.

When tangling test cases, one can specify the number of *channels* via the `channels` attribute. The number of channels selects the degree of multiplexing. To use the example of Section 6.7.1, if `channels="10"` were used then at most ten phones could be off hook at any given time. If there were 100 channels then all phones could potentially be off hook at the same time (though this will not necessarily happen, since the interleaving is random). If the channels attribute is not present, it is assumed that there are infinitely many channels. A random range may be used for the number of channels.

```
<!-- Construct a test with a multiplexing degree of
     at most ten.
-->
<Tangle channels="[2-10]">
    <!-- Randomly generate up to 100 test cases. -->
    <GenTest model="phone" num="[10-100]"/>
</Tangle>
```

The `<Cat>` operator works like the `<Tangle>` operator with channels set to one. That is, it simply concatenates its enclosed test cases, in order one after another, into a single test case. It has no attributes.

The `<Random>` operator randomly selects and performs one child element and discards the rest. In the following example, `<Random>` is replaced with either the minimum coverage scripts, or ten random scripts, but not both.

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

The `<Shuffle>` operator randomly re-orders the enclosed test cases, and takes no attributes. The following will take the minimum coverage set, randomly re-order the scripts, and then concatenate them into a single test case.

```
<Cat>
    <Shuffle>
        <GenTest model="phone" method="min"/>
    </Shuffle>
</Cat>
```

The `<Discard>` operator simply discards all enclosed test cases. This is useful for omitting test cases when generating by weight. The following generates the second-most-likely test case.

```
<Discard>
    <GenTest model="phone" method="weight" id="1"/>
</Discard>
<GenTest model="phone" method="weight" id="1"/>
```

The `<Repeat>` operator just repeatedly evaluates the enclosed operators and test sources some number of times. It has a required attribute `num`, which must be the number of times to repeat and may be a random range. The following chooses a random test case from among the ten most likely.

```
<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>
```

6.8 Using Labels

A “label” is arbitrary data which is attached to a model, state, or arc. In this respect labels are similar to constraints, and use keys in much the same way constraints do. Labels differ in that they can contain arbitrary data and *must always* have a key. There are no “default” labels for a model. In TML constraints are always placed *before* the declaration of the thing they constrain; labels are placed *after* the declaration of the thing they to which they apply.

Labels take two forms:

- Single-line labels start with `|$` and extend to the end of the line. They include any end of line characters such as a carriage return and / or a linefeed character.
- Multi-line labels start with `{$` and end with `$}`. They include everything within these markers.

Labels can contain any data, including instructions to testers, but their primary use is to attach automated testing information to the model. For example, suppose a phone switch is being tested, and each test case must be converted into code which will be executed against the system to simulate the operation of the phones. Figure 6.8 shows such code attached to a portion of the phone model. Note that "boilerplate" information to be included at the start of every test can be attached to either the model (as it is here) or the source, and information to be included at the end of every test can be attached to the sink (in this case, [Exit]).

Labels provide special variables, which are expanded on output. For example, each occurrence of the `$t` in the phone example will be replaced with the event's trajectory number. This allows phones from different trajectories to be distinguished. All variables are listed in Table 6.1.

Table 6.1: Variables Expanded in Labels

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 $0 \leq x \leq n$.

Further, labels (and even state and arc names) allow the use of character escapes. The character escapes supported are listed in Table 6.2. Additionally, if a backslash occurs at the end of a line, the backslash and the end of line are discarded, and the two lines are joined. These character escapes make it possible to include arbitrary data in a label.

The `Export` command of `ManageTest` generates human-readable output by default. To write the data for a label, use the `-k` or `--key` switches. In order for the JUMBL to correctly convert a test case to label data, SM files for all required models must be available in the object search path (see Section 4.2).

```
$ jumbl ManageTest Export -k a phone.str@1
Writing: phone_1.txt.
```

```
// A single use of the phone is all events from
// lifting the receiver until the receiver is
// replaced.
model phone
a:|$ #include "phonetest.h"
  |$ int main() {
  |$     // Temporary phone number.
  |$     int n = 0;
  |$
  |$     // Initialize the test.
  |$     initialize_test();
source [On Hook]
    "lift receiver"
    a:|$     // Lift the receiver.
      |$     lift($t);
    [Off Hook]
    "incoming call"
    a:|$     // Call this phone.
      |$     call($t);
    [Ringing]
[Off Hook]
    "hang up"
    a:|$     // Hang this phone up.
      |$     hangup($t);
    [Exit]
    "dial busy"
    a:|$     // Locate a busy phone and call it.
      |$     n=findbusy();
      |$     dial($t, n);
    [Busy Tone]
...
[Exit]
a:|$     // Finished with the test.
  |$     close_test();
  |$ }
end
```

Figure 6.8: Phone Model with Labels

Table 6.2: Character Escapes

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`.

This produces output similar to Figure 6.9. Observe that there is a problem with this output; the comment lines added by `Export` start with a hash mark (`#`), which will confuse the compiler. There are two options in this case:

- Suppress these lines. This can be done by adding the `--no-header` and `--no-step` switches.

```
$ jumbl ManageTest Export -k a phone.str@1 --no-header --no-step
Writing: phone_1.txt.
```

- Convert them into C comments. This can be done by specifying the comment start with `--start` and the comment end with `--end`, as in:

```
$ jumbl ManageTest Export -k a phone.str@1 --start=/* --end=*/
Writing: phone_1.txt.
```

Instead of specifying these switches every time, one can set options in the options editor (see Section 2.3) on the TestCase tab. The switches mentioned above correspond to the following options.

- [TestCase.writeHeader]
  Disabling this is equivalent to the `--no-header` switch, and suppresses the header block of the test case listing the trajectories and event count.
- [TestCase.writeStep]
  Disabling this is equivalent to the `--no-step` switch, and suppresses the step and trajectory number for each step of the test case.

```
# ========================================
# Trajectory: 0
# Model:      phone
# Key:
# Method:     random
#
# Events:     7
# Including failure information.
# ========================================
# Step: 1, Trajectory: 0
 #include "phonetest.h"
 int main() {
     // Temporary phone number.
     int n = 0;

     // Initialize the test.
     initialize_test();
     // Lift the receiver.
     lift(0);
# Step: 2, Trajectory: 0
     // Locate a good phone and call it.
     n=findgood();
     dial(0, n);
# Step: 3, Trajectory: 0
     // Connect the phones.
     connect(0);
# Step: 4, Trajectory: 0
     // Connect the phones.
     disconnect(0);
# Step: 5, Trajectory: 0
     // Locate a good phone and call it.
     n=findgood();
     dial(0, n);
# Step: 6, Trajectory: 0
     // Connect the phones.
     connect(0);
# Step: 7, Trajectory: 0
     // Hang this phone up.
     hangup(0);
     // Finished with the test.
     close_test();
 }
```

Figure 6.9: Basic Label Output

- [TestCase.infoPrefix]
  This is the prefix to use for "informational" lines, and it is equivalent to the `--start` switch.

- [TestCase.infoSuffix]
  This is the suffix to use for "informational" lines, and it is equivalent to the `--end` switch.

This solves one problem, but the test case is still being written with the .txt extension when what is desired is a .cpp extension. This can be solved using the `-e` or `--ext` switches to specify the file extension, including the period.

```
$ jumbl ManageTest Export -k a phone.str@1 --no-header --no-step -e.cpp
Writing: phone_1.cpp.
```

This will produce the output shown in Figure 6.10, and write it to `phone_1.cpp`.

6.9 About Test Automation

The JUMBL's label system (see Section 6.8) was chosen because it was the most general method for supporting test automation. A common approach has been to develop a library of functions, with one function for each test event. The function executes the test event (clicking a button, sending a command, etc.) and then verifies the result of the action to determine if it failed or succeeded. The test automation can then either write this to a test log file, or invoke the JUMBL to record the results of the test (see Chapter 7). Most of this is common across all tests in a domain, and so a general (though domain-specific) test library must be developed as the first step.

It is also possible to simply execute the events and record the results for later analysis to determine pass / fail. This approach requires that a separate test "oracle" be written, or that a person look over the output.

However pass / fail is determined significant effort is required to support automated testing. It is important that the test environment be known in advance, and that a specification is available for the system to be tested.

```
#include "phonetest.h"
int main() {
    // Temporary phone number.
    int n = 0;

    // Initialize the test.
    initialize_test();
    // Lift the receiver.
    lift(0);
    // Locate a good phone and call it.
    n=findgood();
    dial(0, n);
    // Connect the phones.
    connect(0);
    // Connect the phones.
    disconnect(0);
    // Locate a good phone and call it.
    n=findgood();
    dial(0, n);
    // Connect the phones.
    connect(0);
    // Hang this phone up.
    hangup(0);
    Finished with the test.
    close_test();
}
```

Figure 6.10: Alternate Label Output

Chapter 7

Recording Test Results

As tests are performed, their results must be recorded in the test record. The JUMBL provides a utility to record the results of testing.

7.1 Recording Basic Results

The `RecordResults` command can be used to record the results of running a test in a test case.

```
$ jumbl RecordResults ync_test.str
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
Recording results for test case: 2 ync_test_2.
Recording results for test case: 3 ync_test_3.
Recording results for test case: 4 ync_test_min_1.
Recording results for test case: 5 ync_test_min_2.
Recording results for test case: 6 ync_test_min_3.
Recording results for test case: 7 ync_test_min_4.
Recording results for test case: 8 ync_test_1.
Recording results for test case: 9 ync_test_wt_1.
Recording results for test case: 10 ync_test_wt_1.
```

By default the `RecordResults` command records that every test in each supplied test record was run from start to finish without failure. If failures were detected, select the failed test case (see Section 6.1.2 for information on selectors) and follow this with a comma-separated list of the step numbers of the failures. For example, the following records that steps 5 and 7 failed in the second test case in the test record.

```
$ jumbl RecordResults ync_test.str@2,5,7
Recording results for: ync_test.str
Recording results for test case: 2 ync_test_2.
```

This records that the test was run from start to finish, and that failures occurred at steps 5 and 7. *Note that there must be no spaces in the failure list.*

It is possible that testing was stopped due to a failure (for example if the software crashed and the next step could not be performed). To indicate this, include S at the end of the failure list. The following records that step 5 and 7 failed, and that testing stopped at step 7.

```
$ jumbl RecordResults ync_test.str@4,5,7,S
Recording results for: ync_test.str
Recording results for test case: 4 ync_test_min_1.
```

The `RecordResults` command replaces any prior results in the test case. For example, if a large number of tests are run and only a few fail, it is convenient to first record success for all tests, and then mark the failures. For example, the following records that all tests except the first succeeded.

```
$ jumbl RecordResults ync_test.str
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
...
Recording results for test case: 10 ync_test_wt_1.
$ jumbl RecordResults ync_test_1.str@1,5,7,S
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
```

This can be done with a single command by using more than one selector.

```
$ jumbl RecordResults ync_test.str@1-e@1,5,7,S
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
...
Recording results for test case: 10 ync_test_wt_1.
Recording results for test case: 1 ync_test_1.
```

7.2 Using Results Files

When many tests are being run, as with automated testing, it will be inconvenient to enter the failure information by hand. The `RecordResults` command takes the `--file` switch which can be used to specify a file containing the results of testing. This file has a simple format which is essentially the same as the command line. One test record is specified per line, possibly with a selector and the failures given in a comma-separated list after the selector. If testing stopped after the last failure, an S is included. For example, if Figure 7.1 is the content of the file `test_results`, then the following would record these results in the corresponding test cases.

```
ync_test.str@1,5,7,S
ync_test.str@2
ync_test.str@3
ync_test.str@4
ync_test.str@5
ync_test.str@6,3
ync_test.str@7
ync_test.str@8,2,S
ync_test.str@9
ync_test.str@10
```

Figure 7.1: A Test Results File

```
$ jumbl RecordResults --file=test_results
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
...
Recording results for test case: 10 ync_test_wt_1.
```

7.3 Resetting Results

It is also possible to erase any recorded test results, marking test cases as "not executed." This is done by specifying the `-r` or `--reset` switch to `RecordResults`. The following will reset all test cases to the "not executed" state.

```
$ jumbl RecordResults --reset ync_test.str
Recording results for: ync_test.str
Recording results for test case: 1 ync_test_1.
...
Recording results for test case: 10 ync_test_wt_1.
```

7.4 Viewing Results

If the results of testing have been recorded with `RecordResults`, they can be viewed using the `Export` test management command (see Section 6.1.5). By default `Export` includes informational lines about failures and where testing stopped. Figure 7.2 shows test case output which contains failure information (the italicized lines).

If this information is not desired, it can be suppressed by including the `--no-fail` switch of the `Export` command.

```
# ========================================
# Trajectory: 0
# Model:      ync_test
# Key:
# Method:     random
#
# Events:     12
# Including failure information.
# ========================================
# Step: 1, Trajectory: 0
[Enter]."start test"
# Step: 2, Trajectory: 0
[No Project]."Open Project"
# Step: 3, Trajectory: 0
[Project, No Change]."Work"
# Step: 4, Trajectory: 0
[Project, Change]."Work"
# Step: 5, Trajectory: 0
[Project, Change]."Close Project"
# FAILED AT 5
# Step: 6, Trajectory: 0
ync."No"
# Step: 7, Trajectory: 0
[No Project]."Open Project"
# FAILED AT 7
# TESTING STOPPED AT 7
# Step: 8, Trajectory: 0
[Project, No Change]."Close Project"
# Step: 9, Trajectory: 0
[No Project]."Open Project"
# Step: 10, Trajectory: 0
[Project, No Change]."Close Project"
# Step: 11, Trajectory: 0
[No Project]."Open Project"
# Step: 12, Trajectory: 0
[Project, No Change]."Exit"
```

Figure 7.2: Test Case With Failure Information

Chapter 8

Analyzing a Test Record

Once testing has been performed, many product and process measures are available based on the testing experience informed by the usage model. The JUMBL can generate a comprehensive report based on testing experience, including estimated product field reliability and indicators of where unreliability is most significant, and stopping criteria such as the variance of estimators and the relative Kullback.

As with model analysis (see Chapter 5) one may perform a test analysis against different distributions. See Section 5.1 for information about using distributions.

8.1 Analyzing a Test Record

The JUMBL can analyze a test record in STR or TCML format. To analyze a test record, use the `Analyze` command (note the spelling):

```
$ jumbl Analyze ync_test.str
Scheduling test record ync_test.str.
Analyzing scheduled test records.
Writing analysis for model ync_test.
```

This command will produce a test analysis report in HTML with the name `ync_test_ta.html`, which may be opened for viewing in a web browser, and is described in Section 8.2.

By default the analyses are performed against the default distribution, and analyses are provided for any models represented in the test record. The `--model` switch can be used one or more times to select models whose analysis is to be performed, if analysis of all models is not desired. The distribution to use for analysis can be specified with `--key` or `-k`, and the `Analyze` command supports the suffix switch described in Chapter 4, with a default suffix of `_ta` (for test analysis). For example, to generate an analysis report against the `highw` distribution, use:

```
$ jumbl Analyze --key=highw --suffix=highw ync_test.sm
Scheduling test record ync_test.str.
Analyzing scheduled test records.
Writing analysis for model ync_test.
```

This will produce a file named `ync_testhighw.html`. Note that the distribution from which the test case was generated is ignored; all test cases are analyzed using either the default or the specified distribution. Thus results will be more accurate if test cases in a test record generally come from the same distribution.

When a model contains model references, as does the model of Figure 5.1, these are treated as states for the purpose of analysis (this is similar to flattening by collapsing; see Section 4.8).

8.2 Interpreting Test Analysis Results

Test analysis reports provide considerable information about the testing experience. First, counts are presented.

- The generation count is the number of times the node, arc, or stimulus appears in generated tests.
- The execution count is the number of times the node, arc, or stimulus is recorded as having been executed under test.
- The failure count is the number of failures recorded for the node, arc, or stimulus.

The generation count may be greater than the execution count if some tests have not been executed, or if failures prevented executing all of a test.

Second, reliabilities are presented. These may be interpreted as the long-run probability that an execution of the selected stimulus, arc, or test will not fail. Two kinds of reliability values are presented.

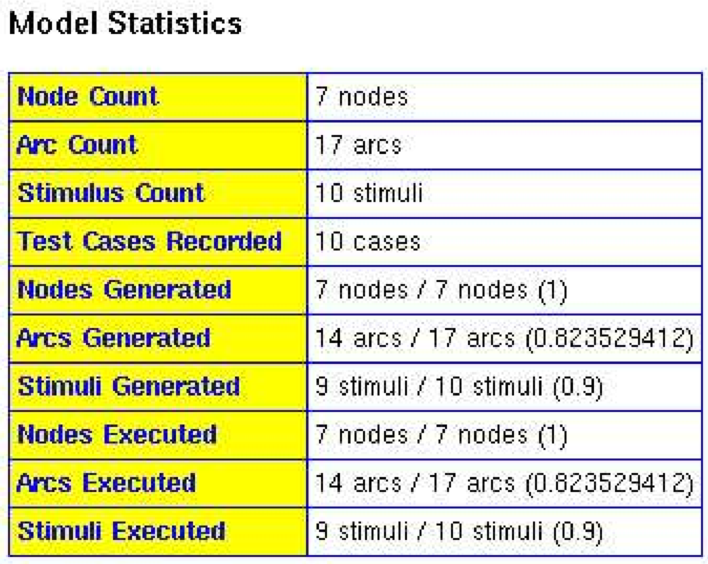
- The usual reliability based on the execution counts, and including the effects of any observed failures.
- An "optimum" reliability based solely on the generation counts. This is the reliability which would be expected if all recorded tests were run without failure, and can be computed before any tests have actually been run.

The test analysis report is divided into four sections, which provide different levels of granularity:

- The **model** section reports overall counts and coverage statistics.
- The **stimuli** section reports stimulus counts and reliabilities.
- The **arcs** section reports arc counts and reliabilities.
- The **reliabilities** section reports overall (end-to-end) reliabilities and stopping criteria.

Model Statistics

An example of the model statistics section is presented in Figure 8.1. While all nodes have been covered in testing, all stimuli (and consequently all arcs) have not. Further testing is required to cover these (perhaps a coverage test; see Section 6.3).



Model Statistics

Node Count	7 nodes
Arc Count	17 arcs
Stimulus Count	10 stimuli
Test Cases Recorded	10 cases
Nodes Generated	7 nodes / 7 nodes (1)
Arcs Generated	14 arcs / 17 arcs (0.823529412)
Stimuli Generated	9 stimuli / 10 stimuli (0.9)
Nodes Executed	7 nodes / 7 nodes (1)
Arcs Executed	14 arcs / 17 arcs (0.823529412)
Stimuli Executed	9 stimuli / 10 stimuli (0.9)

Figure 8.1: Model Test Counts

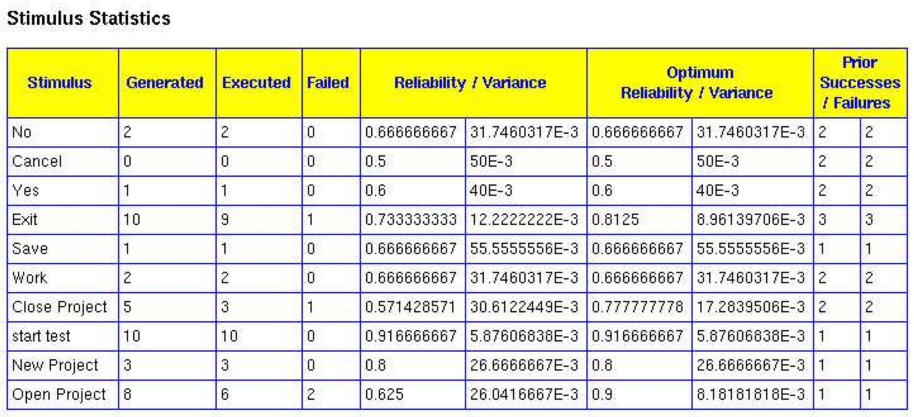
Stimulus Statistics

The stimuli of the model (unique arc labels) are listed down the left-hand column of the table, with each stimulus' statistics listed across the row. Of particular interest is the stimulus reliability, which is the long-run fraction of occurrences of the stimulus which are expected to be successful in the field, based on the current testing information.

An example of the stimulus section is shown in Figure 8.2. The lowest reliabilities (0.5) are associated with the stimuli which have never been executed. Of the stimuli which have been executed under test, the least reliable (0.57) is "Close Test," which has been executed three times and failed once. This stimulus has been generated five times; this implies that one execution never occurred under test, perhaps because it was blocked by another failure.

Arc Statistics

The entire model is reproduced in this section. For each arc of the model statistics are listed across the table. The probability of choosing a particular arc given the cur-



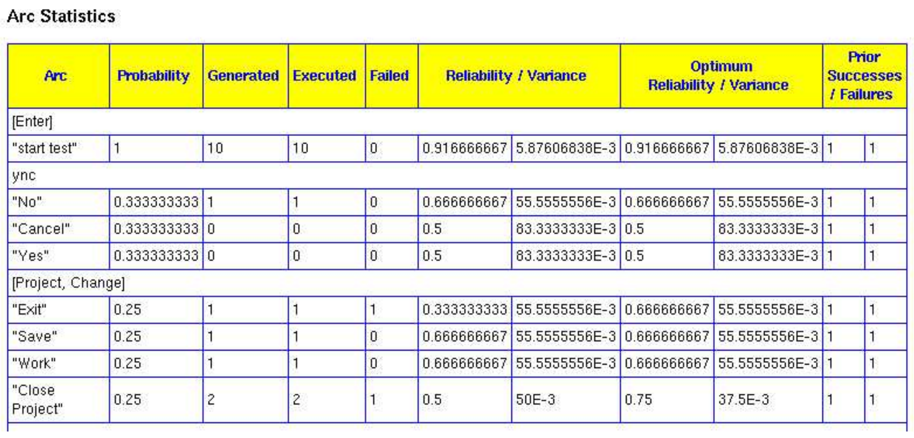
Stimulus Statistics

Stimulus	Generated	Executed	Failed	Reliability / Variance		Optimum Reliability / Variance		Prior Successes / Failures	
No	2	2	0	0.666666667	31.7460317E-3	0.666666667	31.7460317E-3	2	2
Cancel	0	0	0	0.5	50E-3	0.5	50E-3	2	2
Yes	1	1	0	0.6	40E-3	0.6	40E-3	2	2
Exit	10	9	1	0.733333333	12.2222222E-3	0.8125	8.96139706E-3	3	3
Save	1	1	0	0.666666667	55.5555556E-3	0.666666667	55.5555556E-3	1	1
Work	2	2	0	0.666666667	31.7460317E-3	0.666666667	31.7460317E-3	2	2
Close Project	5	3	1	0.571428571	30.6122449E-3	0.777777778	17.2839506E-3	2	2
start test	10	10	0	0.916666667	5.87606838E-3	0.916666667	5.87606838E-3	1	1
New Project	3	3	0	0.8	26.6666667E-3	0.8	26.6666667E-3	1	1
Open Project	8	6	2	0.625	26.0416667E-3	0.9	8.18181818E-3	1	1

Figure 8.2: Stimulus Test Statistics

rent node (called the “single-step” probabilities) is given next, as determined from the constraints.

An example of the arc section is shown in Figure 8.3. In this case the lowest reliability (0.33) is associated with an arc which has failed the only time it was executed.



Arc Statistics

Arc	Probability	Generated	Executed	Failed	Reliability / Variance		Optimum Reliability / Variance		Prior Successes / Failures	
[Enter]										
"start test"	1	10	10	0	0.916666667	5.87606838E-3	0.916666667	5.87606838E-3	1	1
ync										
"No"	0.333333333	1	1	0	0.666666667	55.5555556E-3	0.666666667	55.5555556E-3	1	1
"Cancel"	0.333333333	0	0	0	0.5	83.3333333E-3	0.5	83.3333333E-3	1	1
"Yes"	0.333333333	0	0	0	0.5	83.3333333E-3	0.5	83.3333333E-3	1	1
[Project, Change]										
"Exit"	0.25	1	1	1	0.333333333	55.5555556E-3	0.666666667	55.5555556E-3	1	1
"Save"	0.25	1	1	0	0.666666667	55.5555556E-3	0.666666667	55.5555556E-3	1	1
"Work"	0.25	1	1	0	0.666666667	55.5555556E-3	0.666666667	55.5555556E-3	1	1
"Close Project"	0.25	2	2	1	0.5	50E-3	0.75	37.5E-3	1	1

Figure 8.3: Arc Test Statistics

8.3 Using Prior Information

The model used to compute the reliabilities presented is Bayesian, and it is possible to include *a priori* information about the expected behavior of the system. This information may be used to take advantage of system execution information from other sources, such as prior field use. The details of how this information may be obtained are beyond the scope of this document; this section simply tells how to specify a prior number of failures and a prior number of successes for each arc.

By default, the JUMBL uses the “no prior information” case, which sets both priors to one for each arc, and then accumulates them for the stimulus priors. The prior number of successes and prior number of failures can be overridden by specifying distributions named successes and failures, respectively. Figure 8.4 shows a model in which the default prior successes have been increased to ten, with the prior successes for some arcs increased to twenty. Since no failures distribution is given, the default of one still applies.

```
successes:($ assume(10) $)
model ync_test
[Enter] "start test" [No Project]

[No Project]
    "Open Project"  [Project, No Change]
    "Exit"          [Exit]
    "New Project"   [Project, Change]

[Project, No Change]
  successes:($ 20 $)
    "Close Project" [No Project]
    "Exit"          [Exit]
    "Work"          [Project, Change]

[Project, Change]
  successes:($ 20 $)
    "Close Project" ync
    select "Yes"    [No Project]
           "No"     [No Project]
           "Cancel" [Project, Change]
    end
    "Work"          [Project, Change]
    "Exit" ync
    select "Yes"    [Exit]
           "No"     [Exit]
           "Cancel" [Project, Change]
    end
    "Save"          [Project, No Change]
end
```

Figure 8.4: Setting Priors

The default priors and the names of the distributions to use for priors are themselves configuration options (see Section 2.3).

- [Analysis.defaultPriorFailures]
  The prior number of failures to assume.
- [Analysis.defaultPriorSuccesses]
  The prior number of successes to assume.
- [Analysis.priorFailuresKey]
  The key for the distribution which specifies prior failures.
- [Analysis.priorSuccessesKey]
  The key for the distribution which specifies prior successes.

8.4 Using Test Analysis Engines

The `Analyze` command provides for the use of analytical engines (see Section 5.4 for a discussion of analytical engines). To see the list of test analysis engines which are available for use, use the `--list` switch with the `Analyze` command.

```
$ jumbl Analyze --list
Engines for Model Analysis (with -M):
    Simple
    Fast
Engines for Test Record Analysis (with -T):
    Simple
```

The first engine listed for each kind of analysis is the default analytical engine for that analysis. To override the default for model analyses, specify the engine using the `--model-engine` or `-M` switches. To override the default for test record analyses, specify the engine using the `--test-engine` or `-T` switches. For example, to specify the Simple engine (currently the only choice) for analyzing the ync_test model, use:

```
$ jumbl Analyze ync_test.str -T Simple
Scheduling test record ync_test.str.
Analyzing scheduled test records.
Writing analysis for model ync_test.
```

Current versions of the JUMBL provide only one test analysis engine: Simple. The Simple engine computes results analytically and includes variances where appropriate.

8.5 Performance

For models which are not small (800 states or fewer) it may be necessary to increase the memory available to the Java virtual machine (VM) in order to perform a test analysis. See Section 5.5 for more information.

Bibliography

[1] J. G. Kemmeny and J. L. Snell, *Finite Markov Chains*, Springer-Verlag: New York, NY, 1976.
This is the classic text on Markov chains and related computations. The Markov chain usage models used here take advantage of some properties of ergodic chains and some properties of absorbing chains in order to obtain all the results.

[2] W. Feller, *An Introduction to Probability Theory and Its Applications*, vol. 1, 3rd ed., John Wiley and Sons, Inc.: New York, NY, 1950.
This is the classic text on probability theory, and it covers the computation of expectations and variances from probability distributions.

[3] G. W. Snedecor and W. G. Cochran, *Statistical Methods* (Eighth ed.), Iowa State University Press: Ames, IA, 1989.
This is the latest edition of the classic text on statistical methods.

[4] R. B. Ash, *Information Theory*, Dover Publications, Inc.: New York, NY, 1990.
This text presents several results in information theory which are relevant to Markov chain usage models, including especially the source entropy of the chain.

[5] J.A. Whittaker and J.H. Poore, "Markov Analysis of Software Specifications," *ACM Transactions on Software Engineering and Methodology*, January 1993, pp. 93-106.
This paper introduced the use of Markov chain usage models to capture information about intended or expected software use and develop tests accordingly.

[6] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murriel, and J. M. Voas, "Estimating the Probability of Failure When Testing Reveals No Failures," *IEEE Transactions on Software Engineering*, v. 18, n. 1, January 1992, pp. 33-44.
This paper introduces the Bayesian reliability model adapted for use with Markov chain usage models in the JUMBL.

[7] K. D. Sayre and J. H. Poore, "Stopping Criteria for Statistical Testing," *Information and Software Technology*, 1 September 2000, v. 42, n. 12, pp. 851-7.
This paper discusses the use of the discriminant and other measures to determine when to stop testing, and presents its results in the context of Markov chain usage models.

[8] G. H. Walton, “Generating Transition Probabilities for Markov Chain Usage Models,” Ph.D. Dissertation, The University of Tennessee: Knoxville, TN, May 1995. Dr. Walton’s research introduced the use of constraints to specify transition probabilities in Markov chain usage models.

[9] S. J. Prowell, C. J. Trammell, R. C. Linger, and J. H. Poore, *Cleanroom Software Engineering: Technology and Process*, Addison-Wesley: Reading, MA, 1999. This text discusses Markov chain usage models, and uses an early version of TML.

[10] S. J. Prowell, “TML: A Description Language for Markov Chain Usage Models,” *Information and Software Technology*, 1 September 2000, v. 42, n. 12, pp. 835-44. This paper introduced TML, a subset of which is implemented by the JUMBL.

[11] Jeffrey E. F. Friedl, *Mastering Regular Expressions*, O’Reilly and Associates, 1997.

Appendix A

The MML Format

The Model Markup Language (MML) format is an XML extension for describing usage models. This format can be read and generated by the JUMBL, and is useful for interchange with other tools. This appendix describes the MML format. A reference for the MML format is available on the web:

```
http://www.cs.utk.edu/sqrl/esp/jumbl4/mml/mml.html
```

Note that, with the exception of the tags which specify names and labels, whitespace in MML files is ignored. Further, the order of elements at a given level in the document (with the exception of <trajectory> ... </trajectory> elements) is unimportant.

MML files may contain XML comments, which are ignored during processing. These start with `<!--` and end with `-->`. The following is an example comment.

```
<!-- The ync_test model, reproduced in MML format.
     This model occurs as figure 5.1 in the
     JUMBL User's Guide. -->
```

A.1 Basic Model Structure

An MML document describes a single model. The model is defined between the starting <model> tag and the ending </model> tag, neither of which takes any attributes.

Every model should have a name which consists only of letters, digits, and the underscore. Names which do not fit this pattern are subject to modification by tools which read or generate MML. Note that the model name should match the base name of the file. The extension for MML files is .mml. For example, the following defines an empty model ync_test, and would appear in the file `ync_test.mml`.

```
<model>
<name>ync_test</name>
</model>
```

States of the model are defined within <state> ... </state> pairs. A model may contain zero or more states, and each state must have a name defined inside <name> ... </name> pair, similar to the model. There are no restrictions on state names; they may contain whitespace, multiple lines, and other characters. Characters significant to XML such as less-than (<), greater-than (>), and the ampersand (&) should be escaped as `&lt;`, `&gt;`, and `&amp;`, respectively.

The <state> tag supports two attributes: source and sink. If a state is the unique model source, it should have the source attribute set to true. If a state is the unique model sink, it should have the sink attribute set to true. As with all models, no state may be both the model sink and source, and a model may have at most one source and one sink. Unlike TML, there is no implicit source or sink; they must be explicitly declared, if present. The following adds five states to the ync_test model defined earlier.

```
<model>
<name>ync_test</name>
<state source="true">
    <name>Enter</name>
</state>
<state sink="true">
    <name>Exit</name>
</state>
<state><name>No Project</name></state>
<state><name>Project, No Change</name></state>
<state><name>Project, Change</name></state>
</model>
```

Finally arcs must be added. Every arc has a source, a target, and a name. Arcs are specified similarly to models and states; their definition is given inside an <arc> ... </arc> pair, and they have a name given within a <name> ... </name> pair. As with states, arc names may include any characters.

Arcs must specify their source and target. The source may be implicitly specified by including the arc definition inside a state definition (inside a <state> ... </state> pair). The arc is then an outgoing arc from the specified state. Alternately, the arc may be outside of any state definitions, and use the <from> ... </from> pair to specify the originating state name. Finally, the target of the arc must be specified. This is done by placing the state name of the target state inside a <to> ... </to> pair. The following shows several arcs added to the ync_test model. Some are added inside state definitions, some only inside the model definition.

```
<model>
<name>ync_test</name>
<state source="true">
    <name>Enter</name>
    <arc>
```

```
            <name>start test</name>
            <to>No Project</to>
        </arc>
    </state>
    <state sink="true">
        <name>Exit</name>
    </state>
    <state>
        <name>No Project</name>
        <arc>
            <name>Open Project</name>
            <to>Project, No Change</to>
        </arc>
        <arc>
            <name>Exit</name>
            <to>Exit</to>
        </arc>
        <arc>
            <name>New Project</name>
            <to>Project, Change</to>
        </arc>
    </state>
    <state><name>Project, No Change</name></state>
    <state><name>Project, Change</name></state>
    <arc><name>Close Project</name>
         <from>Project, No Change</from>
         <to>No Project</to></arc>
    <arc><name>Exit</name>
         <from>Project, No Change</from>
         <to>Exit</to></arc>
    <arc><name>Work</name>
         <from>Project, No Change</from>
         <to>Project, Change</to></arc>
    <arc><name>Work</name>
         <from>Project, Change</from>
         <to>Project, Change</from></arc>
    <arc><name>Save</name>
         <from>Project, Change</from>
         <to>Project, No Change</to></arc>
    </model>
```

A.2 Referencing Other Models

Just as with TML, one may reference other models in MML. The structure of such a model reference is similar to TML, and selectors and trajectories may be used. Unlike

TML, an explicit selector must always be used.

To reference a model instead of another state, replace the <to> ... </to> pair with a <model> ... </model> pair, and specify the model name inside the tags (the model name follows the same restrictions as were given in Section A.1). In addition to the <model> ... </model> tags, an explicit selector must be given as a set of arcs inside <select> ... </select> tags. It is possible to specify a default arc in a selector: simply set the default attribute to true. A default arc may omit its name and thus will contain no <name> ... </name> pair. At most one arc in a selector may be marked as default.

The following example shows how to represent a simple model reference with a default next state in both TML and MML.

In TML:
```
"Close Project" ync [No Project]
```

In MML:
```
<arc>
    <name>Close Project</name>
    <model>ync</model>
    <select>
        <arc default="true">
            <to>No Project</to>
        </arc>
    </select>
</arc>
```

The following shows how to handle the case in which a selector has more than one choice. Note that, again, there is a default selection.

In TML:
```
"Close Project" ync
select "Yes"    [No Project]
       "No"     [No Project]
       default  [Project, Change]
end
```

In MML:
```
<arc>
    <name>Close Project</name>
    <model>ync</model>
    <select>
        <arc><name>Yes</name><to>No Project</to></arc>
        <arc><name>No</name><to>No Project</to></arc>
        <arc default="true"><to>Project, Change</to></arc>
    </select>
</arc>
```

Trajectories may also be included in a model reference. To use a trajectory, give each event of the trajectory inside a <trajectory> ... </trajectory> pair, and include the trajectory elements in the arc definition (at the same level as the <model> ... </model> and <select> ... </select> tags) in order. The following shows how one might use a trajectory in equivalent TML and MML.

In TML:
```
"Work" do_work."modified"."checked" [Project, Change]
```

In MML:
```
<arc>
    <name>Work</name>
    <model>do_work</model>
    <trajectory>modified</trajectory>
     <trajectory>checked</trajectory>
    <select>
        <arc default="true"><to>Project, Change</to></arc>
    </select>
</arc>
```

A.3 Using Labels and Constraints

Labels and constraints may be specified in MML. ("Values" may also be specified with <value> ... </value> pairs, but their use is beyond the scope of this document.) These are specified by giving the content of the label or constraint inside <label> ... </label> and <constraint> ... </constraint> pairs, respectively. Both take a single attribute: key, which specifies the label or constraint key. As with TML, constraints which are part of the default distribution do not require a key (though it may be specified as the empty string). Both constraints and labels are attached to model, state, or arc simply by including them in the definition. As with state and arc names, certain characters must be escaped if present; see A.1.

The following shows a section of the model from Figure 5.1 reproduced with constraints.

```
<model>
<constraint key="">assume(1)</constraint>
<constraint key="highw">fill(100) h=80</constraint>
<name>ync_test</name>
<state source="true">
    <name>Enter</name>
    <arc>
        <name>start test</name>
        <to>No Project</to>
    </arc>
</state>
...
```

```
<arc><name>Work</name>
     <constraint key="highw">h</constraint>
     <from>Project, No Change</from>
     <to>Project, Change</to></arc>
</model>
```

Similarly, the following shows a section of the model from Figure 6.8 as it would appear in MML. Note that the odd formatting of the <label> ... </label> blocks is necessary, since whitespace within a label is significant.

```
<model>
<name>phone</name>
<label key="a"> #include "phonetest.h"
 int main() {
     // Temporary phone number.
     int n = 0;
     // Initialize the test.
     initialize_test();
</label>
<state source="true">
    <name>On Hook</name>
    <arc>
        <name>lift receiver</name>
        <label key="a">     // Lift the receiver.
     lift($t);
</label>
        <to>Off Hook</to>
    </arc>
    <arc>
        <name>incoming call</name>
        <label key="a">     // Call this phone.
     call($t);
</label>
        <to>Ringing</to>
    </arc>
</state>
<state>
    <name>Off Hook</name>
    <arc>
        <name>hang up</name>
        <to>Exit</to>
        <label key="a">     // Hang this phone up.
     hangup($t);
</label>
    </arc>
    <arc>
```

```
        <name>dial busy</name>
        <to>Busy Tone</to>
        <label key="a">     // Locate a busy phone and call it.
      n=findbusy();
      dial($t, n);
</label>
    </arc>
</state>
...
<state sink="true">
    <name>Exit</name>
    <label key="a">     // Finished with the test.
      close_test();
 }
</label>
</state>
</model>
```

Appendix B

The TCML Format

The Test Case Markup Language (TCML) format is an XML extension for describing test records. This format can be read and generated by the JUMBL, and is useful for interchange with other tools. This appendix describes the TCML format. A reference for the TCML format is available on the web:

```
http://www.cs.utk.edu/sqrl/esp/jumbl4/tcml/tcml.html
```

Note that, with the exception of tags specifying event names, whitespace in MML files is ignored. MML files may contain XML comments, which are ignored during processing. These start with `<!--` and end with `-->`. The following is an example comment.

```
<!-- This test case is generated from the phone
     model shown in figure 6.5 of the
     JUMBL User's Guide. -->
```

B.1 Basic Test Case Structure

A TCML document describes a single test case. The test case is defined between the starting <testcase> tag and the ending </testcase> tag. A test case may consist of one or more trajectories (see Section 6.7), and each trajectory must be defined by a <trajectory> ... </trajectory> pair. The trajectory element takes a single attribute: id. The value of the id attribute must be a nonnegative integer, and this will be used to identify the events which belong to the trajectory.

Within the <trajectory> ... </trajectory> pair the model, distribution, and method used to generate the test case must be specified. The model name must consist only of letters, digits, and the underscore, and occurs inside a <model> ... </model> pair. The distribution is identified by its key (see Section 5.1) and is given in a <key> ... </key> pair. If there is no relevant distribution, or if the default distribution was used, the <key> ... </key> pair may be replaced with the empty tag <key/>. The method should be one of the strings “random”, “weight”, “coverage”, or “crafted”.

Every event of the trajectory is given within an <event> ... </event> pair. Each event tag takes a trajectory attribute, whose value must match the value of the id attribute of a trajectory element. The following is the test case from Figure 7.2, reproduced as equivalent TCML (but excluding test execution information).

```
<testcase>
    <trajectory id="0">
        <model>ync_test</model>
        <key />
        <method>random</method>
    </trajectory>
    <event trajectory="0">start test</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Work</event>
    <event trajectory="0">Work</event>
    <event trajectory="0">Close Project</event>
    <event trajectory="0">No</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Close Project</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Close Project</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Exit</event>
</testcase>
```

Note that every event has a trajectory attribute which refers to the trajectory defined at the top.

B.2 Using Multiple Trajectories

Using multiple trajectories (see Section 6.7) in TCML is straightforward. Each trajectory is defined by a <trajectory> ... </trajectory> pair, and must be given a unique id. Events then refer to the correct trajectory id in their trajectory attributes. The following reproduces the test case from Figure 6.6 as TCML.

```
<testcase>
    <trajectory id="1">
        <model>phone</model>
        <key/>
        <method>random</method>
    </trajectory>
    ...
    <trajectory id="99">
        <model>phone</model>
        <key/>
        <method>random</method>
    </trajectory>
    <event trajectory="62">incoming call</event>
    <event trajectory="65">incoming call</event>
    <event trajectory="92">lift receiver</event>
    <event trajectory="90">incoming call</event>
    <event trajectory="68">lift receiver</event>
    <event trajectory="22">incoming call</event>
    ...
    <event trajectory="0">hang up</event>
    <event trajectory="91">dial bad</event>
    <event trajectory="20">hang up</event>
    <event trajectory="91">hang up</event>
    <event trajectory="43">hang up</event>
</testcase>
```

B.3 Including Execution Information

There are two pieces of test information which can be recorded in TCML documents: the last step executed under test, specified by the laststep attribute of the testcase element, and a failure at a given step, specified by setting the fail attribute for the corresponding event element to true.

The following is the test case from Figure 7.2, reproduced as TCML with failure information included.

```
<testcase laststep="7">
    <trajectory id="0">
        <model>ync_test</model>
        <key />
        <method>random</method>
    </trajectory>
    <event trajectory="0">start test</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Work</event>
    <event trajectory="0">Work</event>
    <event trajectory="0" fail="true">Close Project</event>
    <event trajectory="0">No</event>
    <event trajectory="0" fail="true">Open Project</event>
    <event trajectory="0">Close Project</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Close Project</event>
    <event trajectory="0">Open Project</event>
    <event trajectory="0">Exit</event>
</testcase>
```

Appendix C

The EMML Format

Extended Model Markup Language (EMML) is a powerful way to describe complex usage models in terms of the underlying usage variables. Whereas one typically describes usage models in terms of states and arcs (see Chapter 4), with EMML one describes usage models in terms of assertions about allowable states and arcs.

Use of EMML is recommended only for experienced users, and even then only when the usage being modeled is well-understood in terms of underlying usage variables. It is difficult to create a "bad" model in TML (one which cannot be analyzed, which is too big to work with, or which isn't what is intended). It is much easier to do this with EMML. EMML allows you to define very complex models succinctly, but it also provides many opportunities to get things wrong. Use EMML with caution.

C.1 Overview: High-Level Models

One can think of use of a system in terms of *usage variables*. These are simply stochastic variables, or variables whose value changes over time in accordance with a probability distribution based on prior values. Each usage variable has a finite set of possible values. The cross product of finitely-many usage variables gives the set of potential *states*, some or all of which correspond to realizable states of use.

For example, one may have a usage variable "Status" for a project, whose values are "empty," "saved," and "unsaved." The usage event of saving the project may only be possible when Status is equal to unsaved. When this usage event occurs, the usage variable Status is set to saved.

A particular mapping of variables to values is a *state*. Two states are distinct if they map at least one usage variable to distinct values. A transition between two states labeled with a usage event is an *arc*. All arcs from a particular state must be labeled with distinct usage events. A high-level model describes this structure in terms of tests (predicates) on and changes (transforms) to usage variable values.

A *high-level model* consists of high-level states and high-level arcs. A *high-level state* is a predicate on usage variables, like "Status equals unsaved." A *high-level arc* consists of a usage event, like "save the project," and a transform (assignments), like "set Status to saved."

Each high-level state has an associated collection of high-level arcs. High-level states need not be disjoint; a high-level state may describe any number of model states. For example, the high-level state "Status equals unsaved" says nothing about other usage variables, and may thus refer to many distinct model states. When the predicate is true, the associated high-level arcs are "enabled," and the corresponding usage event may occur with the associated change in usage variable values.

The *source* for the model is determined by stating an initial value for each usage variable. The *sink* is determined by a special predicate; all states for which the predicate is true are regarded as the same sink state.

Certain assignments of values to usage variables may make no sense. For a telephone, one might have a "Switchhook" variable with the values "on" and "off," and another "Tone" variable with values "none," "dial," "busy," "ring," and "error." The combination "Switchhook equals on" and "Tone equals busy" makes no sense, because the phone cannot receive a tone when it is on-hook. To eliminate such states, one creates *assertions*. For example, one might exclude the states just considered with the assertion "If Switchhook equals on, then Tone equals none." This eliminates any state in which Switchhook is on and Tone is something other than none.

C.2 Declaring a High-Level Model

Every EMML document describes a single high-level model. The top-level element of every EMML document must be <hlmodel>, with a name attribute specifying the name of the model being described. For example, the following defines a model called "newphone:"

```
<hlmodel name="newphone">
...
</hlmodel>
```

This declaration should occur inside a file whose name matches the model name, with the extension .emml. The example model would be declared in the file newphone.emml.

The <hlmodel> element contains declarations of sets, usage variables, high-level states, the sink predicate, and assertions.

C.3 Declaring Sets and Usage Variables

Sets are unordered collections of distinct values, and they are very important to high-level models. Sets are declared by <set> elements, with each element given by the value attribute of an enclosed <element> element. For example, a boolean set can be defined as:

```
<set>
    <element value="true"/>
```

```
        <element value="false"/>
    </set>
```

One could use this declaration every time a set is required, but it is simpler to give the set a name and then refer to the name later. This can be done by including a name attribute in the set declaration, as follows.

```
    <set name="bool">
        <element value="true"/>
        <element value="false"/>
    </set>
```

If this declaration is a child of the <hlmodel> element, then this set is available throughout the document by the name "bool."

Usage variables are declared by a <variable> element, with the variable name specified by a name attribute, the initial value given by an initial attribute, and the range of values given by either an enclosed <set> element, or a set attribute naming a previously-declared set (or both). The following defines two usage variables. One is the Tone usage variable discussed earlier (see Section C.1), and the other uses the bool set defined above.

```
    <variable name="Tone" initial="none">
        <set>
            <element value="none"/>
            <element value="dial"/>
            <element value="ring"/>
            <element value="busy"/>
            <element value="error"/>
        </set>
    </variable>

    <variable name="Connected" set="bool" initial="false"/>
```

If both a set element and set attribute are present, the two sets are combined (set union).

C.4 Predicates

Like sets, predicates are a fundamental part of an EMML file. Predicates are used to declare high-level states, the model sink, and assertions. This section describes the content and meaning of these predicates.

Simple Predicates

The simplest predicate is one which is always true or always false. These are specified as follows:

```
<true/>
<false/>
```

An empty predicate is occasionally useful (see Section C.5), and is synonymous with <true/>. Only slightly more complex is the equality predicate, declared as an <equal> element. The usage variable is specified by a variable attribute, the value by a value attribute. To specify the predicate "Status equals unsaved," use the following:

```
<equal variable="Status" value="unsaved"/>
```

The opposite meaning is achieved by using a <notequal> element. Assuming that Connected has the meaning given in Section C.3, the following two predicates have the same meaning:

```
<equal variable="Connected" value="true"/>
<notequal variable="Connected" value="false"/>
```

One can also check whether a usage variable's value is in (or not in) a given set with the <in> (or <notin>) element. The usage variable is specified by a variable attribute; the set can be given as an enclosed <set> element, or specified by a set attribute (or both). If both are used, they are merged (set union). The following checks whether the phone is receiving a busy or error tone:

```
<in variable="Tone">
    <set>
        <element value="busy"/>
        <element value="error"/>
    </set>
</in>
```

The opposite sense can be achieved with the <notin> element.

Logical Connectives

The sense of any predicate can be reversed by enclosing it in a <not> element. The following two predicates have the same meaning:

```
<notequal variable="Status" value="unsaved"/>
<not>
    <equal variable="Status" value="unsaved"/>
</not>
```

Two or more predicates can be combined with <or>, so that the composite is true if and only if any of the enclosed predicates is true. This is sometimes called a *join*, or *logical or*. Two or more predicates can be combined with <and>, so that the composite is true if and only if all enclosed predicates are true. This is sometimes called a *meet*, or *logical and*. Both <and> and <or> are extremely useful for specifying high-level states. The following predicate is true if and only if the phone is off hook and receiving either the busy tone or the error tone (see Section C.3):

```
<and>
    <equal variable="Switchhook" value="off"/>
    <in variable="Tone">
        <set>
            <element value="busy"/>
            <element value="error"/>
        </set>
    </in>
</and>
```

Exactly two predicates can be combined such that the composite is true if and only if either the first is false, or the second is true, or both. This may seem a bit odd, but it is the usual formal meaning of “if.” In Section C.1, the assertion “if Switchhook equals on, then tone equals none” is used. This can be expressed as follows:

```
<implies>
    <equal variable="Switchhook" value="on"/>
    <equal variable="Tone" value="none"/>
</implies>
```

This predicate is equivalent to the following:

```
<or>
    <notequal variable="Switchhook" value="on"/>
    <equal variable="Tone" value="none"/>
</or>
```

In general, any expression of the form “if *P* then *Q*” can be expressed as <implies>*PQ*</implies>. The <implies> element is most often used for assertions.

If several predicates appear in sequence (inside a <predicate> element, for instance; see Section C.5), their action is as if they were enclosed in an <and> element.

C.5 Declaring High-Level States

High-level states are declared by an <hlstate> element, which contains a predicate and some number of high-level arcs. The predicate for a high-level state is declared in a <predicate> element.

Each high-level arc is declared in an <hlarc> element. The usage event is specified in either a name attribute or inside an enclosed <name> element (if the usage events contains characters which cannot easily be represented in attribute values). Transformations (if any) in usage variable values are declared by enclosed <assign> elements; the usage variable is specified by a variable attribute, and the new value is specified by a value attribute.

The following high-level state declaration might be used as part of a telephone model:

```
<hlstate>
    <predicate>
        <equal variable="Switchhook" value="off" />
    </predicate>
    <hlarc name="Hang Up Phone">
        <assign variable="Switchhook" value="on" />
        <assign variable="Tone" value="none" />
        <assign variable="Connected" value="false" />
    </hlarc>
</hlstate>
```

In this case the high-level state applies whenever the phone is off hook, and enables one usage event, “Hang Up Phone.” When the phone is hung up, Switchhook is changed to on, Tone is changed to none, and Connected is changed to false.

Assignments can be made conditional by enclosing a predicate in the <assign> element. The following allows the switch to be toggled from any state (it is a high-level state matching any state, one usage event, and two conditional assignments):

```
<hlstate>
    <predicate/>
    <hlarc name="Toggle Switch">
        <assign variable="Switch" value="on">
            <equal variable="Switch" value="off"/>
        </assign>
        <assign variable="Switch" value="off">
            <equal variable="Switch" value="on"/>
        </assign>
    </hlarc>
</hlstate>
```

C.6 Declaring the Sink

Every model must have a unique sink state. The sink is declared via a predicate. All states which match the predicate are merged into a single sink state. Obviously the source state cannot match the sink state predicate. The predicate for the sink state is declared in a <sink> element.

The following example states that use is complete when the phone has connected and is on hook.

```
<sink>
    <equal variable="HasConnected" value="true"/>
    <equal variable="Switchhook" value="on"/>
</sink>
```

C.7 Declaring Assertions

Not all states are possible, and it is not always easy to write high-level states and high-level arcs such that impossible states are never reached. To account for this one can use assertions. These are predicates enclosed in <assert> elements which must be true for all states in the model. If an assertion fails for any state, the state is silently discarded.

The following asserts that a phone can only ring when on hook:

```
<assert>
    <implies>
        <equal variable="Ringing" value="true"/>
        <equal variable="Switchhook" value="on"/>
    <implies>
</assert>
```

C.8 Nested Elements

One can nest high-level states and assertions inside other high-level states. Let the predicate of the enclosing state be *P*. Then the predicate *Q* of each enclosed high-level state or assertion is replaced with *P* implies *Q*. This nesting can be to arbitrary depth, and allows factoring common elements out of predicates.

It may be reasonable to break usage into several modes, and create one high-level state for each mode. Then one can describe the state space inside each high-level state. The following shows one example, with a nested state near the end:

```
<!-- Phone is on hook. -->
<hlstate>
    <predicate>
        <equal value="on" variable="SwitchHook"/>
    </predicate>
    <hlarc name="Lift Receiver">
        <assign value="off" variable="SwitchHook"/>
        <assign value="dial" variable="Tone">
            <equal value="false" variable="Ringing"/>
        </assign>
        <assign value="true" variable="Connected">
            <equal value="true" variable="Ringing"/>
        </assign>
        <assign value="true" variable="HasConnected">
            <equal value="true" variable="Ringing"/>
        </assign>
        <assign value="false" variable="Ringing">
            <equal value="true" variable="Ringing"/>
        </assign>
    </hlarc>
    <!-- Note that multiple Receive Call events can
```

```
          occur before answered. -->
     <hlarc name="Receive Call">
         <assign value="true" variable="Ringing"/>
     </hlarc>
     <hlstate>
         <predicate>
             <equal value="true" variable="Ringing"/>
         </predicate>
         <hlarc name="Caller Hang Up">
             <assign value="false" variable="Ringing"/>
         </hlarc>
     </hlstate>
 </hlstate>
```

C.9 Constraints and Labels

Constraints and labels are added to EMML in much the same way they are added to MML (see Section A.3). Nested <constraint> and <label> elements can be included in <hlmodel>, <hlstate>, and <hlarc> elements to attach constraints and labels. See Section C.12 for an example of this.

C.10 Implicit Variables

All usage variables must be declared, unless the variable name begins with an underscore (_). In this case, the usage variable is taken to be an *implicit* variable, and it need not be declared. Implicit variables are used in the same manner as regular usage variables, but their allowable values are deduced from the EMML source. The initial value of an implicit variable is the empty string. See Section C.12 for an example (only the exit arc and sink declaration use the implicit variable).

The general use of implicit variables is not recommended, since the compiler cannot check their values. If you misspell (or mis-capitalize) a value or even the variable name, the compiler cannot catch this and the error will be very difficult to track down.

C.11 Compiling EMML

Models written in EMML cannot be directly analyzed, nor can test cases be generated from them. First the model must be converted to a form such as TML (see Section 4.3) or SM (see Section 4.2). This is done with the `Write` command. The following will convert newphone.emml to TML:

```
$ jumbl Write -tTML newphone.emml
Writing: newphone.emml as TML.
```

State names in the generated model by default include both the usage variable name and the value, with usage variable names output in alphabetical order. For example, a state name might be:

```
[Connected=false; HasConnected=false; Ringing=false; SwitchHook=off; Tone=busy]
```

If the configuration option [HLModel.variableNames] (see Section 2.3) is set to false, the usage variable name is suppressed. The above state name would thus become:

```
[false; false; false; off; busy]
```

Since usage variable names are alphabetized, it is always possible to figure out what these state names mean. However, it is better if, given that one intends to suppress usage variable names, one uses descriptive values. This could give an alternative state name such as the following:

```
[unconnected; never; not_ringing; off_hook; busy_tone]
```

EMML can be used to describe huge models. For this reason, compiling EMML to SM or TML can take some time. To get some feedback, the [HLModel.showProgress] configuration option can be set to true (see Section 2.3):

```
$ jumbl Write -tTML telephone.emml
Writing: telephone.emml as TML.
Expanding: false; false; false; on; none
Expanding: false; false; false; off; dial
...
Expanding: false; false; true; on; none
```

It is also possible to use `Write` to convert a model to EMML. For example, the following would convert the model phone.tml to EMML:

```
$ jumbl Write -tEMML phone.tml
Writing: phone.tml as EMML.
```

C.12 An Example

EMML is particularly useful for models with a large number of very similar states. Consider a system with three pumps, each of which may be on or off, and three valves. Each valve may be in one of four conditions: open, closed, stuck open, or stuck closed. The system can operate in two modes: run mode, in which at most one pump can be on, and emergency mode, in which any number of the pumps may be on. A use begins with all valves closed and all pumps off, and ends randomly if all pumps are off and all valves either stuck or closed.

Declaring the system in terms of usage variables is simple. The following shows the declarations of the usage model. Note that a constraint is used to assume equally-likely probabilities everywhere (unless indicated otherwise) and to define some variables to be used elsewhere.

```
<hlmodel name="pv">
  <constraint key="">
    assume(1) stick=1/2 exit=1/100
  </constraint>
  <set name="pump">
    <element value="on"/>
    <element value="off"/>
  </set>
  <set name="valve">
    <element value="open"/>
    <element value="closed"/>
    <element value="stuck open"/>
    <element value="stuck closed"/>
  </set>
  <set name="unstuck">
    <element value="open"/>
    <element value="closed"/>
  </set>
  <variable name="P1" set="pump" initial="off"/>
  <variable name="P2" set="pump" initial="off"/>
  <variable name="P3" set="pump" initial="off"/>
  <variable name="V1" set="valve" initial="closed"/>
  <variable name="V2" set="valve" initial="closed"/>
  <variable name="V3" set="valve" initial="closed"/>
  <variable name="Mode" initial="run">
    <set>
      <element value="run"/>
      <element value="emergency"/>
    </set>
  </variable>
```

Any pump which is off may be turned on, and any pump which is on may be turned off.

```
  <hlstate>
    <predicate><equal variable="P1" value="on"/></predicate>
    <hlarc name="P1 off">
      <assign variable="P1" value="off"/>
    </hlarc>
  </hlstate>
  <hlstate>
```

```
    <predicate><equal variable="P2" value="on"/></predicate>
    <hlarc name="P2 off">
      <assign variable="P2" value="off"/>
    </hlarc>
  </hlstate>
  <hlstate>
    <predicate><equal variable="P3" value="on"/></predicate>
    <hlarc name="P3 off">
      <assign variable="P3" value="off"/>
    </hlarc>
  </hlstate>
  <hlstate>
    <predicate><equal variable="P1" value="off"/></predicate>
    <hlarc name="P1 on">
      <assign variable="P1" value="on"/>
    </hlarc>
  </hlstate>
  <hlstate>
    <predicate><equal variable="P2" value="off"/></predicate>
    <hlarc name="P2 on">
      <assign variable="P2" value="on"/>
    </hlarc>
  </hlstate>
  <hlstate>
    <predicate><equal variable="P3" value="off"/></predicate>
    <hlarc name="P3 on">
      <assign variable="P3" value="on"/>
    </hlarc>
  </hlstate>
```

One may send a signal to toggle a valve at any time, independent of the valve's status. Stuck valves do nothing.

```
  <hlstate>
    <predicate/>
    <hlarc name="V1 toggle">
      <assign variable="V1" value="open">
        <equal variable="V1" value="closed"/>
      </assign>
      <assign variable="V1" value="closed">
        <equal variable="V1" value="open"/>
      </assign>
    </hlarc>
    <hlarc name="V2 toggle">
      <assign variable="V2" value="open">
        <equal variable="V2" value="closed"/>
      </assign>
      <assign variable="V2" value="closed">
        <equal variable="V2" value="open"/>
      </assign>
    </hlarc>
    <hlarc name="V3 toggle">
      <assign variable="V3" value="open">
        <equal variable="V3" value="closed"/>
      </assign>
      <assign variable="V3" value="closed">
        <equal variable="V3" value="open"/>
      </assign>
    </hlarc>
  </hlstate>
```

A valve may become stuck at any time. It sticks in the current position, and this is only half as likely as the usual toggle.

```
<hlstate>
  <predicate>
    <in variable="V1" set="unstuck"/>
  </predicate>
  <hlarc name="V1 stuck">
    <constraint key="">stick</constraint>
    <assign variable="V1" value="stuck open">
      <equal variable="V1" value="open"/>
    </assign>
    <assign variable="V1" value="stuck closed">
      <equal variable="V1" value="closed"/>
    </assign>
  </hlarc>
</hlstate>
<hlstate>
  <predicate>
    <in variable="V2" set="unstuck"/>
  </predicate>
  <hlarc name="V2 stuck">
    <constraint key="">stick</constraint>
    <assign variable="V2" value="stuck open">
      <equal variable="V2" value="open"/>
    </assign>
    <assign variable="V2" value="stuck closed">
      <equal variable="V2" value="closed"/>
    </assign>
  </hlarc>
</hlstate>
<hlstate>
  <predicate>
    <in variable="V3" set="unstuck"/>
  </predicate>
  <hlarc name="V3 stuck">
    <constraint key="">stick</constraint>
    <assign variable="V3" value="stuck open">
      <equal variable="V3" value="open"/>
    </assign>
    <assign variable="V3" value="stuck closed">
      <equal variable="V3" value="closed"/>
    </assign>
  </hlarc>
</hlstate>
```

Now that the basic operation is accounted for, there just remains the problem of operating in the different modes (run versus emergency). This is handled with an assertion.

```
  <hlstate>
    <predicate>
      <equal variable="Mode" value="run"/>
    </predicate>
    <hlarc name="Go To Emergency Mode">
      <assign variable="Mode" value="emergency"/>
    </hlarc>
    <assert>
      <not>
        <and>
          <equal variable="P1" value="on"/>
          <equal variable="P2" value="on"/>
          <equal variable="P3" value="on"/>
        </and>
      </not>
    </assert>
  </hlstate>
  <hlstate>
    <predicate>
      <equal variable="Mode" value="emergency"/>
    </predicate>
    <hlarc name="Go To Run Mode">
      <assign variable="Mode" value="run"/>
    </hlarc>
  </hlstate>
```

Finally, the model can be exited randomly if all pumps are off and all valves are either closed or stuck. An implicit variable is used for this, and the sink is declared.

```
  <hlstate>
    <predicate>
      <and>
        <equal variable="P1" value="off"/>
        <equal variable="P2" value="off"/>
        <equal variable="P3" value="off"/>
        <notequal variable="V1" value="open"/>
        <notequal variable="V2" value="open"/>
        <notequal variable="V3" value="open"/>
      </and>
    </predicate>
    <hlarc name="Exit">
      <constraint key="">exit</constraint>
      <assign variable="_" value="Exit"/>
    </hlarc>
  </hlstate>
  <sink>
    <equal variable="_" value="Exit"/>
  </sink>
</hlmodel>
```

When this model is written to TML or SM, it produces a model with 961 states and 7,598 arcs, even though the EMML description is only 180 lines long. If the number of valves were doubled, then the total number of states would be multiplied by $4^3 = 64$, since each valve has four different states. Thus the total number of states would grow to (approximately) 61,441. One must be careful when using EMML so as not to create a model which is too large to be useful.

Index