

Scriptable User Interaction

Motama GmbH, Saarbruecken, Germany

<http://www.motama.com>

April 2010

Copyright (C) 2005-2010
Motama GmbH, Saarbruecken, Germany
<http://www.motama.com>

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with the Invariant Sections being all sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license can be found in the file COPYING.FDL.

THE DOCUMENT IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, TITLE AND NON-INFRINGEMENT. IN NO EVENT SHALL THE COPYRIGHT HOLDERS OR ANYONE DISTRIBUTING THE DOCUMENT BE LIABLE FOR ANY DAMAGES OR OTHER LIABILITY, WHETHER IN CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE DOCUMENT OR THE USE OR OTHER DEALINGS IN THE DOCUMENT.

This document describes the use of the scriptable user interaction framework. This framework can be used for testing interactive applications by logging timestamped user commands to interaction scripts and controlling interactive applications by interaction scripts.

1. Introduction

Interactive applications are sometimes hard to test and debug. Some bugs can only be reproduced by a specific sequence of user interactions, sometimes with specific timing. Furthermore, NMM applications may be distributed to multiple processes on different machines, and all of these processes may be interactive.

To make testing and debugging of interactive and distributed NMM applications easier, we provide a framework for scriptable user interaction. The framework can be easily integrated into existing applications. It can be used to record user interactions along with their timestamps into interaction scripts, and play back these interaction scripts on multiple processes using the same timing for interactive commands as during recording. It is also possible to write or modify interaction scripts manually in order to simulate certain sequences of user interaction. Timing and user commands can also be randomized to a certain extent.

Once the framework has been integrated into an existing application, additional command-line options are available which control recording or playback of interaction scripts. These command-line options are specified in addition to the normal command-line options of the application. Scriptable user interaction has already been integrated in the `serverregistry` application.

Usage of scriptable user interaction and all important command-line options are documented in Section 2. How to integrate scriptable user interaction into existing applications is described in Section 3. The specification of the interaction script syntax can be found in Section 4. A brief overview of the implementation of the scripting framework is given in Section 5.

2. Using Scriptable User Interaction

This section describes how to use the scriptable user interaction framework. In this release, scriptable user interaction is integrated only in the `serverregistry` application, so the examples throughout this section will use a hypothetical application called "your_application" which also provides scriptable user interaction. Please go to section Section 3 to see how to integrate scriptable user interaction into your applications.

Scripting functionality is available through command-line options that begin with three dashes ("---"). The command-line parser removes these options from the command-line before it is passed on to the actual application. Application arguments and scripting arguments can be placed in any order. If no scripting options are provided, the application behaves as usual. For a list of scripting options and their descriptions, run the command:

```
<APPLICATION> ---help
```

e.g.

```
your_application ---help
```

The remainder of this section describes common use cases.

2.1. Requirements

Scriptable user interaction requires that the clocks of all participating hosts are synchronized to a common time source. We recommend using the network time protocol (NTP) for synchronizing the clocks of all hosts.

2.2. Log interaction of a single process

General Syntax:

```
<APPLICATION> ---log-to=<FILENAME> ---identifier=<ID> <ARGS>
```

Example:

```
your_application ---log-to=your_application.log ---identifier=Test \  
  buffalo_soldier.mp3
```

In this example, the application `your_application` is run with the argument `buffalo_soldier.mp3`, and all user interaction is logged to the file `your_application.log` with the given identifier.

2.3. Playback with a single process

General Syntax:

```
<APPLICATION> ---play=<FILENAME>
```

Example:

```
your_application ---play=your_application.log buffalo_soldier.mp3
```

In this example, the application `your_application` is run with the arguments `buffalo_soldier.mp3`. No user interaction is possible. Instead, all user commands are read from the file `your_application.log` and executed at the same relative times as during recording.

Note: Some user commands in some applications may take a significant amount of time to execute, and this time may vary between multiple executions of the command. Often no other user interaction is possible while the command is in progress. An example for such commands are commands which trigger any series of blocking operations or distributed method calls.

It is possible that such a command takes more time to finish during playback than during recording. If the delay of the next command in the interaction script is too short, then the next command can

not be executed in time. Therefore, if a command is delayed for more than a tolerance interval of 50 milliseconds, the following warning is displayed:

```
WARNING: Command delayed for 1 sec, 575115000 nanosec
WARNING: Tolerated delay is 0 sec, 50000000 nanosec
WARNING: Maybe commands can not be executed as fast as requested?
```

When recording or writing interaction scripts, such situations can be avoided by making the pauses between commands long enough to allow the interaction to complete.

Even if a command is delayed and the above warning is displayed, the interaction script system will attempt to execute subsequent user interactions at the same time relative to the start time, as during recording. If there are pauses between user interactions, playback may eventually catch up. However, user interaction of multiple participating processes may be temporarily out of sync.

2.4. Log interaction of multiple processes

General Syntax:

```
host1> <APPLICATION> ---log-to=<FILENAME_1> \
  ---identifier=<ID> ---processes=<N>
host2> <APPLICATION> ---log-to=<FILENAME_2> \
  ---identifier=<ID> ---processes=<N>
...
hostN> <APPLICATION> ---log-to=<FILENAME_N> \
  ---identifier=<ID> ---processes=<N>
```

Example:

```
host1> serverregistry ---log-to=serverregistry.log \
  ---identifier=DistWavPlayback ---processes=2
host2> your_application ---log-to=your_application.log \
  ---identifier=DistWavPlayback ---processes=2 \
  buffalo_soldier.mp3 host1
```

In this example, the `serverregistry` application is started on `host1`, and the `your_application` application is started on `host2` with the arguments `buffalo_soldier.mp3 host1`. All user interaction is logged to the files `serverregistry.log` and `your_application.log` respectively.

Note: During recording, all processes run independently. There is no additional communication between participating processes for the purpose of recording user interaction.

Note: It is recommended to prepare the command-lines and start all processes within a short time interval. This avoids unnecessary long delays at start of playback. Of course, if it is necessary to wait for a certain time between starting the participating processes, this can be done, and the absolute start times of the processes will be recorded, so they can be reproduced during playback.

Note: You must specify the same identifier for all participating processes. The identifier is the name of the group of processes which are used in the test. It is an arbitrary string that describes the test, and avoids potential errors caused by using the wrong interaction scripts during playback or processes belonging to a different test connecting to the interaction script synchronization service.

2.5. Playback with multiple processes

General Syntax:

```
host1> <APPLICATION> ---play=<FILENAME_1> \  
    ---start-service [=<PORT>]  
host2> <APPLICATION> ---play=<FILENAME_2> \  
    ---sync-to=host1[:<PORT>]  
...  
hostN> <APPLICATION> ---play=<FILENAME_N> \  
    ---sync-to=host1[:<PORT>]
```

Example:

```
host2> your_application ---play=your_application.log ---start-service \  
    buffalo_soldier.mp3 host1  
host1> serverregistry ---play=serverregistry.log ---sync-to=host2
```

In this example, the `serverregistry` application is started on `host1`, and the `your_application` application is started on `host2` with the arguments `buffalo_soldier.mp3 host1`. No user interaction is possible, and all user commands are read from the files `serverregistry.log` and `your_application.log` respectively and executed at the same relative times as during recording.

Note: When playing back an interaction script, all processes synchronize their time offsets once when playback starts. After this, there is no additional communication between participating processes for the purpose of playing back user interaction. Synchronization of the timing of commands between different processes is achieved by assuming that the system clocks are synchronized to a common time source and by timing all user interactions relative to the initial time offset.

Note: The order in which processes are started during playback is arbitrary. Section 3 describes how to guarantee this property in custom applications. However, the process which is started first must always start the synchronization service (`--start-service`), and all other processes must connect to the synchronization service (`--sync-to`).

Note: The exact timing of starting the processes does not matter. The scriptable user interaction system will delay each process in order to synchronize their start times. For example, in a typical NMM application, this can be done immediately before starting the flowgraph. In this way, the timing of user interaction after starting the flowgraph is preserved, whereas variance in setup time of the flowgraph does not affect the timing during playback.

Section 3 describes how to guarantee this property in custom applications.

Note: The default port used is 20030. The default port can be changed as specified above. Please make sure that the chosen port is not blocked by a firewall.

2.6. Simultaneous Playback and logging

It is possible to log user interaction to a new interaction script during playback by simply using the necessary options for logging in addition to the options for playback.

Example:

```
host2> your_application ---play=your_application.log ---start-service \  
  ---log-to=your_application2.log ---identifier=LogAgain ---processes=2 \  
  buffalo_soldier.mp3 host1  
host1> serverregistry ---play=serverregistry.log ---sync-to=host2 \  
  ---log-to=serverregistry2.log ---identifier=LogAgain ---processes=2
```

In this example, the `serverregistry` application is started on `host1`, and the `your_application` application is started on `host2` with the arguments `buffalo_soldier.mp3 host1`. No user interaction is possible, and all user commands are read from the files `serverregistry.log` and `your_application.log` respectively and executed at the same relative times as during recording. All user interactions and their exact timings are recorded in the files `serverregistry2.log` and `your_application2.log` respectively.

Note: The `---identifier` and `---processes` options must be given again. This allows for the flexibility of logging only some of the participating processes.

2.7. Terminating Scripts

If the end of the interaction script is reached, normal user interaction is possible again. If recording is enabled, both the scripted commands at the beginning and the interactive commands at the end will be recorded. This functionality can be used for extending existing interaction scripts.

If you want to setup a batch of tests using scriptable user interaction, it is required to end a test to go on with the next test. In order to exit a playback of a user interaction script, you simply need to add the corresponding command that usually terminates the application during recording of the script. For example, if you want to terminate `your_application` in the end of performing a number of user interaction, stop recording the log by entering "q" and ENTER.

2.8. Looping Scripts

Some interaction scripts also support looping. These interaction scripts can perform a sequence of commands repeatedly for a number of times which can be given on the command-line. By default, the loop sequence is executed only once. After repeating the loop sequence for the given number of times, the remaining commands (if any) at the end of the script are executed. This is useful for long-term unit tests which perform a sequence of user interaction repeatedly and then exit.

For example, the option

```
---loop=10
```

causes the loop sequence of a script to be repeated 10 times. If a script does not support looping, the `---loop` option has no effect.

How to make a script loopable is described in Section 4.5.

If the `---loop` option is used with multiple processes, then these processes will not be synchronized again when the interaction script loops. Playback simply continues at the beginning of the file as if the first command was the next command. It is allowed to specify `---loop` for some participating processes and not specify `---loop` for other participating processes of the same group or use different numbers of iterations.

2.9. Echoing Commands

By default, all commands are echoed to standard output during playback. Also, if the end of the interaction script is reached, an appropriate message is displayed on standard output. All output of the interaction script reader can be suppressed by using the `---no-echo` option.

2.10. Starting multiple processes automatically

If a test involves multiple participating processes, starting all of these processes can become inconvenient and error-prone. Therefore, we provide a shell script named `ScriptPlay` that can be used for automatically starting multiple processes that play back corresponding interaction scripts. These interaction scripts may have been created manually or recorded from previous runs of the test. No such shell script is provided for recording.

`ScriptPlay` can be found in the directory

```
scripts/scriptplay.sh
```

If NMM has been installed, then `ScriptPlay` can be found in the `bin` subdirectory of the installation prefix. For example, if NMM was installed in `/home/bob/nmm`, then `ScriptPlay` will be named

```
/home/bob/nmm/bin/scriptplay.sh
```

`ScriptPlay` can only be run on Linux and requires all participating hosts to be either Linux machines or have a Unix-like environment with SSH access (such as Cygwin on Windows or Mac OS X). Furthermore, `ScriptPlay` requires the Bash shell to be installed as `/bin/bash` on all hosts. However, it is not necessary to use Bash as the login shell. On the host where the `ScriptPlay` is run, it must be possible to connect to all participating hosts of the test using SSH and without having to manually enter a password.

The general usage of `ScriptPlay` is

```
scriptplay.sh <SCRIPT_1> [<SCRIPT_2>] [...] [<SCRIPT_N>]
```

where `SCRIPT_1` to `SCRIPT_N` are the filenames of interaction scripts. For example, if you have two interaction scripts, `your_application.isc` and `serverregistry.isc`, you can run `ScriptPlay` as follows:

```
scriptplay.sh serverregistry.isc your_application.isc
```

`ScriptPlay` will run the programs specified by the `commandline` header field in each interaction script on the hosts specified by the `hostname` header field in each interaction script. Please see Section 4 for a detailed description of interaction scripts and their header fields. You can edit the interaction scripts manually if needed.

`ScriptPlay` uses information in optional header fields of interaction scripts. This information is normally written to each interaction automatically during recording. However, if you created interaction scripts manually, the following header fields must be present in each interaction script for `ScriptPlay` to work:

```
version=1
identifier=<Identifier>
processes=<Number of processes>
commandline=<Command line>
hostname=<Host name>
workingdirectory=<Working directory>
```

The following header fields are also used by ScriptPlay. However, they are optional:

```
username=<User name>
NMM_DEV_DIR=<Value of NMM_DEV_DIR environment variable>
LD_LIBRARY_PATH=<Value of LD_LIBRARY_PATH environment variable>
DYLD_LIBRARY_PATH=<Value of DYLD_LIBRARY_PATH environment variable>
PATH=<Value of PATH environment variable>
```

Note: The processes are started in the same order as the corresponding interaction scripts are specified on the command-line. The order in which processes are started is arbitrary, and therefore the order in which the filenames of interaction scripts are provided is also arbitrary.

ScriptPlay will display information about the processes that are started, the hosts where they are started and the files the output (standard output and standard error) of the processes will be written to. The output looks similar to

```
Starting process #1 on host host1
Command: ./test_serverregistry
Output file: /home/bob/nmm/installed/bin/output1.txt on host1

Starting process #2 on host host2
Command: ./your_application host1
Output file: /home/bob/nmm/installed/bin/output2.txt on host2
```

If ScriptPlay does not work (i.e. the processes that are part of the test are not started or do not behave as expected), please take a look at the output files first. If you need to do many test runs, you can also try to run the processes manually first (e.g. to see if your interaction scripts work and your setup meets the requirements for the test) before using ScriptPlay for running the processes automatically.

3. Integrating Scripting into Interactive Applications

This section describes how to integrate scriptable user interaction into existing applications.

- Include `InteractionScript.hpp`

```
#include "nmm/services/test/InteractionScript.hpp"
```

- Create an instance of `InteractionScript` (at the beginning of `main()`)

```
int main(int argc, char* argv[])
{
    InteractionScript script(argc, argv);

    // ...
}
```

Note that at this line, the scripting command-line options are parsed, and both `argc` and `argv` are modified such that these options are no longer contained within the command-line. The application can therefore parse its own command-line options without having to handle the scripting command-line options. However, the line above should be inserted before the position where the application parses its own command-line options.

- Decide at which point in the application, the process is ready for accepting connections from other processes. At this point, the process should wait for all other participating processes to be started when playing back a interaction script. This is achieved by inserting the following line into the application code:

```
script.prepare();
```

If an application does not accept connections from other processes (e.g. a typical "client" application, such as `your_application`), the `prepare` method can be called immediately after creating the instance of `InteractionScript`. However, if the application needs to open ports or register services, then `prepare` should be called after all ports have been opened and all services have been registered and are ready for use.

Typical command-line applications first parse command-line options and then disable NMM debug output based on whether a `-v` option was given. In such applications, `prepare` should not be called before disabling debug output, because otherwise the communication framework of NMM, which is used by `prepare`, will produce unwanted debug output.

As an example, a typical NMM application should call `prepare` immediately after command-line options have been parsed and debug output has been disabled and before the application starts creating a flowgraph. On the other hand, the `serverregistry` application calls `prepare` after the registry service has been fully initialized. This allows both processes to be started in arbitrary order when playing back an interaction script. The proper placing of the `prepare` call ensures that the registry service is initialized before an NMM application starts requesting a flowgraph from the registry.

- Decide at which point in the application logging and/or playback should begin. A good approach is to start logging and/or playback after initialization of a flowgraph but before the flowgraph is started.

Initialization can take some time that may vary between systems. By starting logging and/or playback after initialization, it is guaranteed that initialization does not affect the timing of scripted user commands. Furthermore, by starting logging and/or playback before the flowgraph is started guarantees that the timing of commands relative to the time when the flowgraph was started will be roughly the same during playback and recording.

Once the proper location has been determined, insert the following code at this location:

```
script.begin();
```

Example:

```
// ...

// The graph description
GraphDescription graph;

// ... initialize graph ...

graph.realizeGraph();
graph.activateGraph();

script.begin();

// Start the graph
graph.startGraph();

// ... user interaction happens here ...
```

- Find the statements in the application which read commands from standard input (or a different device) and enclose them in `BEGIN_SCRIPTING` and `END_SCRIPTING` macro calls.

General syntax:

```
BEGIN_SCRIPTING(script, command) {
    // Code for reading a command from an input device
    // and storing it in the variable "command"
    // (which must be a string)
}
END_SCRIPTING(script, command)

// Code for evaluating the command
```

Example: The following code could appear in any console-based NMM application. It reads a command from standard input and evaluates it:

```
string s;
// read a line from standard input
char buffer[256];
```

```

cin.getline(buffer, sizeof(buffer)-1);
s = buffer;

// create a string stream for parsing the input
string command;
istringstream is(s);
is >> command;

// parse the command typed by the user
if(command == "q") {
    // ...
}
else if(command == "+") {
    // ...
}
// ...

```

This would be changed to:

```

string s;
BEGIN_SCRIPTING(script, s) {
    // read a line from standard input
    char buffer[256];
    cin.getline(buffer, sizeof(buffer)-1);
    s = buffer;
}
END_SCRIPTING(script, s);

// create a string stream for parsing the input
string command;
istringstream is(s);
is >> command;

// parse the command typed by the user
if(command == "q") {
    // ...
}
else if(command == "+") {
    // ...
}
// ...

```

If commands in your application are not strings which are read from the console, you can write utility functions which convert commands to and from strings. Then the framework can also be used in your application.

3.1. The Setup Phase

For some applications, it may be necessary to allow scriptable user interaction during a setup phase, i.e. before or during initialization of an application. The user has the possibility to control the initialization of an application or set parameters by interactive commands. For these commands, the exact timing is typically not relevant to the flow of the application. They can simply be executed one after another.

As an example, consider a simple player application that plays files from a play list. The entries of the play list are entered by the user on the console after application startup. When the user enters a "start", playback begins. Such an application would have a setup phase during which tracks can be added to the playlist using command such as "addTrack". After the setup phase, the user (or an interaction script) tells the application to start using the "start" command. With this command, the setup phase ends. The application initializes itself by setting up an NMM flowgraph. Finally, the flowgraph is started, and timed scriptable user interaction begins.

A setup phase can be added to an application by adding another block of code that reads user commands and supports scriptable user interaction between the `prepare` and `begin` calls. Example:

```
script.prepare();

// ...

while(true) {
    string s;
    BEGIN_SCRIPTING(script, s) {
        // read a line from standard input
        char buffer[256];
        cin.getline(buffer, sizeof(buffer)-1);
        s = buffer;
    }
    END_SCRIPTING(script, s);

    // create a string stream for parsing the input
    string command;
    istringstream is(s);
    is >> command;

    // parse the command typed by the user
    if(command == "start") {
        // Exit setup ophase
        break;
    }
    else if(command == "addTrack") {
        // ...
    }
    // ...
}

// ...
```

```
script.begin();
```

3.2. Starting Processes dynamically at Runtime

In some application scenarios, processes may dynamically start and exit while other processes are running. For example, networked devices may be powered on and off, and applications may be running on these devices while the devices are powered on.

The scriptable user interaction framework does not directly support starting and stopping processes at runtime. When playing back interaction scripts, all processes must be started at the beginning, then playback will start.

However, application scenarios like the one described above can be simulated by adding interactive "power on" and "power off" commands. These commands would simulate application startup and application shutdown. For example, a server application would register a service and open a TCP listener in response to the "power on" command and unregister the service and close the TCP listener in response to the "power off" command.

Also recall that, due to the time synchronization which is performed when playing back interaction scripts with multiple participating processes, it is always possible to start one or more processes later than other processes. For example, if during recording `your_application` is started 5 minutes after `serverregistry` is started, then during playback, the starting of script playback in `your_application` will also be delayed by 5 minutes, regardless which process was started first during playback, and when each process was started.

4. Interaction Script File Format

This section describes the file format and syntax of interaction scripts. The information in this section can be used for modifying interaction scripts manually or writing complete scripts from scratch.

Interaction scripts are text files in native format (i.e. interaction scripts generated on Linux will have Linux-style line breaks). The general format is:

```
<HEADER>
endheader
<SETUP-PART>
begin
<BODY>
```

Unless stated otherwise, lines containing only whitespaces as well as lines beginning with a comment character ('#') will be ignored.

4.1. Interaction Script Header

The header of the interaction script consists of "key=value" pairs. Keys and values are arbitrary strings. Keys may not be duplicate. Values must not be enclosed in quotes, and they may contain whitespaces. General syntax:

```
<KEY>=<VALUE>
```

Examples:

```
version=2
identifier=YourTest
commandline=your_application buffalo_soldier.mp3
```

The interaction script reader ignores keys it does not know. The keys that are mandatory in every interaction script are summarized in Table 1.

Table 1. Mandatory Script Header Fields

Key	Description
version	The version number. Must be 2
identifier	The identifier that was given with ---identifier when recording
commandline	The command-line that was used when recording, not including logging options
processes	The number of processes, must be an integer >= 1

When recording an interaction script, some optional keys are also written into the interaction script. They contain additional information about the process and are summarized in Table 2.

Table 2. Optional Script Header Fields

Key	Description
fullcommandline	The command-line that was used when recording, not including logging options
hostname	The name of the host that the interaction script was recorded on
username	The name of the user that ran the process that recorded the interaction script
workingdirectory	The initial working directory of the process that recorded the interaction script

Key	Description
NMM_DEV_DIR	The value of the <code>NMM_DEV_DIR</code> environment variable in the process that recorded the interaction script
LD_LIBRARY_PATH	The value of the <code>LD_LIBRARY_PATH</code> environment variable in the process that recorded the interaction script
DYLD_LIBRARY_PATH	The value of the <code>DYLD_LIBRARY_PATH</code> environment variable in the process that recorded the interaction script
PATH	The value of the <code>PATH</code> environment variable in the process that recorded the interaction script

The interaction script header ends with a line containing only the word "endheader".

4.2. Interaction Script Setup Part

Between the header and the body of an interaction script lies the setup part, which consists of a list of untimed commands for setting up an application before the actual timed user interaction starts.

During playback, commands in the setup part of an interaction script are ignored by applications, unless they implement a setup phase as described in Section 3.1. If a script was recorded by an application without a setup phase, the setup part is empty. The setup part may contain empty lines and comments. The setup part ends with the beginning of the body of the interaction script.

As an example, an interaction script for the hypothetical playlist application described in Section 3.1 might have a setup part which looks as follows:

```
addTrack buffalo_soldier.mp3
addTrack never_gonna_give_you_up.mp3
start
```

Commands in the setup part are executed during the setup phase of an application one after another. The last command is typically a command which causes the application to leave the setup phase (such as the "start" command in the above example).

4.3. Interaction Script Body

The interaction script body begins with a line containing only the word "begin". This line is followed by the time offset (the absolute system time at which recording has started), which has the following format:

```
<SECONDS> <NANOSECONDS>
```

where SECONDS and NANOSECONDS are integers. If the script was generated by recording, then the time is the Unix time, as returned by `gettimeofday()` (on POSIX systems) or `TimedElement::getTime()` in NMM (on all platforms), when recording has started. However, the time offset is arbitrary, as long as the time offsets of the scripts of all participating processes of the same test correspond to each other (i.e. they should be within a reasonable range, and they will control at what times the processes are started relative to each other).

Example:

- Process #1 has a time offset of 1211553583 970344000.
- Process #2 has a time offset of 1211553593 970344000.

Then process #2 will always be started 10 seconds after process #1.

The exact same behavior is achieved if

- Process #1 has a time offset of 0 0 and
- Process #2 has a time offset of 10 0.

The remainder of the interaction script body consists of timed commands. A timed command has the format

```
<DELAY>
<COMMAND>
```

As an exception to the general syntax rules, COMMAND must be in the next line after DELAY. No blank lines or comment lines are allowed in between. Furthermore, COMMAND may consist of only whitespaces or begin with a comment character; it will always be interpreted as the command string.

The DELAY of a command specifies the time when a command is executed relative to the previous command. For the first command in an interaction script, the DELAY specifies the time when the command is executed relative to the time offset (the time when playback has started or looped).

More precisely: Let the "timestamp" of a command be the time when the command is expected to begin its execution. Then, the timestamp of the first command is

```
timestamp[0] = time_offset + DELAY[0]
```

and the timestamp of the Nth command is

```
timestamp[N] = timestamp[N-1] + DELAY[N]
```

Note: Note that if a command could not be executed at the time given by its timestamp, for example because the execution of the previous command took too much time, this does not affect the timestamp of this command. In other words, the DELAY value of a command is relative to the expected time at which the previous command was executed, not the actual time during playback.

This is important when executing commands that may block or take a substantial and varying amount of time to finish. If commands can not be executed in time, they will be executed as fast as possible, and the following message will be displayed on the console:

```
WARNING: Command delayed for 1 sec, 575115000 nanosec
WARNING: Tolerated delay is 0 sec, 50000000 nanosec
WARNING: Maybe commands can not be executed as fast as requested?
```

The format of the delay is:

```
<SECONDS>,<XXX>.<YYY>.<ZZZ>
```

where SECONDS is an integer, and XXX, YYY and ZZZ are integers consisting of exactly 3 decimal digits. The delay value in seconds is computed as

$$\text{delay} = \text{SECONDS} + \text{XXX} * 10^{(-3)} + \text{YYY} * 10^{(-6)} + \text{ZZZ} * 10^{(-9)}$$

i.e. XXX are milliseconds, YYY are microseconds and ZZZ are nanoseconds. For example, the delay string

```
10,111.002.033
```

means: 10 seconds plus 111 milliseconds plus 2 microseconds plus 33 nanoseconds.

The COMMAND is a free-form string consisting of one line in the interaction script, which will be passed to the application as a user command.

4.4. Random Delays

It is possible to specify ranges of delays using the following syntax:

```
<MIN_DELAY> : <MAX_DELAY>
<COMMAND>
```

where MIN_DELAY and MAX_DELAY are delay values which have the same format and meaning as DELAY above. The difference is that they define the possible range for a random delay. On playback, a random delay within the range is chosen as the delay of the command.

4.5. The Loop Marker

As described in Section 2.8, interaction scripts may be loopable or not. A loopable interaction script contains a "loop" marker in the interaction script body. The marker can be inserted between any two commands. Only one loop marker is possible per interaction script. Example:

```
# Switch to next track after 2 seconds
2,000.000.000
n

# Loop
loop

# Exit application after 2 seconds
2,000.000.000
q
```

This script executes the "n" command every two seconds in a loop. After the maximum number of loop iterations has been reached, the "q" command is executed 2 seconds after the last "n" command.

Note that the "loop" marker has no delay value itself. If the script loops, then the first command of the script is timed relative to the last command before the "loop" marker, using the delay of the first command of the script as delay value. The effect is the same as if the looped sequence of the script was repeated multiple times in the script file.

4.6. Random Fields

Commands can contain random fields. A random field is an expression enclosed in escape brackets. The grammar of random fields is

```
<RANDOM> := "{" <OPTIONS> "}"
<OPTIONS> := <RANGE> ["|" <OPTIONS>] *
<RANGE> := <ANY> | [<NUMBER> ":" <NUMBER>]
<NUMBER> := (a floating-point or integer number)
<ANY> := (any sequence of characters)
```

Examples:

```
# Seek to a random position between 0 and 100
# (e.g. "seek 55" will seek to position 55%)
1,000.000.000
seek {{0:100}}
```

```
# Go to the next or previous track
# (e.g. "next" will go to next track)
1,000.000.000
{{next|previous}}
```

```
# Enter fast forward or rewind mode with a time scale between 1 and 10
# (e.g. "rew 5" enters rewind mode with a time scale of 5)
1,000.000.000
{{rew|ff}} {{1:10}}
```

```
# Seek to a position between 0% and 10% or between 40% and 50%
1,000.000.000
seek {{0:10|40:50}}
```

For example, the following interaction script could be used as a stress test for a player, which understands the commands "play", "pause", "ff" and "rew". The "ff" and "rew" commands accept an additional numerical parameter which is the time scale relative to normal playback speed.

```
version=2
identifier=TestPlayer
processes=1
commandline=your_application buffalo_soldier.mp3
endheader
begin
0 0

# Enter fast forward or rewind mode
# The time scale is chosen randomly between 1 and 10
0,020.000.000 : 1,000.000.000
{{rew|ff}} {{1:10}}
```

```
# Pause or play
0,020.000.000 : 1,000.000.000
{{pause|play}}
```

```
# Pause or play (again)
0,020.000.000 : 1,000.000.000
{{pause|play}}
```

```
# Loop
loop
```

```
# Exit application
0,020.000.000
```

9

This interaction script first waits between 20 milliseconds and 1 second and then randomly enters rewind or fast forward mode with a time factor between 1 and 10. After another delay between 20 milliseconds and 1 second, playback either resumes, or pause mode is entered. Finally, after another delay, a random switch between "pause" and "play" mode may happen.

If this interaction script is played in a loop, and all actual commands are recorded to a new interaction script, e.g, by using the following command

```
your_application buffalo_soldier.mp3 ---play=the_above.isc \  
  ---log-to=the_following.isc ---identifier=PlayerTest ---loop
```

where `the_above.isc` is the name of a file containing the above interaction script, then the output (in `the_following.isc`) might be something similar to:

```
0,864.645.000  
ff 4  
0,996.060.000  
pause  
0,605.078.000  
play  
0,427.993.000  
rew 4  
0,536.010.000  
pause  
0,691.057.000  
pause  
0,408.025.000  
ff 9  
0,948.059.000  
play  
0,164.872.000  
play  
0,759.185.000  
rew 7  
0,624.798.000  
play  
0,055.998.000  
pause  
0,275.260.000  
rew 1  
0,456.031.000  
pause  
0,240.721.000  
pause  
0,203.306.000  
rew 7
```

```
0,028.677.000  
pause
```

5. Source Code

Source code of the scriptable user interaction framework can be found in the directory

```
nmm/services/test
```

and it consists of the following classes and interfaces

- `InteractionScript`
- `InteractionScriptReader`
- `InteractionScriptWriter`
- `InteractionSynchronizer`
- `IInteractionSynchronizer`

Please refer to the Doxygen documentation of these classes for further details.