LIME Interface Test Bench User Guide

Xi Chen, Janne Kauttio, Olli Saarikivi and Kari Kähkönen

October 6, 2009

Contents

1 Introduction

LIME (LightweIght formal Methods for distributed component-based Embedded systems) Interface Test Bench (LimeTB from now on) is a toolkit for testing software components written in Java programming language (limited support for C programming language is also available). With LimeTB, the user can specify behavioral aspects of the software component interfaces as annotations in the source code by using LIME Interface Specification Language (ISL). These specifications can then be automatically monitored for violations during the execution of the program. LimeTB also has tools for automating the testing of a program, and for generating JUnit unit tests.

This document introduces the tools provided by LimeTB and how they can be used with the provided graphical user interface. An example program is also introduced that will be used as a running example throughout the guide. The example program is a simple Java program with interface annotations, and will hopefully motivate the reader on how the tools can be useful also in real-life applications. All the LimeTB tools can be used from the user interface and also have a command-line variants that can be used independently.

Conventions used in this guide are as follows:

- `Monospace` font will be used for both commands in the graphical user interface as well as command-line commands.
- **Bold** font will be used for different file names throughout the guide.
- *Italic* Italic font will be used to denote names of directories.

The guide is structured as follows: In chapter 2 the example program will be presented on a source code level, along with explanations what the different parts of the program do. Chapter 3 will go through the initial steps of using the tool, mainly how it's installed and how the graphical user interface can be started. Chapters 4 and 5 will go through the main features of LimeTB by presenting the tools and how they work with the example program. Finally, Chapter 6 will briefly consider the utilities specific to the user interface that are designed to make it more usable. In every chapter where a feature of LimeTB is presented, also the equivalent command-line tools will be considered briefly, mainly to let the user know what command does what.

2 The example program

In this section the example used during the rest of the guide will be demonstrated in detail with complete source code of all the files used. The files will be presented one at a time, each of them followed by notes about what it actually does and what we are trying to achieve. The example program is a simple lock implementation that can be locked and unlocked, and an associated main-method that generates one such lock instance and does some operations on it. It should also be noted that this example is also available in the *LimeTB/examples/limtlct/example_lock* directory in the tool package.

```
package example_lock;

import fi.hut.ics.lime.aspectmonitor.annotation.CallSpecifications;

@CallSpecifications(
  regexp = { "StrictAlteration ::= (lock() ; unlock())*" }
)

public interface Lock {
    public void lock();
    public void unlock();
}
```

Figure 1: Lock.java

In Figure 1 is the interface definition of a "Lock" interface. Apart from the extremely simple interface, the only interesting part in this file is the "CallSpecifications" annotation at the top. This LIME Interface Specification Language annotation specifies a spec called "StrictAlteration", which basically requires that the calls to methods `lock()` and `unlock()` must be done alternately, so a lock that is locked shouldn't be locked again before it is opened first. Remember that this is just one example of what a user can do with Interface specifications. For more information please check the other examples provided with LimeTB or the specification language documentation [1]

[1] Available at `http://www.tcs.hut.fi/Software/lime/`

```
package example_lock;

public class LockImpl implements Lock {
    private boolean locked;

    public LockImpl() {
     locked = false;
    }

    public void lock() {
     locked = true;
    }

    public void unlock() {
     locked = false;
    }

    public boolean getLocked() {
     return locked;
    }
}
```

Figure 2: LockImpl.java

In Figure 2 is shown a class file that implements the interface in Figure 1. The main thing to note here is that there is no need to write anything special in the classes implementing the interfaces with annotations to make them work with LimeTB tools.

```
package example_lock;

public class Main {

    public void testme (int x, int y) {
        Lock lock = new LockImpl();

        lock.lock();

        System.out.println("x = " + x + ", y = " + y);

        if (x > y)
            if (x * y == 1452)
                if (x >> 1 == 5)
                    lock.lock();

        lock.unlock();
    }

    public static void main(String[] args) {
       Main m = new Main();

       if (args.length < 2) {
           System.out.println("please give two integers as arguments");
           System.exit(1);
       }

       int x = Integer.parseInt(args[0]);
       int y = Integer.parseInt(args[1]);

       m.testme(x, y);
       System.exit(0);
    }
}
```

Figure 3: Main.java

Finally in Figure 3 is the main class of the example program. The main method here takes two arguments, and then calls a `testme(int, int)` method. What this method does is that it first creates a "LockImpl" object (that implements "Lock" interface) and then locks it. Then the method checks if the arguments fulfill a specific criteria (these tests are carefully crafted to only accept certain specific values for the sake of demonstrating the capabilities of the testing tools) and if they do, the lock is locked a second time, leading to a violation of the specification. At the end the method unlocks the lock and exits.

As the reader has propably noticed by now, the example doesn't really do anything useful. It should be noted however, that it demonstrates quite well

a case where we have a program with an erroneous execution only with very certain input values. Now imagine a case where such a component is part of some bigger software, and the error could be considerably difficult to locate.

Next we'll go into the installation of LimeTB, and how the user interface can be launched. After that the tools themselves will be presented one at a time.

3 Installation and starting up

Installation of LimeTB should be relatively straightforward depending a bit on the system where it's going to be installed. The tarball **LimeTB.tar.gz** holds basically everything a user should need to start using the tools, apart from few dependencies. Most of the dependencies are provided in the LimeTB package both as precompiled binaries (for x86 linux) and as source code, but some still need to be installed separately.

Dependencies needed by and not provided by LimeTB:

Java 1.6: Can be obtained from Sun website [2].

ACC: (AspeCt oriented C) only required for using LIMT with C programming language, can be obtained from ACC website [3] (read the installation instructions carefully).

Boolector: Is a SMT-solver used by the test generation tool. It can be downloaded at Boolector website [4]. After downloading Boolector, a shared library used by LimeTB must be compiled. A makefile for this task is provided in the LimeTB/dependencies/solvers/ folder. See the LCT README for more detailed instructions to compiling the library.

Yices: Yices is alternative to Boolector and it is available from SRI website [5]. The instructions for compiling a shared library for Yices are also given in the LCT README.

Dependencies provided by LimeTB:

Scheck: In case scheck doesn't work, necessary sources and patches for recompilation are provided in the *LimeTB/dependencies* directory.

Doxygen: Only required for using LIMT with C programming language, sources available under *LimeTB/dependencies* and precomipiled binaries under *LimeTB/bin* directory.

The installation should be as simple as extracting the LimeTB package to a directory of the user's choosing and compiling either Boolector or Yices libraries is the test generation functionality is needed. Once this has been done, the tool should be basically ready to use. The contents of LimeTB package are as follows:

[2] `http://java.sun.com/`
[3] `http://research.msrg.utoronto.ca/ACC`
[4] `http://fmv.jku.at/boolector/`
[5] `http://yices.csl.sri.com/`

bin/	This directory holds all the runnable scripts and binaries LIME-tools are used with.
ChangeLog	User can see how the tools have been updated over time from this file.
dependencies/	From this directory the user can find the necessary dependencies to run LIME-tools that are shipped with LimeTB.
doc/	Documentation (including the READMEs for individual tools) is contained in this directory.
examples/	All the example programs are under this directory, sorted by the tool they are designed for.
lib/	This directory holds all the .jar-files and other library items necessary for LIME-tools.
README	The general README file for the package, read this first.
LICENSE	The license LimeTB uses is MIT, and the license text can be found in this file.
source/	From this directory the user can find all the source files for LIME-tools that are included in LimeTB.

The easiest way to use LimeTB is to put the *bin/* directory in the search path for commands in the current shell (variable `$PATH` with bash for instance). After this has been done, the tools can basically be run from any directory in the system.

In the following chapters we'll present how LimeTB utilities can be used in the provided graphical user interface. To start up this interface, run the script named `limegui` in the *bin* directory. If the directory was added to path as advised above, then just running the command `limegui` in any directory should also work.

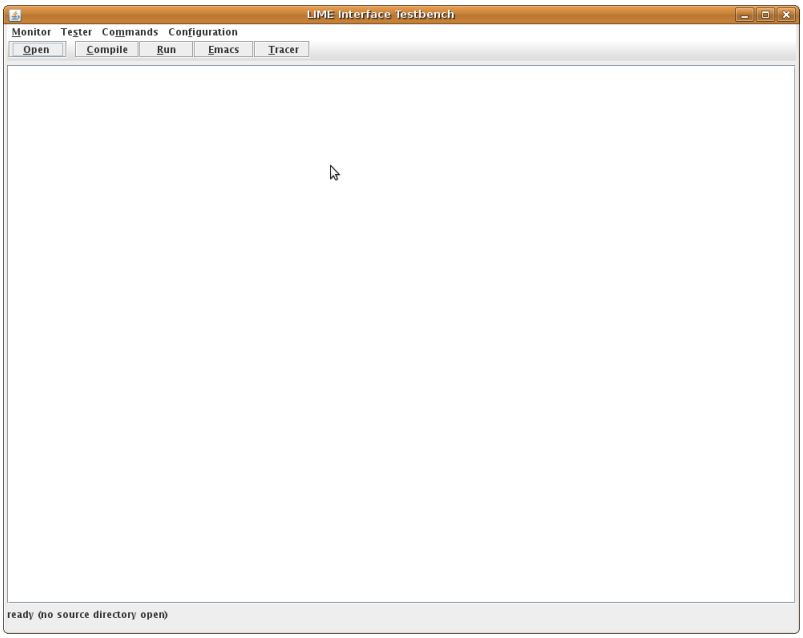

Figure 4: Freshly started user interface

Once the user interface (Figure 4) pops up, the first thing we need to do is to configure it. In order to do this, select `Configuration` from the menu on the top of the window, select `Set toolkit path...` and point the opening file dialog to the extracted *LimeTB* directory. Once this is done, select `Save config` from the `Configuration` menu and the path will be saved into a configuration file, and need not be set again after closing and restarting the user interface. The `Save config` command saves all the settings that have been set in the user interface thus far (including source directories that are open among other things), so whenever changes have been made, using this command quarantees they will be remembered.

4 Monitoring tools

In this chapter the compilation and running of an annotated program will be demonstrated. We'll also show how you can give arguments to the program and how the specification violations are reported.

4.1 Opening and compiling the example program

Once the graphical user interface has been started, the first thing that must be done is to show it where the source files are located. This can be done by selecting `Monitor`, and then choosing `Open source directory...` from the menu. Doing this will pop up a file chooser, that can be navigated to the source files. The thing to note here is that the tool needs the *directory* that holds the files as an argument, not the files themselves. In the case of our example, we

shall point it to the *LimeTB/examples/limt-lct/example_lock* directory (as can be seen in Figure 5).

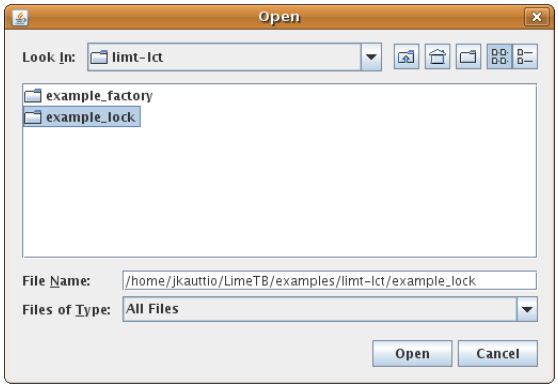

Figure 5: Select a source directory to be opened.

Once the directory has been set, the program can be compiled. To do this, the `Compile` command in the `Monitor` menu can be used, or the `Compile` button in the main interface that does the same thing. Compiling the example program (providing all the dependencies are working correctly), should produce output similar to Figure 6.

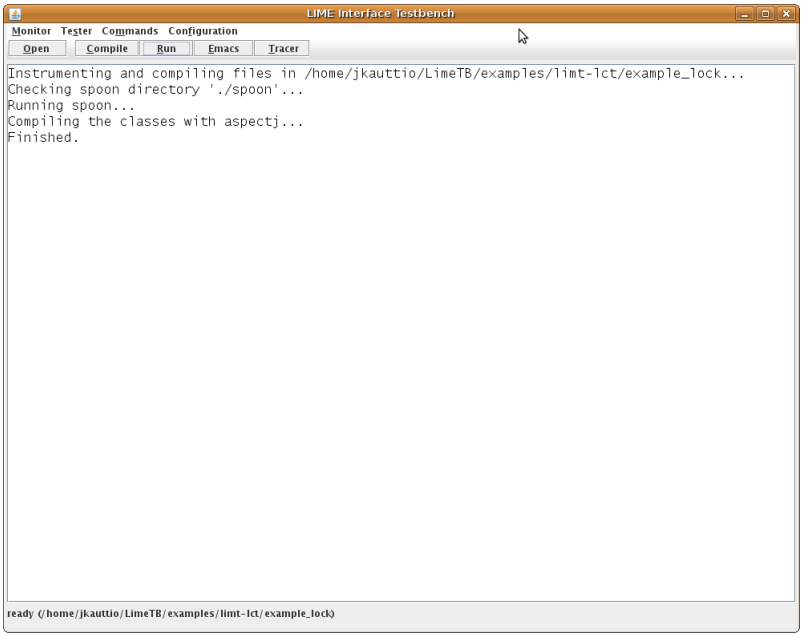

Figure 6: Output of `Compile`.

4.2 Running the example program

Now that the example has been compiled (with the monitors created automatically during the process), it can be executed. To do this, either the command `Run` under `Monitor` menu or the similarly labeled button on the main interface can be used. Notice now that the tool automatically assumes the name of the main class (class containing the runnable `main`-method) to be "package.name.Main", which it is for this example. If the running is successful, the mainwindow should look like Figure 7.

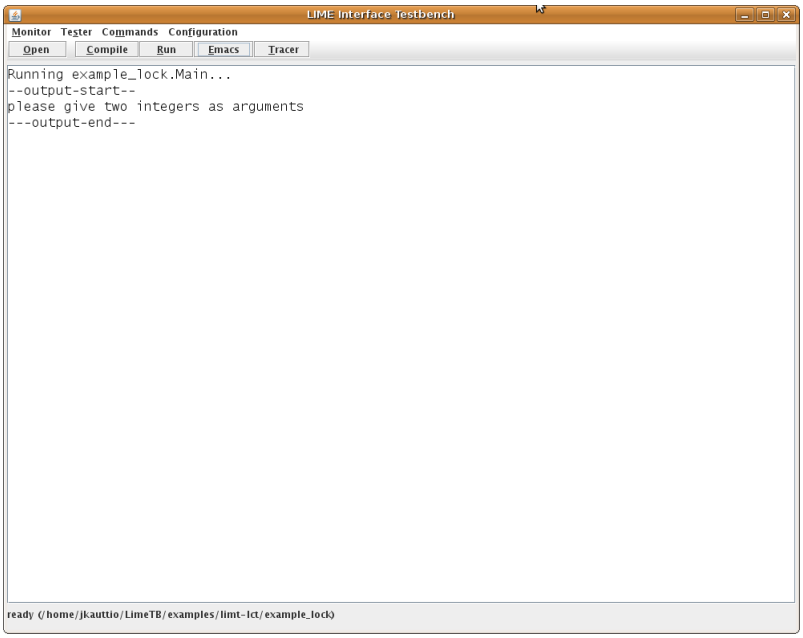

Figure 7: Output of a failed `Run` of the program.

The program didn't run properly, since as seen in Chapter 2, the program needs two parameters. To set these parameters, open the `Monitor` menu again, and choose the command labeled `Set main class...`. As can be guessed from the name of this command, this is mainly used for setting the name of the runnable class (if it isn't Main), but it can also be used for giving arguments for the program. Now insert any two integer values after the class name (13 and 42 are used in Figure 8) and try running the program again.

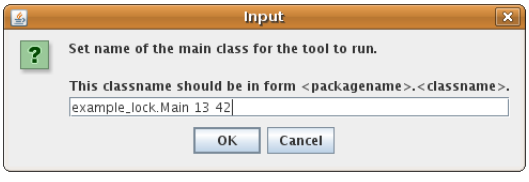

Figure 8: Setting some arguments for the example program.

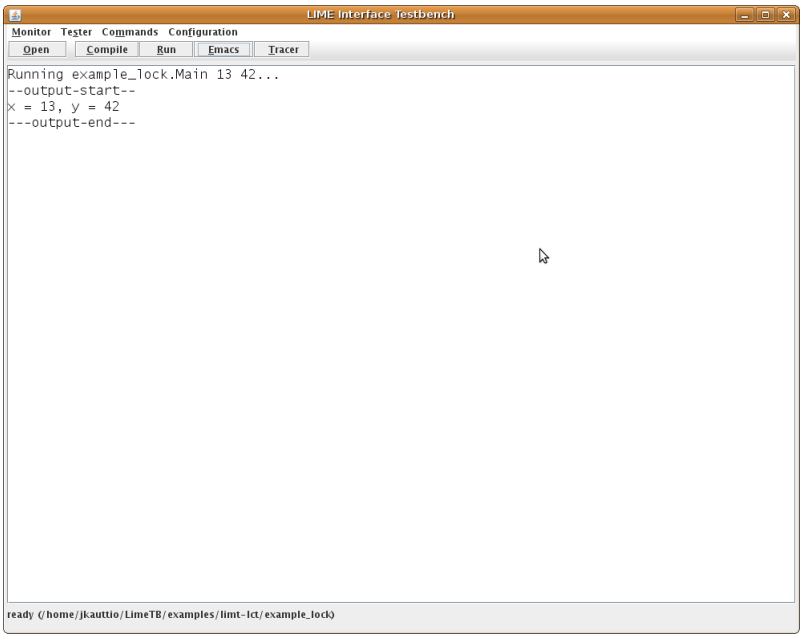

Figure 9: Output of a successful `Run` of the program.

Now the program should actually finish, and the output it produces can be seen on the interface window (Figure 9). Now any specification violations didn't happen (since 13 and 42 don't fulfill the criteria needed for the second call to lock() in the code), but we can try that again. Through some math or by previous knowledge, it can be deduced that the only values for arguments that will lead to a specification violation are 10 and -858993314 (in the testing section of the guide it is shown how these values can be obtained using the LimeTB tools). Now let's try setting the arguments to these values (Figure 10) and see what happens.

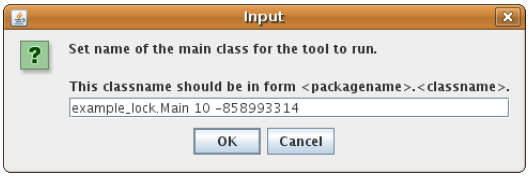

Figure 10: Setting different arguments for the example program.

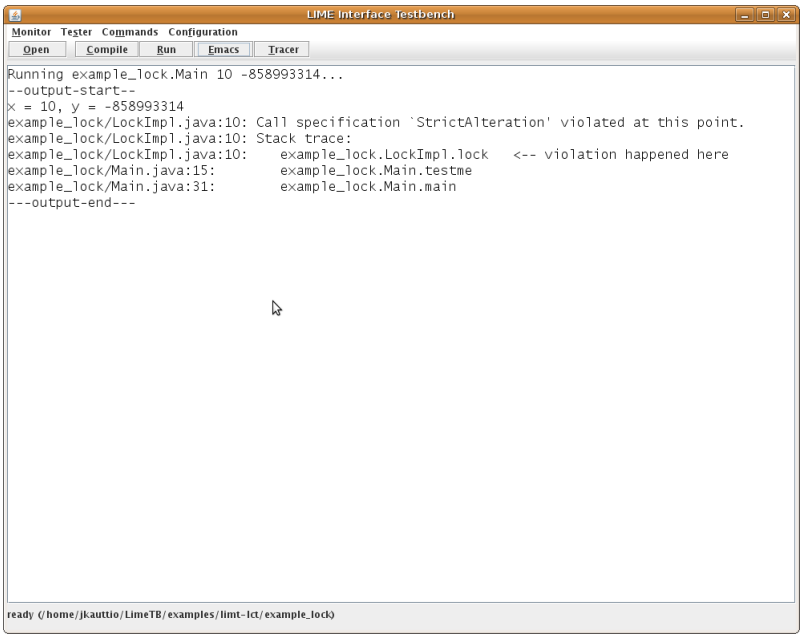

Figure 11: Output of `Run` with an exception trace.

As can be seen from Figure 11, a specification violation happened. Specification violations cause a special exception to be thrown in the code, and our tools clean up the normal java exception output to a more readable form automatically before showing it. From the output we can now see that the class "LockImpl.java" caused the exception (since the `lock()` method is implemented there), and it was called from "Main.java" on line 15 (which is where the second call to `lock()` is). From the output we can also see the type of specification that was violated was a "Call specification", and that the name of the spec was "StrictAlteration". By looking the source code of the example in Chapter 2 we can see that this indeed is the case.

4.3 Other monitoring tools

Now for the other previously unmentioned features of the monitoring toolkit. The user interface has buttons for `Emacs` and `Tracer`. The `Emacs` command opens the text editor "emacs" on the current output on the screen, which is helpful for these exception violations since the user can navigate straight to the violating source files by just clicking the name of the file with the line number in the editor. This kind of integration isn't provided for any other text editor at the moment, but the user can of course edit the source files in any way he likes.

The `Tracer` feature is a simple text-based tracing utility, that can be used to view the run of the program from the eyes of the specification monitors. The tracing utility can be launched by clicking the button, and it will automatically open the log file for the most recent run of the program as can be seen in Figure 12.

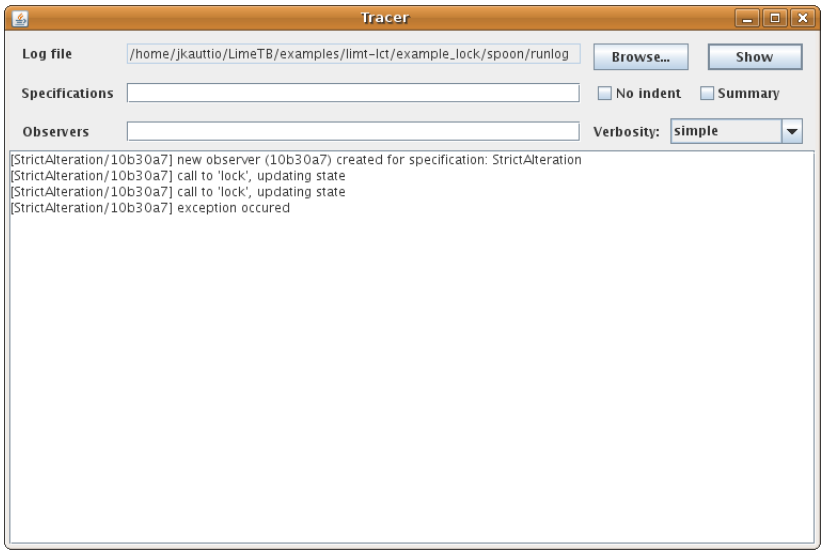

Figure 12: Tracer with "simple" verbosity.

The tracer has several filtering options, including only showing the output of monitors for certain specifications (the "Specifications:" field), or for specific observer instances (the "Observers:" field). To find out all the names of the specifications monitored and the observers created during the run, the user can check the "Summary" option and a list of them will be printed, along with information about which observer noticed the specification violation. The filtering fields should be filled with the names of the specifications (or observers) to be followed separated by commas (","). The "Verbosity" setting has three different values, "simple", "propositions" and "stacktraces", which respectively print the basic output, information how specific propositions were evaluated each time an observer's state was updated, and the stacktraces for each function call an observer noticed. Different observers have also different layers of indentation in the output to make it more readable, which can be disabled with the "No indent" option. In Figure 13 is the output for the previous run with the verbosity set to maximum level to show the effect.

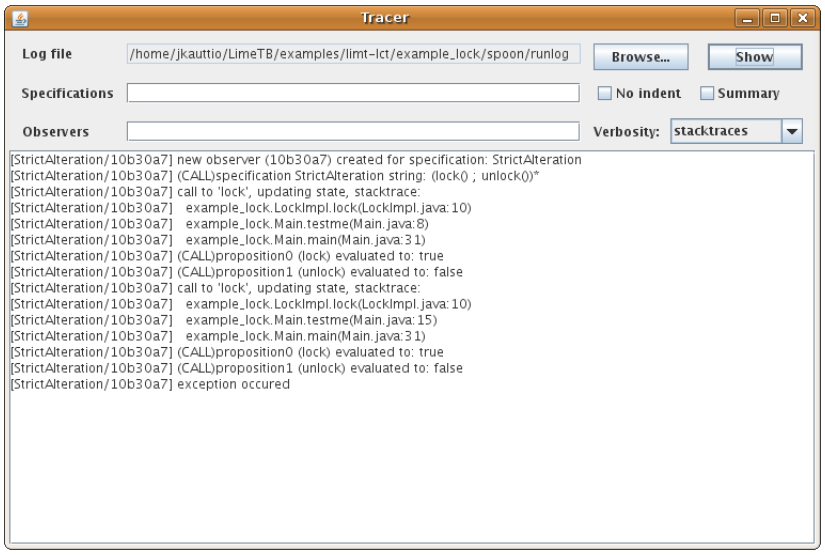

Figure 13: Tracer with "stacktraces" verbosity.

4.4 Command-line usage

All the tools mentioned above can be also used from the command line.

`limejc` is a script that takes care of instrumentation and compilation of the target program, (same as `Compile command`. The usage of this script is pretty straightforward; just run `limejc path/to/source/dir` where `path/to/source/dir` points to the directory with the source files to be compiled. If the user wants to compile the files in the current directory, `limejc .` can be used. The script creates a directory called `spoon`, which will hold all the produced class files. This directory will be removed automatically by the script if it exists already, so some care should be taken when the script is used.

In order to run the compiled program, a script called `limejava` is provided. This script should be run from the `spoon` directory created in the previous step, and must be run with the complete package and class name of the file (e.g. for the `example_lock` used in this guide, first go to the directory with the source files and run `limejc .`, then change to the created *spoon* directory and finally run `limejava example_lock.Main 13 42`).

In order to achieve the parsed output seen in the previous Section, a script called `beautify.py` can be utilized. This script takes the output of a program as input, and if it notices a LIME exception, it "beautifies" that part. The usage varies with each shell a bit (since the exceptions are printed in the error stream and that must be directed to the beautifier), but for bash a command like `limejava example_lock.Main 10 -858993314 2>\&1 | beautify.py` will do the trick.

Finally, the tracer can be used with command `tracer` (use `tracer -h` for information about additional options). The log file of the last run the tracer needs will be generated under the *spoon* directory by the name **runlog**, notice that this file will be replaced with each additional run so if the user wants to save a log for a certain run, the file should be copied somewhere safe before

running the program again.

In the next chapter we shall move on to the testing tools, and demonstrate how the program can be tested automatically (and how the input values that caused the exception in the example can be found).

5 Testing tools

This Chapter focuses on the usage of the testing utilities provided by LimeTB.

5.1 Using Lime Concolic Tester

In order to use the Lime Concolic Tester (LCT) tool, the source file for the Main class needs to go through some small changes. Basically, the tester works by generating "input" values for methods and trying to lead the execution to an erroneous state by altering those values. The LCT doesn't actually know what values it should generate unless we specifically request them, so the class file in Figure 14 shall serve as the "Main" class for test generation purposes (the file is available in the example directory under the name **MainLCT.java**).

```
package example_lock;

import fi.hut.ics.lime.tester.LCT;

public class MainLCT {

    public void testme (int x, int y) {
        Lock lock = new LockImpl();

        lock.lock();

        System.out.println("x = " + x + ", y = " + y);

        if (x > y)
            if (x * y == 1452)
                if (x >> 1 == 5)
                    lock.lock();

        lock.unlock();
    }

    public static void main(String[] args) {
       Main m = new Main();

       int x = LCT.getInteger();
       int y = LCT.getInteger();

       m.testme(x, y);
       System.exit(0);
    }
}
```

Figure 14: MainLCT.java

The input values for `testme()` are no longer read from arguments, but instead they are set to be generated by LCT. LCT supports generating primitive type input values with the exception of floating point numbers and also some support for generating input objects is provided. To read an input value from LCT, method calls LCT.getInteger(), LCT.getShort(), LCT.getLong(), LCT.getByte(), LCT.getBoolean(), LCT.getChar() and LCT.getObject() can be used. For more information about these methods, see the LCT README file in the documents folder of LimeTB. It should also be noted that this file is also completely runnable. If ran by itself the values LCT generates will just be random, which is basically a single test run for the program.

The usage of testing tools in the user interface follows the same track as the monitoring tools. First, we need to select the directory where the tests should be executed (this step can be omitted if a source directory has already been selected

for the testing tool, the path will be filled out automatically). This can be done by opening the `Tester` menu, and choosing the command `Set test path...` which will open a new file chooser. When selecting this directory manually, the user should remember to set it to the parent directory of the source files if they belong into a package (the directory that contains the package directory).

Once the path has been set, the program can be instrumented and compiled by using the `Compile and/or instrument...` command in the `Tester` menu. Now unlike the monitoring tools, this compilation process takes a single class or java **file** as an argument. So in the case of this example, point the file chooser first to *example_lock* directory (under the *spoon* directory where the file chooser opens), and then to either **MainLCT.java** or **MainLCT.class** (Figure 15). If the compilation process in the monitoring tools hasn't been run before and these files don't exist, for the sake of this example the user should go back to Chapter 4 and do the necessary steps mentioned there.

It should also be mentioned that even though we use LCT here with the monitoring tools, also programs that don't have specification language annotations in them and haven't been compiled by the monitoring tools can be tested with LCT. However, as the error in our example depends on the interface specifications, we must use the monitoring tools in this case.

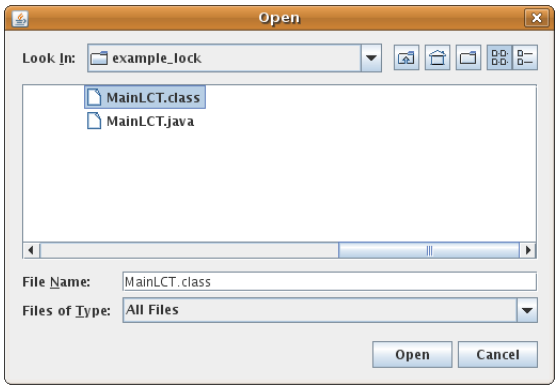

Figure 15: Choose a file to be compiled and/or instrumented.

A new window will be opened where the output of the compilation and instrumentation process (providing it was successful) should look like in Figure 16.

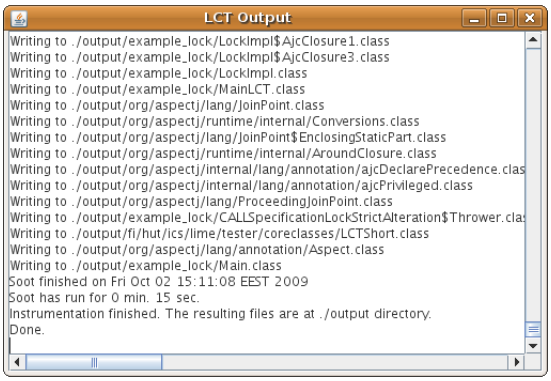

Figure 16: Output of `Compile and/or Instrument...`

The next step now is to run the tester for the instrumented program. In order to do this, choose `Run...` command from the `Tester` menu. The correct class file should automatically be selected in the opening file chooser (Figure 17), so accepting that will usually be enough to run the tester. If something goes wrong however and the file isn't selected automatically, the user must navigate to it manually. LCT compilation and instrumentation process creates a directory called *output* under the directory which has been set as the test path with `Set test path...` command. Under this directory the class files generated by the process should be found (in their respective package directories), and can then be selected for the tester to run tests on.

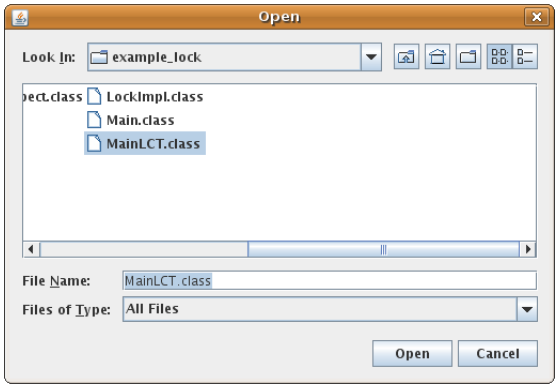

Figure 17: Choose an instrumented class file to run.

As the testing commences, a new window is opened where all the single test case outputs will be printed as can be seen in Figure 18. In the main window a test server will be run, which will print out important information about the performed tests. Output for the example should look like Figure 19. Here can be seen (among other things) the following: One error was found using input values 10 and -858993314 (which are the only ones leading to an error and LCT found them automatically), total number of execution paths explored was 4,

and the "test coverage" was 44%. The low value for test coverage is caused by the fact that we now have two different Main classes in the package (Main and MainLCT), and the other one wasn't explored at all during the run.

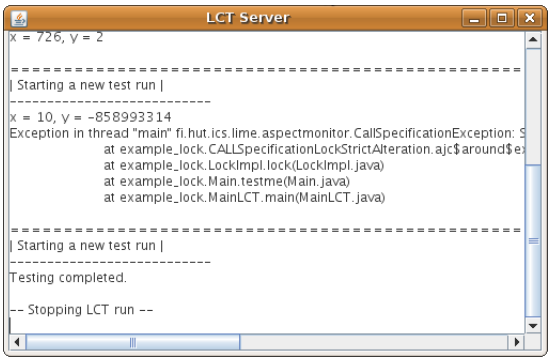

Figure 18: Output of `Run` command; information about individual tests.

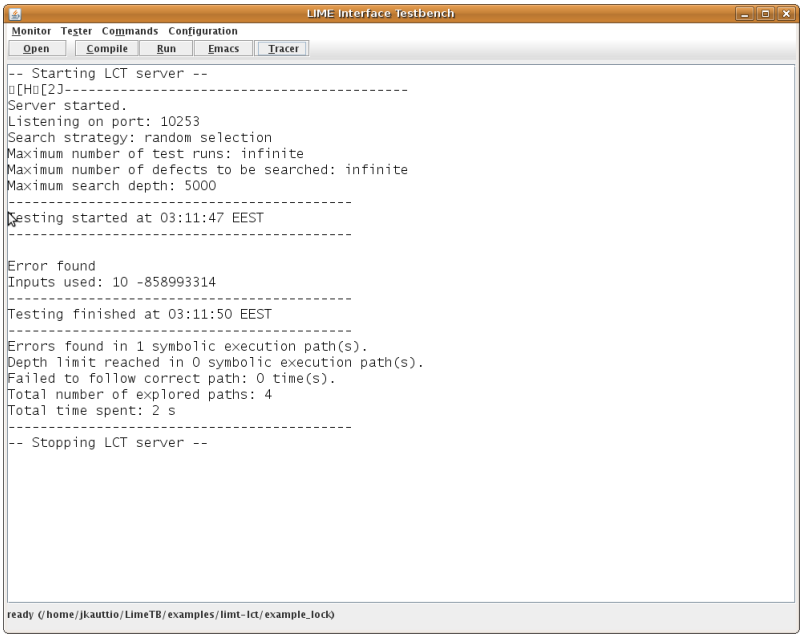

Figure 19: Output of `Run` command; found errors and general summary.

If something goes wrong when running the LCT tester and the server is left running, it can be killed by closing the output window so the run can be started again (with default values only one server can be running at a time on one machine).

The `Tester` menu also has a `Tester configuration` submenu, where many of the settings used by the tester can be changed. For normal usage the default

values should be fine but should the user want to change them, more documentation on what they do can be found in the README file of LCT.

5.2 Limitations of LIME Concolic Tester

LCT has some limitations that affect what kind of programs it can test efficiently. One of the current major limitations with LCT is that it does not instrument any of the standard Java libraries by default. This means that it is not possible to use, for example, String objects as inputs created by LCT. There is, however, support for Integer, Boolean, Short and Byte classes provided. The support for these classes is implemented by providing a custom implementation of the classes and replacing the original Java standard classes in the program under test with these new classes. This has the downsize, that if the program under test uses a piece of code that has not been instrumented by LCT and it receives one of the supported standard class objects from the instrumented code, the program may crash as the types of the original and replaced classes are different. Therefore it is recommended to use the above mentioned classes in the program under test only if the program can be fully instrumented. For a more detailed information about this limitation, see the research report on the test generation method available at the LIME website [6].

Another limitation of the tool regards handling arrays. Currently the tool provides approximative support for generating tests for programs that use arrays. More detailed explanation of how arrays are handled is presented in the above mentioned research report in section 3.3. As an example of this limitation, consider the following piece of code.

```
// variable a is an array of Strings

int x = LCT.getInteger(0,10);
System.out.println(a[x]);
```

The program prints out the element in array a based on the input values generated by LCT. However, for this program LCT generates only one test case where the value of x is selected randomly between 0 and 10, instead of generating test cases that print all the values in the array. This is because LCT tries to generate possible execution paths to explore based on the branching statements (e.g., if-statements) written in the program under test. In this particular example there are no such branching statements and therefore LCT thinks that there is only one possible execution path in the program.

5.3 Using JUnit Tools

The LIME JUnit Tools (LJUT) provides tools for generating test drivers for unit testing and interface testing. The generated test drivers retrieve input from the LCT to drive testing. LJUT will also save the tests that were run with the LCT as a static JUnit test case to be included in a unit test suite.

The LJUT tools can be accessed from `Test drivers and JUnit tests...` menu item in the `Tester` menu. A new window for accessing the LJUT tools opens (Figure 20) . This window contains a row of buttons for accessing the

[6] `http://www.tcs.hut.fi/Software/lime/`

tools and two file path fields for providing the JUnit test generator tool with the input files it needs.

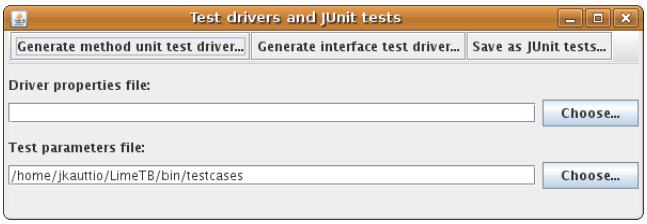

Figure 20: LJUT window, some of the fields empty.

To be able to generate test drivers the toolset must be pointed to the correct project folder. LJUT uses the same folder as the LCT, which is set from the `Set test path...` menu item in the `Tester` menu. Notice that if you have been following this guide from the beginning, the path doesn't need to be set again at this point.

The `Generate method unit test driver...` button in the `Test drivers and JUnit tests` window provides access to generation of straightforward unit test drivers for single methods. Pressing the button will open a dialog box asking for a constructor and method signature. These are written in a Java like syntax. Examples of valid constructor signatures are "java.lang.String()" and "my.package.hierarchy.MyClass(int,long)". Valid method signatures include "foo(boolean,byte)" and "bar(java.lang.Object)". For our example, fill in the values "example_lock.Main()" as a constructor signature, and "testme(int,int)" as the method signature in order to generate testcases for the testme method. After pressing the `OK` button in the dialog, an output window is opened (see Figure 21) and the test driver is generated in the project folder.

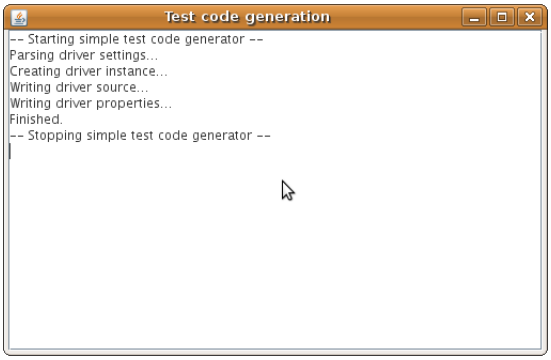

Figure 21: LJUT simple test driver generation output.

The `Generate interface test driver...` tool is similar to the unit test driver generator. It also takes a constructor signature, but instead of a method signature it needs a qualified name of an interface. The class the constructor signature refers to must implement that interface. The test driver generated

will perform testing on sequences of calls to the provided interface's functions. The dialog also asks for the number of iterations in the test driver that will determine the length of the call sequences to be tested.

After a test driver is generated it can be used with the LCT. The test drivers are generated as files named ClassNameLCTDriver.java in the project folder (notice that for our example this isn't the folder where the other source files are since they belong to the package "example_lock", but one level higher in the file hierarchy), where ClassName is the name of the class under test. A second file ClassNameLCTDriver.properties is generated and is needed for JUnit test case generation (if this functionality is not needed the file can be discarded). As described in the section on using the LCT the test driver must be compiled and instrumented before use. When the tests are run with the LCT the server writes a testcases file in the *bin* folder in the LIME distribution folder. This file is used in the generation of JUnit tests.

To generate JUnit tests the properties file and testcases file fields in the `Test drivers and JUnit tests` window must point to valid files (Figure 22). These are normally prefilled during the generation of test drivers, but may not be correct in all situations. To generate the JUnit tests press the `Save as JUnit tests...` button. An output window is opened again (the output should look like Figure 23) and JUnit tests are written in to a file named ClassNameLCTDriverTestCase.java in the project folder.

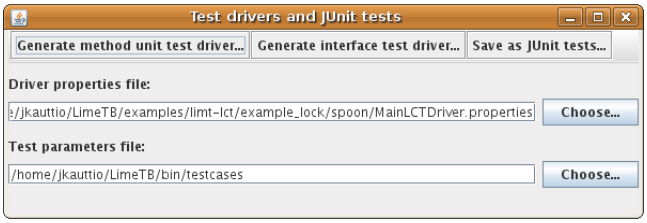

Figure 22: LJUT window, all fields filled.

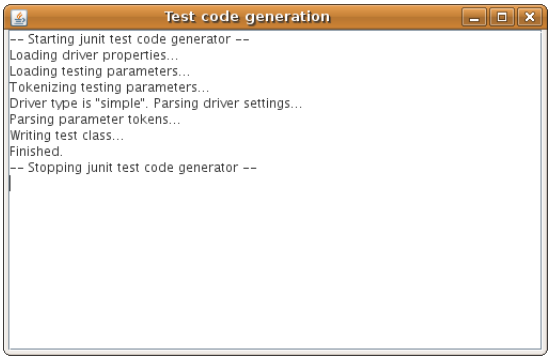

Figure 23: LJUT JUnit test generation output.

5.4 Command-line usage

Before running the tools from the command line, make sure *bin* folder is in `$PATH` as described in Chapter 3.

LCT can be used from the command line with commands `LCTcompile`, `LCTinstrument`, `LCTserver` and `LCTrun`. Basically the testing process works like in the user interface: Change to the proper directory, compile the files if needed (if working with .java files you'll need to do this, recompilation is not needed if LCT is used together with the monitoring tool), instrument the files, start the server (preferably in a different terminal) and run the tester.

`LCTcompile` and `LCTinstrument` commands take .java and .class files as arguments respectively, and `LCTinstrument` saves the output files it produces in and *output* directory which will be generated automatically.

In order to run `LCTrun`, the test server must be running. The server can be started with `LCTserver` command, and should be started in a new terminal window as mentioned above. `LCTrun` takes .class files as arguments, and these files should be instrumented so please use the files from the *output* directory.

Using LJUT tools from the command line is essentially identical to using them from the GUI. All of the command line tools provide help on the possible options when given a `--help` option.

The unit test drivers for single methods are created with `ljut-simple`. This tool needs a constructor and method signature. For example it might be called like this:

```
ljut-simple "example_lock.Main()" "testme(int,int)"
```

The interface test drivers are created with `ljut-interface`, which needs an interface name, a constructor signature and the number of iterations to do. For example:

```
ljut-interface "example.File" "example.FileImpl" 2
```

JUnit test generation is done with the `ljut-junit` tool that expects a properties and a parameters file (the "testcases" file). It could be called for example like this:

```
ljut-junit --parameters-file /home/user/LimeTB/bin/testcases
TestClassLCTDriver.properties
```

6 Other utilities

The last set of utilities a use can find in the interface is located under the `Commands` menu. Usually this menu shouldn't need to be used at all, but for instance in the rare cases where the build process for the program under test can't be handled by the Lime tools, the commands under this menu might prove useful.

First there are the `Add custom run command...` and `Remove custom run command...` items. With these features the user can add any command that's

runnable from command line to a list of "runnable commands", and run it from the user interface. In order to add a command, three things need to be specified: A name for the command which is used to just tell te commands apart from each other, the directory where the command should be run, and last the command itself. Some care should be taken when making and running custom commands, since a badly placed `rm -fr` for instance can do a serious amount of damage.

`Change PATH variable...` and `Change CLASSPATH variable...` commands can be used to add something extra to the respective environmental variables if that's needed. In most cases the Lime scripts take care of having all the necessary variables set correctly, but if something ends up missing it can be added here. Another thing to note is that these "extra" variables take effect before the actual variables set by the scripts. So if the system has one version of Java installed and the user wants to use a different one that's installed somewhere else, the path to the preferred Java executable directory can be set with `Change PATH variable...` and it will be used instead of the default one.