CS 544 Compilers - Pollice
DSL project documentation
Nathan Longnecker

Mut: A Mutation Testing DSL

Mut is a mutation testing domain specific language. It allows users to quickly specify Java source and JUnit test files and run mutation tests on them using very few commands.
The purpose of this project is to make it as easy as possible to mutation test Java projects. It was successful, but the resulting program is not as stable, efficient, well tested, and organized as I would like it to be.

Contents

Submission contents

This zip contains:

MutationTestingDSL
: The Eclipse project for Mut.

MutatorTestProject
: A sample project that can be tested using Mut. Contains some simple functions to be mutated.

Mut.jar
: The compiled jar of Mut. Can be run from command line using the Java JDK.

README.pdf
: This file.

binaryLogic.mut, mathFun.mut, testScript.mut
: A set of .mut files that can be used to test the program.

Running

The program must be run with the JDK. The JRE will not work, because the JRE does not contain a java compiler. If you attempt to run it with the JRE, the program will throw an error when you try to compile something (the first time the mutate command is used) and recommend you try using the JDK.

REPL mode

To run the program in repl mode, run the jar with no arguments (Or just a verbosity argument). This will start the repl. To exit the repl, type 'q', 'quit', or 'exit'.
Example commands:
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar -verbosity 3
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar -verbosity normal

Script mode

To run the program in script mode, provide the names of the scripts as arguments when running the jar. The program will run the scripts sequentially and then exit. A verbosity setting may be added anywhere in the list of scripts.
Example commands:
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar testScript.mut
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar -verbosity 5 testScript.mut
"C:\Program Files\Java\jdk1.8.0_40\bin\java" -jar mut.jar testScript.mut -verbosity veryverbose scriptTwo.mut

Verbosity

In either repl or script mode, the program may be run with modified verbosity settings. By default it is set to default, or 2. I do not recommend modifying the verbosity above normal, or you will get WAY more output than you want.
When specifying verbosity settings, you may use either the names or an integer. Example usage is above. The -verbosity argument may be anywhere in the list of command line arguments.
Verbosity settings: sparse (1), default (2), normal (3), verbose (4), veryverbose (5)

In Eclipse

The program may also be run in Eclipse by running the Main method. If you do it this way, it will be in REPL mode (unless you added arguments in the run config). You can use the Eclipse console to execute commands.

Commands

source: *filelist*

Sets the source files to be the ones specified in the *filelist*. Removes all of the previous source files and adds the new ones to the in memory file system. When this command is run, the files are read into memory, so modifications to the files after running this command will not be reflected in mutations during this session.
Example usage:
source: MutatorTestProject/src
source: Mutator*/src/*.java
source: MutatorTestProject/src/nwl/MathParser.java, Mutator*/src/*Functions.java

test: *filelist*

Sets the test files to be the ones specified in the *filelist*. Removes all of the previous test files and adds the new ones to the in memory file system. When this command is run, the files are read into

memory, so modifications to the files after running this command will not be reflected in mutations during this session.
Example usage:
test: MutatorTestProject/test
test: Mutator*/test/*.java
test: MutatorTestProject/test/nwl/MathParserTest.java, Mutator*/test/*FunctionsTest.java

add (**source** | **test**) *filelist*

Adds the files in the *filelist* to the list of source or test files, depending on which was specified. When this command is run, the files are read into memory, so modifications to the files after running this command will not be reflected in mutations during this session.
Example usage:
add source MutatorTestProject/src/nwl/MathParser.java
add test MutatorTestProject/test

remove (**source** | **test**) *filelist*

Removes the files in the *filelist* from the list of source or test files, depending on which was specified, and removes them from the in memory file system.
Example usage:
remove source MutatorTestProject/src/nwl/MathParser.java
remove test MutatorTestProject/test

list (**source** | **test**)

Lists the current set of source or test files. Has no effect on the state of the system.
Example usage:
list source
list test

use *filelist*

Imports the files in the *filelist* as mut files. Immediately interprets them using the current state of Mut. If the files contain strains, those strains will be defined or redefined if they are already defined. If the file is a script, the script will execute. The script may change the source and test file lists. If the script contains mutations, those mutations will be executed. If a file does not conform to the mut grammar, antlr parsing errors will be displayed and Mut will not attempt to parse additional files in this *filelist*.
Example usage:
use binaryLogic.mut, mathFun.mut
use *.mut // Note that this imports all .mut files in this directory **and** in all subdirectories

strain *id mutate+* **end**

Defines a strain (module) for use later. The strain can later be referred to using the *id*. If a strain already exists with this id, the previous definition is overwritten. A strain can contain one or more mutate statements. Does not execute any of the mutations in the strain at this time, just registers the strain id with the symbol table for use later.

In REPL mode, newline is the command delimiter, so the entire strain must be typed on one line or Mut will be unable to parse it. It is recommended to define strains in files and **use** those files to avoid defining strains during a REPL session. See some of the example .mut files for more example strains.
Example usage:

strain logic mutate &&,|| to &&,|| end
strain comparators mutate <,>,<=,>= to <,>,<=,>= end
strain math mutate +,-,/,* to +,-,/,* end

mutate (*idlist* | *symbollist* **to** *symbollist*)

Executes a set of mutations. If an *idlist* is provided, this command executes the *mutate* commands in each of the strains specified by the *idlist*. Otherwise, this command spawns a new thread to process mutating each of the symbols in the first *symbollist* to each of the symbols in the second *symbollist*. See the next section for more details on what the process of mutating involves.
Example usage:
mutate math, logic, comparators
mutate <,>,<=,>= to <,>,<=,>=

report (**last** | **all**) (**survived** | **killed** | **stillborn** | *filelist*)?

Reports information about either all the mutations run during this session or just the last mutate command run. If no additional arguments are provided, reports general statistics including number of mutants that survived, were killed, and were stillborn, and reports a mutation score. If the user specifies survived, killed, or stillborn, then it reports about that subset of mutants, giving the percentage of mutants in that category and listing the filename, line number, and what was mutated for each mutant. If a *filelist* is provided, then it provides general statistics about the files in the *filelist*, and lists the filename, line number, and what was mutated for each of the mutants that survived.
Example usage:
report last
report all
report last survived
report all stillborn
report last killed
report last MutatorTestProject/src/*Functions.java

A *filelist* is a comma separated list of files. Each file in the list can be a relative or absolute path to a file or directory. If it is a directory, it is expanded to include every file in the directory. It recursively searches subdirectories to obtain a complete list of files in that directory.
Files in the list can also be regular expressions, using * to represent any character or set of characters. This will also recursively search subdirectories, but will not match directory names, so a regex either needs to end with * to match all files in the directory or *.java to match all java files in the directory. The regular expressions must all be relative paths.
disclaimers below
The regular expression pattern matching creates a list of all the files in the current directory and all subdirectories and then attempts to match the entered regular expression against each. This means that if this program is in a directory with many subdirectories it could run very slowly.
The file matching regular expression takes the input file string, replaces all instances of * with .* and then treats it as a java regular expression to match against all files in the directory. This could potentially cause the regex to match unexpected things (. is treated as a wildcard). The user could theoretically also use more complex java regular expression constructs, but this hasn't been tested so behavior in this case is undefined. Future versions of this program should test and either support or disallow these behaviors.

An *idlist* is a comma separated list of ids. Each id must begin with a letter and can contain letters, digits, and underscores. Ids are only used to denote the names of strains in this language. In the future, the language could be modified so that ids and symbols use the same anltr rule and the difference between them is determined contextually at runtime. This would allow for support of symbols that do not contain any special characters, which the language currently does not support because antlr identifies them as ids instead of symbols.

A *symbollist* is a comma separated list of symbols. Each symbol can be composed of almost any set of characters except for whitespace, number signs, commas, and colons. In the future this language could be expanded to allow support for these characters by implementing escape characters.

Mutating

This section goes into more detail about what actually happens in the program when it processes a mutation command. For each mutate command entered, a new thread is spawned. Each thread has its own in memory file system (when a new thread is created, the master in memory file system is cloned). This allows the threads to work simultaneously without interfering with each other.
A mutation runner thread (In class mut.mutator.MutationRunner) first attempts to compile the original source and run tests against it. If either one of these fails (The source does not compile or one of the tests fails), the thread reports an error and returns. If everything works, the thread can then begin mutating.
The thread then goes through each of the symbols being mutated from and searches for those symbols in the specified source files. For each one that is found, it mutates it to each of symbols being mutated to and then attempts to compile the code. If the code compiles, it then runs tests on it. Depending on the outcome of the code compiling and the tests running, the thread logs the mutant as either survived, killed, or stillborn. When it has finished compiling the code and running tests on it, that mutant is thrown out and the thread retrieves the original source from the in memory file system and is ready to do the next mutation. Once finished searching all the files for all the symbols being mutated from, the thread reports that it has finished and exits.

In Memory File System

Mut uses an in memory file system to reduce the number of disk I/Os and reduce running time. It stores the file system internally as a HashMap of filenames to file contents, and uses a custom JavaFileManager to allow the JavaCompiler to interface with the in memory file system as if it were a standard file system on the disk. It also uses a custom ClassLoader to allow the JUnitRunner to load and run the classes from the in memory file system.

Known Bugs

This program has a few known bugs. The two main ones are a NullPointerException in the JavaCompiler and the program hanging during a long computation.
Sometimes when attempting to compile files the compiler will throw a NullPointerException in the IsClientWrapperTrusted method (or something similar). I'm not sure why this occurs and I'm not sure how to trigger it consistently, so I wasn't able to look into it too much. I noticed it tended to happen when I was playing around with adding and removing source files in between mutations, so I suspect it may be caused when the files in the in memory file system get out of sync with what the JavaCompiler thinks is there.

When running long computations, the program sometimes hangs. This happens fairly consistently when attempting to run the program on a compiler. I'm not sure if it's due to the NullPointerException discussed earlier, whether it's garbage collection taking a long time (this usually happens when Mut is using between 1.5GB and 2.5GB of memory), or whether it's another issue.
When testing on a compiler, I used Steve Malis' compiler, as mine was getting NoClassDefFound errors with DijkstraRuntime when attempting to run tests; I suspect that was due to the fact that the ClassLoader that instantiated the test program didn't know about DijkstraRuntime, but I'm not entirely sure why it was different when running tests normally vs through Mut. Steve's compiler doesn't use DijkstraRuntime, instead it calls Java's System.out.println() method directly.