Project 2: Comparing Coverage

In this project, you write tools that collect and compare the coverage of multiple program runs in order to detect anomalies. Your task is to implement those tools and to apply them to two programs:

- The middle[1] program and its test runs as described in the book. Contrast the failing run with the passing runs. Where does coverage information point to as the reason for the failure?
- The XMLProc parser from Project 1[2]. The XMLData archive[3] contains a number of passing and failing test inputs. For each of the three failing test inputs, compare their coverage with the coverage obtained from the passing test inputs. Where does coverage information point to as the reasons for the failures?

Obtaining Coverage (30% of the points)

Create a tool that obtains coverage of a given program run on a given input. The main file of your tool should be called `get_coverage.py` and it should be callable as follows:

```
python get_coverage.py OUTPUT_FILE PROGRAM [ARGS]
```

`OUTPUT_FILE` is the name of the file to write coverage information to. `PROGRAM` is the program to be executed and `ARGS` are optional arguments to be passed to the program. Two sample calls to obtain coverage of the `middle.py` program[4] and the XMLProc parser, respectively, look as follows:

```
python get_coverage.py middle.cov middle.py 2 1 3
python get_coverage.py xmlproc.cov xmlproc/xpcmd.py input.xml
```

The output file should be a text file containing information about all the lines in the analyzed program that were executed, with each line in the following format:

```
RELATIVE_FILE_NAME:LINE_NUMBER
```

where `RELATIVE_FILE_NAME` is the path to the file relative to the directory, from which your tool was called. Sample output snippet for the first of the above calls looks as follows:

[1] http://www.st.cs.uni-saarland.de/whyprogramsfail/code/middle/middle.py
[2] http://www.st.cs.uni-saarland.de/whyprogramsfail/code/xmlproc-0.70a.zip
[3] http://www.st.cs.uni-saarland.de/whyprogramsfail/code/XMLdata.zip
[4] See the book "Why Programs Fail", section 11.2

```
middle.py:21
middle.py:22
middle.py:23
```

Sample output snippet for the second of the above calls looks as follows:

```
xmlproc/outputters.py:6
xmlproc/xml/__init__.py:1
xmlproc/xml/parsers/__init__.py:1
xmlproc/xml/parsers/xmlproc/__init__.py:1
xmlproc/xml/parsers/xmlproc/charconv.py:103
```

Be sure not to include information about lines executed in the external libraries used by the program being analyzed (i.e., ignore all source files that do not lie within the directory structure of the program being analyzed).

Comparing Coverage (35% of the points)

Create a tool that compares coverage of multiple passing and failing runs. The main file of your tool should be called `diff_coverage.py` and it should be callable as follows:

```
python diff_coverage.py PASSING_SET FAILING_SET OUTPUT_FILE
```

where `PASSING_SET` is the name of the file that contains information about passing runs, `FAILING_SET` is the name of the file that contains information about failing runs (in both those cases, the files should contain filenames of coverage files, look below for an example), and `OUTPUT_FILE` is the name of the file to write the coverage comparison to. A sample call to compare coverage of the passing and failing runs of the `middle` program could look as follows:

```
python diff_coverage.py middle_p.txt middle_f.txt diff.txt
```

with the file `middle_p.txt` having the following content:

```
middle_p1.cov
middle_p2.cov
middle_p3.cov
```

and the file `middle_f.txt` having the following content:

```
middle_f1.cov
middle_f2.cov
```

The output file should be a text file containing information about all the lines covered during one of the runs, as specified by the `.cov` files listed in the input files. Each line in the output file should have the following format:

```
RELATIVE_FILE_NAME:LINE_NUMBER : TESTS / FAILING
```

where `RELATIVE_FILE_NAME` and `LINE_NUMBER` identify the covered line (see previous section), `TESTS` is the percentage of test cases that covered this line (e.g, if there are in total 5 test cases (i.e., coverage files, as in the example above, 3 passing and 2 failing), and 2 test cases (any, passing or failing) covered the line, then this number will be 40%), and `FAILING` is the percentage of failing test cases that covered this line (e.g, if there were 2 test cases that covered this line and 1 of them was a failing one, then this number will be 50%). All numbers should always be rounded towards minus infinity (this is default Python behavior when dividing two integers). Sample output snippet for the call shown above could look as follows:

```
middle.py:11 : 60% / 66%
middle.py:13 : 20% / 0%
middle.py:14 : 20% / 0%s
middle.py:18 : 100% / 40%
```

Graphical Coverage Comparison (15% of the points)

Extend your tool that compares coverage (`diff_coverage.py`) to output a `coverage.html` file with coverage information given in the style of the Tarantula tool (see Jones, Harrold, and Stasko, "Visualization of Test Information to Assist Fault Localization" (ICSE 2002)[5] for details on how to color the output; see figures 3 and 4 for examples). You should output each line (covered or not) of every source file covered. If the line was not covered at all, use some neutral color, like gray. Remember to escape special HTML characters when outputting the lines!

Nearest Neighbour (10% of the points)

Extend your tool for comparing coverage (`diff_coverage.py`) such that it accepts only one failing run, picks the passing run with the most similar coverage, and compares only against this run. For details, see Renieris and Reiss, "Fault Localization with Nearest Neighbor Queries" (ASE 2002)[6], especially section 3.2 describing the distance function to be used. Add a switch to your tool that allows it to use this method, as follows:

```
python diff_coverage.py -nn PASSING_SET FAILING_SET OUTPUT_FILE
```

If the switch `-nn` is present, you can assume that `FAILING_SET` contains only one filename, with the coverage of the failing run.

Your tool should output on the standard output the name of the file with the nearest passing run that it uses, for example:

```
Nearest passing run: middle_p2.cov
```

[5] http://pleuma.cc.gatech.edu/aristotle/Publications/Papers/icse02.pdf
[6] http://www.cs.brown.edu/people/er/papers/renieris-ase2003.pdf

In case there is more than one nearest passing run (i.e., there are several passing runs with the same distance from the failing run), pick the one that was listed first in the `PASSING_SET` file.

Is this technique more effective in pinpointing the failure cause, as compared to the normal coverage comparison?

Call/Location Sequences (10% of the points)

Write two additional tools: `get_sequences.py` that obtains sequences of calls/locations of a given program run on a given input, and `diff_sequences.py` that compares those sequences. For details, see Dallmeier, Lindig, and Zeller, "Lightweight Defect Localization for Java" (ECOOP 2005)[7]; however, do **not** implement the exact method from this paper.

`get_sequences.py` should be callable as follows:

```
python get_sequences.py [-stmt|-call] WINDOW_SIZE OUTPUT_FILE PROGRAM [ARGS]
```

`WINDOW_SIZE` is the size of the window to be used (see the paper above for details). `OUTPUT_FILE` is the name of the file to output the results to. `PROGRAM` is the program to be executed and `ARGS` are optional arguments to be passed to the program. The switch `-stmt` should result in your tool obtaining sequences of locations and the switch `-call` should result in your tool obtaining sequences of calls. You can assume that one and only of those switches will always be present.

The output file should have the following format:

```
SEQ_1_ELEMENT_1
SEQ_1_ELEMENT_2
...
SEQ_1_ELEMENT_n
<empty line>
SEQ_2_ELEMENT_1
...
SEQ_2_ELEMENT_n
<empty line>
SEQ_m_ELEMENT_1
...
SEQ_m_ELEMENT_n
<empty line>
```

Here we have `m` sequences, each having size `n`. Each `SEQ_x_*` is one sequence obtained by your tool. Empty lines serve as separators between sequences. Sample output snippet for the call using `-stmt` switch with window size 2 looks as follows (notice that the path to the file is given relative to the directory from where `get_sequences.py` was invoked, just like for `get_coverage`):

[7] http://www.st.cs.uni-sb.de/papers/dlz2004/

```
xmlproc/xpcmd.py:8
xmlproc/xpcmd.py:28

xmlproc/xpcmd.py:28
xmlproc/xpcmd.py:32

xmlproc/xpcmd.py:32
xmlproc/outputters.py:4
```

Sample output snippet for the call using `-call` switch with window size 3 looks as follows (notice that each output line consists of the location of the function being called and its name—in some cases this is `<module>`, an internal Python name, and in other cases this is the name of the class [not instance!] being created, as in the case of `ConverterDatabase`):

```
xmlproc/xml/parsers/xmlproc/xmlapp.py:9:<module>
xmlproc/xml/parsers/xmlproc/xmlutils.py:3:<module>
xmlproc/xml/parsers/xmlproc/charconv.py:8:<module>

xmlproc/xml/parsers/xmlproc/xmlutils.py:3:<module>
xmlproc/xml/parsers/xmlproc/charconv.py:8:<module>
xmlproc/xml/parsers/xmlproc/charconv.py:103:ConverterDatabase

xmlproc/xml/parsers/xmlproc/charconv.py:8:<module>
xmlproc/xml/parsers/xmlproc/charconv.py:103:ConverterDatabase
xmlproc/xml/parsers/xmlproc/charconv.py:107:__init__
```

`diff_sequences.py` should be callable as follows:

```
python diff_sequences.py PASSING_SEQS FAILING_SEQS OUTPUT_FILE
```

`PASSING_SEQS` is the name of the file with the sequences obtained from a passing run (using `get_sequences.py`). `FAILING_SEQS` is the name of the file with the sequences obtained from a failing run (again using `get_sequences.py`). `OUTPUT_FILE` is the name of the file to put the difference between the two sets to. This file should have the same format as the output file produced by `get_sequences.py` and it should contain all sequences present in the failing set, but absent from the passing set.

How effective is this technique for both locations and calls sequences? Which kind of location and which length do you find most effective?

Submission

Submit your solution as a **.zip** file to `wasylkowski@st.cs.uni-sb.de` with `[Project 2]` in the subject. Provide your full name and matriculation number in the body of the email. Be sure to adhere **exactly** to the input and output specification as given above.

The deadline is **2008-12-19 23:59**.