Project 3: Detecting Anomalies

In this project, you will write a tool that collects invariants during a program run and reports violations of the invariants learned. Your task is to implement the tool and apply it to the XMLProc parser from Project 1[1]. The parser comes with a failing test input `demo/urls.xml`. Apply your tool to the run of XMLProc on that input. Do invariant violations help in discovering the cause of the failure? If so, which ones?

Collecting Invariants and Reporting Violations (60% of the points)

Overview

Create a tool that collects invariants from a a given program run on a given input, and at the same time reports all violations of the invariants learned. You will implement a simplified version of the DIDUCE tool (see Hangal and Lam, "Tracking Down Software Bugs Using Automatic Anomaly Detection" (ICSE 2002)[2]). The main file of your tool should be called `pyduce.py` and it should be callable as follows:

```
python pyduce.py PROGRAM [ARGS]
```

`PROGRAM` is the program to be executed and `ARGS` are optional arguments to be passed to the program. A sample call to analyze invariants of the XMLProc parser looks as follows:

```
python pyduce.py xmlproc/xpcmd.py input.xml
```

Your tool should output invariants' violations (more on that below).

Tracing Program Run

For a given program and its arguments, your tool should run the program on those arguments and trace[3] all lines executed in the given program. Take care not to trace lines executed in the external libraries used by the program being analyzed (i.e., ignore all lines located in source files that do not lie within the directory structure of the program being analyzed). For each traced line, your tool should update invariants associated with all local variables[4].

[1] http://www.st.cs.uni-saarland.de/whyprogramsfail/code/xmlproc-0.70a.zip

[2] http://suif.stanford.edu/papers/Diduce.pdf

[3] use Python's sys.settrace function for this

[4] You can access those via the frame parameter of the tracing function you will create

Updating Invariants

Invariants should be associated with combinations of variables *and* source code locations where these are accessible. More precisely, instead of keeping track of invariants of the form "`x` is always between 2 and 5" your tool should keep track invariants of the form "`x` is always between 2 and 4 in file `tooldir/main.py` at line 8" and "`x` is always between 3 and 5 in file `tooldir/main.py` at line 9".

An invariant for each variable and a location in the source code where it is accessible is represented by two integer values V and M. V is the initial value first encountered for this variable at this place (it should be set only once). M is a mask representing a range of values (the ith bit is 0 if a difference was found in the ith bit, and 1 if the same value has always been observed for that bit). Formally, if the first value of a variable is W, then $M := \neg 0$ and $V := W$ hold. With each new encountered value W'[5], the mask M becomes $M := M \wedge \neg(W' \otimes V)$, where $\otimes$ is the exclusive-or operation.

The following is an example. If some variable i is first encountered at some location with a value of 16, then $V = 16 = 1000$ (in binary representation) holds (M is initially $\neg 0 = 11111$[6]). If i is later encountered *at the same location* with a value of 18, V is still unchanged, but in M the second bit is cleared because the difference between V and 18 is the second bit. Thus, M becomes 11101.

Your tool should keep track of invariants for all booleans, integers, longs and instances of classes. In order for this to be possible, you must first convert each variable to an integer (such that the representation based on V and M described above is applicable). For booleans and integers this is straightforward—just use `int(x)` where `x` is the value. For longs you should take a hashcode of the value (using Python's built-in `hash` function), and for instances of classes you should take the hashcode of the class (*not* the instance!). This way all values are represented as integers.

Reporting Invariants' Violations

Your tool should report every change of an invariant. On each invariant update, it should check if the mask M remains the same or if it needs to be changed (i.e., the current invariant has been violated and it needs to be relaxed). If M needs to be changed, this should be reported in the standard output. A sample output snippet looks like this:

```
Invariant violated at xmlproc/xml/parsers/xmlproc/xmlutils.py:650
    Old invariant: sum == 0
    New invariant: 0 <= sum <= 64
Invariant violated at xmlproc/xml/parsers/xmlproc/xmlutils.py:646
    Old invariant: 48 <= char <= 52
    New invariant: 32 <= char <= 117
```

You can extract the ranges from V and M by manipulating and combining those values in an appropriate way.

[5] at the same location in the source code!

[6] actually, for a 32-bit integer there would be 32 1's, but we'll stick to 5 for simplicity

Collecting Invariants for Call Sites (20% of the points)

Extend your tool such that it collects invariants for callees' arguments at call sites. This should allow your tool to represent invariants of the form "The argument `x` is always between 10 and 15 at call to `foo` in file `tooldir/main.py` at line 85". You can extend the tracing function to handle calls, but take care to differentiate between the caller and the callee: the tracing function gets informed about the call not at the call site, but where the callee starts. It is your job to find an appropriate way to associate the arguments with the call site and not the callee.

Filtering Invariants' Violations (20% of the points)

Hangal and Lam in their paper introduce the notion of an *invariant confidence*. Implement this notion in your tool and only output invariant violations that result in a confidence changing by more than a given threshold (this should be a constant value in your source code). Your output should now contain the confidence drop given as an information about a violation, for example:

```
Invariant violated at xmlproc/xml/parsers/xmlproc/xmlutils.py:622
Violation confidence drop: 1
    Old invariant: pos == 3
    New invariant: 3 <= pos <= 55
Invariant violated at xmlproc/xml/parsers/xmlproc/xmlutils.py:623
Violation confidence drop: 3
    Old invariant: start == 2
    New invariant: 2 <= start <= 54
```

Which threshold value do you find most effective? How many invariants' violations remain after you apply the filtering? Are these enough to discover the reason for the failure in the XMLProc parser run on the `demo/urls.xml` file?

Submission

Submit your solution as a **.zip** file to `wasylkowski@cs.uni-saarland.de` with `[Project 3]` 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. Include the answers to the questions stated in the assignment in the body of the email. If you have any questions or need clarification regarding the assignment, send an e-mail to `wasylkowski@cs.uni-saarland.de`.

The deadline is **2009-01-09 23:59**.