CS239: Delta-Debugging Project

Xinyu Wang, Zhao Weng, Zhaowei Tan

June 6, 2018

Abstract

This report reviews our project of building *ddmin* debugger in Python programming language. We describe the overview of our debugger, including the optimization we implement, and then use a few show cases to demonstrate the usage of our tool.

1 Introduction to *ddmin*

ddmin [1] minimizes the test case which reflects the same error as the original test case does. Input of *ddmin* is a function, data to be tested, and granularity. Granularity is used to indicate number of subsets to divide. The program of *ddmin* has a while loop to keep processing of function.

First, the program will divide the data into subsets whose length is specified by granularity. If one of the subset throws the same error as the original data, that means only that specific subset contain the error we are interested in. Data are assigned the value of that subset and continue the next loop round.

If all subsets cannot throw the same error as the original data, the complement of each subset will be extracted out to test whether it can cause the error or not. If the specific complement can cause the same error, granularity will be decremented by 1 and maintain the value if the value drops below 2. Granularity is decremented by 1 because we need to keep the subset size in the next round the same after one subset is left out. After decrementing granularity, data will be assigned with value of the complement and continue the next loop round.

If all complements cannot cause the same error either, granularity will be first checked against length of the data. If granularity is the same as length of data, that means data cannot be split into smaller pieces and one of those pieces can result in the same error. No single unit of data can be left out to cause the error we are interested. In that case, data at current round will be returned. Otherwise, granularity will be doubled so that the data can be cut into even smaller and more pieces to try for errors. To avoid the case the number of pieces surpass the length of the data, granularity will be upper bound by the length of data.

2 Implementation and Optimization regarding *ddmin* in Python

Implementation We write xxx code in Python, what's the overall architecture . [Describe the workflow here] The program first checks and pinpoint the error by running the program and then run xx and then xxx to minimize.

Granularity improvement. We found that *ddmin* is actually not optimal because granularity is doubled if both all subsets and complements cannot generate the same error as the original. *ddmin* can work optimally only if the size of the original problem is 2's power. Some divisions not of one over 2's power size won't be checked. We tuned *ddmin* so that granularity is incremented by 1 as well as multiplied by 1.5, and found that *ddmin* works better if granularity is incremented by 1 although it is still not optimal. But in that case, *ddmin* can cover more divisions of the original program and has better capability to minimize the original test case.

Infinity loop. Describe the problem and the solution

3 Evaluation

findSubset description. Given the number of pieces to cut into and the original set, findSubset will output an array of subset with the size of each subset equal to the lower bound of length of the original set divided by number of pieces.
findComplement description. Given the number of pieces to cut into, array of subsets, and specific index of the subsets array which should be skipped, findComplement will output the complement subset with the specified subset left out from the original subsets.
Testcase. Supported Exception Type: StopIteration, OverflowError, FloatingPointError, ZeroDivisionError, AssertionError, IndexError, KeyError, UnboundLocalError, notImplementedError
Unsupported Exception Type. SyntaxError, NameError, TypeError, ModuleNotFoundError, ImportError. We choose not to support these exception types because these exception type will happen when ddmin try to divide the program. It is hard to distinguish between the cases in which the original program has the specific error and the error is actually caused by ddmin while dividing the program.
Test case Example Input.

- DivideZeroError: bubble sort with zero division
- AssertionError: Sum of 1, 1/2, 1/3 ... to 1/10 with assertion.
- OverflowError: Sum of 1, 1/2, 1/3 ... to 1/10 with overflow.
- IndexError: Sum of 1, 1/2, 1/3 ... to 1/10 with index violation.

Type perspective. 1. Input data is array of primitive types, we want to make sure *ddmin* can find out the smallest subset that causes the same error as the original input array 2. Input data is a python program without calling other functions. Input python program can

cause different types of exceptions. For each specific exception, our ddmin algorithm can first catch the type of that exception and find the minimized piece of the program which will throw that specific exception.

3. Input data is python program with several layers of calling stack. Exception is involved in a specific layer of function call. Our *ddmin* program should catch the type of the exception and find the minimized piece of the program which will throw that specific exception.

Size perspective. 1. Our *ddmin* program is able to find minimized test case that causes the same error as the original program when the program size is relatively small (within 30 lines of code)

2. Our *ddmin* program is able to find minimized test case that causes the same error as the original program when the program includes complex logic and class inheritance.

3. Our *ddmin* program is able to find minimized test case even when the script being input for testing calls a external library in the code.

References

[1] Zeller, A., and Hildebrandt, R. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering 28*, 2 (2002), 183–200.