README.md

**Table of Content**

1. Introduction of Tool
2. Requirements for Tool
3. Features of Tool
4. How to Work
5. Folders Description
6. Toy examples
7. **Introduction**

Dynamic Analyzer tool has leverage the capabilities of LibClang to analyze the dynamic aspects of C language programs. LibClang is considered as a stable high level C interface to clang and used for high level abstractions by traversing an AST through a cursor.

1. **Features:**

Besides dynamic aspects of a program, the tool would help to obtain the following features as follow.

* To extract the name, mangled name and qualified name of functions
* To extract the function definitions with its line number of source code.
* To extract the type of a function such as Template function, a method or normal function
* To extract the calling points location of a function along with parameters
* To extract the cyclomatic complexity of each function
* To extract the longest functions call complexity
* To accumulate the total cyclomatic complexity

1. **Requirements**

* llvm (Preferable llvm-11.0.0)
* Clang
* Python ( Supports both Python 2 ad 3)
* Conda
* Ubuntu/macOS

1. **How to Work**

Firstly open the Terminal and clone the repository as follow.

**>** git clone https://github.com/HPCL/code-analysis.git

Code-analysis folder will be created on system current root path. It will be open by applying the following command

**>** cd code-analysis

In code-analysis folder execute the environments.yml file as follow.

**>** conda env create –f environments.yml

The content of environments.yml is covering the dependencies which are used to execute the Dynamic analyzer tool smoothly. If you are executing environments.yml first time and initially dependency requirements are not available on your system then it will take few seconds.

To check the status of all tests, apply the following commands

**>** cd /Users/…/code-analysis/Dynamic/tests

**>** pytest

To execute a specific test, apply the following command and observe the output

**>** pytest –q –s –name example3.c test\_Function\_List.py

This command is passing a file example3.c as an argument with reference –name to test\_Function\_List.py. In case of passing more than one arguments with references, you can update conftest.py file in tests folder.

1. **Folders Descriptions**

The hierarchy of clone folder is as follows

Code-analysis

------Dynamic

---------- examples

---------- src

---------- tests

---------- README.md

---------- requirments.txt

------Static

------\_\_init\_\_.py

-------environments.yml

* Src Folder:

This folder contains the python scripts which are used to cover the different aspects of dynamic analysis.

* tests Folder

This folder contain three types of files as follows

* Tests script written in python and can be executed through pytest.
* Toy examples of \*.c to understand the functionality of scripts of src folder and are passed as an argument when a test is checked through pytest.
* Conftest.py file is used to declare the arguments reference which are passed through pytest. Initially, --name reference is given for passing \*.c file as argument of pytest command.

1. **Toy Examples**
   1. Example 1:

//Example2 in tests folder

void Large(int a, int b)

{

if(a>=b)

printf("large number is %d", a);

else

printf("Large number is %d", b);

}

void main() {

int A=2, B=3, Larg;

if(A == B)

printf("Both number are same");

else

Large(A, B);Large(5,4);

printf("Bye");

}

When test\_Function\_list.py file of tests folder is executed with pytest as follows

> pytest –q-s –name example2.c test\_Function\_list.py

Then the following output will displayed.

**Output:**

[‘File Name’, ‘Function Name’, ‘Mangled Name’, ‘Function Type’]

[‘example2.c’, ‘Large’, ‘\_\_Z5Largeii’, ‘Function’]

[‘example2.c’, ‘main, ‘\_main, ‘Function’]

Similarly, when test\_Function\_Definition\_Location.py file of tests folder is executed with pytest as follows

> pytest –q-s –name example2.c test\_Function\_Definition\_Location.py

Then the following output will displayed.

**Output:**

[‘File Name’, ‘Function Name’, ‘Function Qualified Name’, ‘Function Definition’, ‘Line Number’, ‘Column Number’]

[‘example2.c’, ‘Large’, ‘Large’, ‘void Large(int a, int b)’, ‘1’, ‘6’]

[‘example2.c’, ‘main, ‘main, ‘void main() {’, ‘9’, ‘6’]

Moreover, when test\_Function\_Compleity.py file of tests folder is executed with pytest as follows

> pytest –q-s –name example2.c test\_Function\_Complexity.py

Then the following output will displayed.

**Output:**

[‘File Name’, ‘Function Name’, ‘Function Qualified Name’, ‘Function Definition’, ‘Mangled Name’, ‘Name Space’, ‘Line Number’, ‘Column Number’, ‘Cyclomatic Complexity’]

[‘example2.c’, ‘Large’, ‘Large’, ‘void Large(int a, int b)’, ‘\_\_Z5Largeii’, ‘Anonymous Namespace’, ‘1’, ‘6’]

[‘example2.c’, ‘main, ‘main, ‘void main() {’,’\_main’, ‘Anonymous Namepspacpe’, ‘9’, ‘6’]

Subsequently, when test\_Called\_Calling\_Function.py file of tests folder is executed with pytest as follows

> pytest –q-s –name example2.c test\_Called\_Calling\_Function.py

Then the following output will displayed.

**Output:**

[‘Calling Function Name’, ‘Function Definition’, ‘Line#’, ‘Column#’, ‘Name Space’, ‘ Called Function Qualified Definition’, ‘Line#’, ‘Column#’ ]

[‘main’, ‘Anonymous Namespace’,’void main(){\n’, ‘9’, ‘6’, ‘Large’, ‘Large(A, B); Large(5, 4);\n’, ‘14’, ‘4’]

[‘main’, ‘Anonymous Namespace’,’void main(){\n’, ‘9’, ‘6’, ‘Large’, ‘Large(A, B); Large(5, 4);\n’, ‘14’, ‘16’]

Similarly, when test\_Function\_Definition\_Calls.py file of tests folder is executed with pytest as follows

> pytest –q-s –name example2.c test\_\_Function\_Definition\_Calls.py

Then the following output will displayed.

**Output:**

[‘Function Name’, ‘Name Space’, ‘Calling Point’, ‘Line#’, ‘Column#’]

[‘Large’, ‘Anonymous Namespace’, ‘Large(A, B); Large(5, 4);\n’, ‘14’, ‘4’]

[‘Large’, ‘Anonymous Namespace’,, ‘Large’, ‘Large(A, B); Large(5, 4);\n’, ‘14’, ‘16’]