# User Manual - CLARA

Maheen Riaz Contractor

May 18, 2022

## Contents

**1 Overview of CLARA** **2**

**2 Dependencies and Requirements of CLARA** **2**

**3 Running CLARA** **2**
3.1 App Start-up . . . . . 2
3.2 Command Line Flags and Options . . . . . 2
3.3 Processes . . . . . 3

**4 Files Changed/Created** **7**
4.1 clara.py . . . . . 7
4.2 repair.py . . . . . 7
4.3 graph_matching.py . . . . . 8
4.4 parser.py . . . . . 8
4.5 py_parser.py . . . . . 9
4.6 convert_to_py.py . . . . . 9
4.7 feedback_repair.py . . . . . 10
4.8 ilp.py . . . . . 10
4.9 interpreter.py . . . . . 10
4.10 matching.py . . . . . 10
4.11 model.py . . . . . 11
4.12 py_interpreter.py . . . . . 11

**5 Script Files for Experimentation** **12**
5.1 clara_script.py . . . . . 12
5.2 clara_extract_incorrect_data.py . . . . . 12
5.3 makeBoxPlots.py . . . . . 12

# 1 Overview of CLARA

CLARA is an automated CLuster And RepAir tool created in 2016. Its code[1] has mostly been developed by Ivan Radiček and can work on three langauges, Java, C++, and Python. Each language has its own parser which converts the source code into CLARA's model and an interpreter which executes the model's expression. The model is CLARA's internal representation of the source code of a program. CLARA's source code is written in Python.

# 2 Dependencies and Requirements of CLARA

1. Linux environment, preferably Ubuntu 18.X
2. Python 3.X
3. gcc
4. cython
5. lpsolve 5.5

# 3 Running CLARA

## 3.1 App Start-up

To run CLARA:

1. Run `make` every time changes are made to the source code
2. Run `export LD_LIBRARY_PATH=/usr/lib/lp_solve/` every time the terminal is restarted
3. Run `clara correctFilePath incorrectFilePath -argsfile testcaseFolderPath`
4. Append test case filename without the `.txt` extension as a comment at the end of the incorrect file. For example, if the incorrect file fails using the file 2.txt, add # 2 at the end of the incorrect file.

## 3.2 Command Line Flags and Options

- entryfnc : entry function for matching or repair, (default is 'main')
- verbose : controls the amount of output information printed, (default is 0)
  - 0 -> tool standard output is suppressed
  - 1 -> tool standard output is not suppressed
- lang : language of the source files, (default is to guess from the first source file extension)
  - java -> Java
  - cpp -> C++

[1] `https://github.com/iradicek/clara`

  - py -> Python
- args : arguments for matching or repair, need to be in square brackets
- argsfile : test case file or file with arguments for matching or repair
- timeout : timeout in seconds (for the repair), (default is 300)
- checkAllRep : denotes whether repair should be checked against all other possible test cases available in the file path
  - True -> Should check
  - False -> Should not check
- matchOp : graph matching algorithm version, (only for graph matching)
  - 0 -> run normal CLARA repair
  - 1 -> run graph matching, taking into account labels and edges and rebuild the program using the correct program's edges.
  - 2 -> run graph matching, taking into account only labels and rebuild the program using the correct program's edges.

### 3.3 Processes

1. repair : runs the repair process. Figure 1 shows an example of a possible input, command and feedback for the repair process.
2. match : runs the matching process. Figure 2 shows an example of a possible input, command and feedback for the match process.
3. graph : runs the repair process using the graph matching algorithm. Figure 3 shows an example of a possible input, command and feedback for the graph process.
4. convert : takes source code and converts it into python (does not support loops and some other expressions), (saves the python code into 'c.py')
5. model : converts the programs into CLARA's model
6. eval : takes in a program and prints its trace
7. cluster : creates a cluster of all the programs within the given cluster directory
8. feedback : generates feedback amongst multiple programs

```
print(val % 2 == 0)
```

(a) Correct program (eg/cap1.py)

```
print(val % 1 == 0)
# 1
```

(b) Incorrect program (eg/cap2.py)

```
clara repair eg/cap1.py eg/cap2.py --argsfile "eg/tests/"
    --checkAllRep 1 --verbose 0
```

(c) CLARA command

```
Test Case Available
Locs Same
Cost: 1.0
Repairs:
  * Change 'print(((a % 1) == 0))' to 'print(((a % 2) == 0))' at
    line 12 of the incorrect program and at line 17 of the correct
    program (cost=1.0)
Locs in Correct Program Model 2
Exprs in Correct Program Model 3
Locs in Incorrect Program Model 2
Exprs in Incorrect Program Model 3
Number of Repairs 1
Percentage of the model modified 16.666666666666664
No further repair needed after completeing the previous ones!
Locs Same
No repair!
All Repaired
```

(d) CLARA Repair feedback

Figure 1: Example of CLARA's repair process

```python
a = int(input())
ans = 0
b = 2
ans = a + b
print(ans)
```

(a) Program 1 (eg/cap1.py)

```python
ans = 0
a = int(input())
b = 1
ans = a + b + 1
print(ans)
# 1
```

(b) Program 2 (eg/cap2.py)

```
clara match eg/cap1.py eg/cap2.py --argsfile "eg/tests/" --verbose 1
```

(c) CLARA command

```
Test Case Available
[debug] Guessed language: py
[debug] Reading and parsing source file 'eg/cap1.py'
[debug] Reading and parsing source file 'eg/cap2.py'
[debug] Programs executed, matching traces
[debug] Couldn find match for main-1-b
No match!
```

(d) CLARA Match Output

Figure 2: Example of CLARA's match process

```python
a = int(input())
ans = 0
b = 2
ans = a + b
print(ans)
```

(a) Program 1 (eg/cap1.py)

```python
ans = 0
a = int(input())
b = 2
for x in range(0, a):
    ans += x
ans += b
print(ans)
# 1
```

(b) Program 2 (eg/cap2.py)

```
clara graph eg/cap1.py eg/cap2.py --argsfile "eg/tests/" --checkAllRep 1
--verbose 0 --matchOp 1
```

(c) CLARA command

```
Test Case Available
Score: 0.7999999999999999
 {0: 0, 1: 1, 2: 5}
Locs Deleted: 3
Cost: 3.0
Repairs:
  * Delete '$cond = (ind#0 < len(iter#0))' at line 14 of the incorrect
    program (cost=2)
  * Delete 'ans = (ans + b)' at line 16 of the incorrect program (cost=2)
  * Delete 'x = iter#0[ind#0]' at line 14 of the incorrect program
    (cost=2)
  * Delete 'ind#0 = (ind#0 + 1)' at line 14 of the incorrect program
    (cost=2)
  * Delete 'ans = (ans + x)' at line 15 of the incorrect program (cost=2)
  * Change 'ans = 0' to 'ans = (a + b)' at line 11 of the incorrect
    program and at line 19 of the correct program (cost=3.0)
Locs in Correct Program Model 2
Exprs in Correct Program Model 5
Locs in Incorrect Program Model 2
Exprs in Incorrect Program Model 7
Locs in Old Incorrect Program Model 5
Exprs in Old Incorrect Program Model 12
Number of Repairs 1
Percentage of the model modified 26.923076923076923
No further repair needed after completing the previous ones!
Locs Deleted: 3
No repair!
All Repaired
```

(d) CLARA Graph Matching Output

Figure 3: Example of CLARA's graph process

# 4 Files Changed/Created

## 4.1 clara.py

This file is responsible for running the processes of CLARA. It contains definitions for each process and calls its corresponding functions for repair, match, graph, etc. The function $saveRepairs$ saves the feedback from a successful repair. For testing, change the path accordingly
Functions Edited/Created:

- repair
- graph
- process_source
- process_sources
- findTestCaseNo
- checkRepairs
- saveRepairs
- printLocExpValues

## 4.2 repair.py

This file contains the functions involved in the repair process. It uses the two models and their corresponding test case to generate repairs. Functions Edited/Created:

- __init__
- gettrace
- repair
- filter_potential
- noCostAllLoc
- repair_fnc
- one_to_ones
- potential
- applyRepairs
- removeValFromResult
- findAndSub
- substituteInFunc
- preprocessTrace
- calculateRepairPercentage

## 4.3 graph_matching.py

This is a new file that contains the functions for the graph matching process which helps eliminate the control flow problem. Functions Created:

- jaccard
- __init__
- createGraphs
- createLocLabelDict
- connect
- getLableValue
- makeLabel
- makeGraph
- printGraph
- createMatchDict
- findBestMatch
- permBackTrack
- createModel

## 4.4 parser.py

This file is used for model generation. It is a general parser file for important language constructs and executes functions from specific language parsers. Functions Edited/Created:

- visit
- add_import
- add_from_import
- add_ret_loc
- add_print_loc
- getlangparser

## 4.5 py_parser.py

This file contains functions which are specific to parsing Python. It takes in a python file, creates the abstract syntax tree using the $ast$ module and helps parse each node. We updated it to include the changes mentioned in the paper. Functions Edited/Created:

- __init__
- parse
- addargs
- visit_Module
- visit_Main
- visit_FunctionDef
- visit_Constant
- visit_Assign
- visit_AugAssign
- visit_Call
- visit_Import
- visit_ImportFrom
- visit_Return

## 4.6 convert_to_py.py

This is a new file which takes expressions which are in model form and converts them into python expressions that when modeled, return to the original expression. This file helps pretty print CLARA's feedback. It can not convert loops and conditionals that are not $ite$ expressions. Functions Created:

- convertExp
- makefunc
- getExprs
- process_sources
- findTestCaseNo
- checkRepairs
- saveRepairs
- printLocExpValues

## 4.7 feedback_repair.py

This file is used to process the repairs returned by CLARA and convert it into actual feedback. We updated it to use convert_to_py to process the repairs and to never have feedback containing the same variable in both sides of an expression, e.g. $a = a$. Functions Edited/Created:

- __init__
- genRemovedLocFeedback
- genConvertedfeedback

## 4.8 ilp.py

This file contains the linear programming solver for CLARA to take all of the costs per location and find the best mapping between variables. Functions Edited:

- encode_P
- decode_model

## 4.9 interpreter.py

This file contains the general interpreter for the model. It takes a model expression and passes it to the corresponding interpreter for that language. We updated it to include the main, input, and other library functions. Functions Edited/Created:

- __init__
- getfnc
- run
- execute_Op
- execute_FuncCall
- getlanginter

## 4.10 matching.py

This file two models and checks if they match. They should have the same control flow and for every variable in one model there should be a corresponding variable in the other model. Both should hold the same value at every point in the programs. Functions Edited/Created:

- match_mems
- match_traces
- match_struct
- match_programs

## 4.11 model.py

This file takes in a program and uses the parser to generate a model. The model is CLARA's internal representation of the program code. We update it to include the changes mentioned in our paper. Functions Edited/Created:

- getUnprimedVersion
- getPrimedVersion
- Expr -> __init__
- add_import
- add_from_import
- addfnc
- add_ret_loc
- add_print_loc

## 4.12 py_interpreter.py

This file works with *interpreter.py* to execute the model in Python. We updated it to include main, input, and builtin functions. Functions Edited/Created:

- execute_builtin_fnc
- execute_lib_fnc
- execute_math_log
- execute_values
- execute_strip
- execute_split
- execute_lower
- execute_upper
- execute_startswith
- execute_find
- execute_capitalize
- execute_swapcase
- execute_title

# 5 Script Files for Experimentation

## 5.1 clara_script.py

Given a list of correct and incorrect program names, this script runs the graph matching process.

- *correct*: The array *correct* stores the names of all the correct programs involved in the repair. For example,

```
correct = [
    "150875064",
    "144834824",
    "149760373",
    "136939961",
    "150604669"]
```

- *path* : The string *path* is the path of the main folder containing the testing files
- *problem_name* : Contains the name of the problem we are testing, e.g. '977C'
- *testcase* : Contains the path of the folder where the testcases are located
- *ic* : Contains the path to the file where the names of the incorrect files involved in the experiments are located

## 5.2 clara_extract_incorrect_data.py

This file takes in the path of the folder where the results are located by running *clara_script.py* and creates a summary of the data.

## 5.3 makeBoxPlots.py

This file takes in the path to the results and creates box plots for each problem. An example of the usage is

$$python3\ makePerBoxPlots.py\ --path\ '/Thesis/977C/'\ --plots\ 1\ --prob\ 977C$$

The flag *path* contains the path to the results folder. *plots* can hold two options, 0 or 1 where 0 indicates that a plot should not be created and 1 indicates it should. Lastly, *prob* contains the name of the problem.