Problem A: Smart EC: Program-Building for Name Mapping

Topic Chairs: Chi-An (Rocky) Wu, Ching-Yi Huang, and Chih-Jen (Jacky) Hsu

Cadence Design Systems, Inc.

I. Introduction

In the ASIC design flow, implementation tools change the names of design components to comply with the implementation rules while still keeping the information to track the design intention. For example, tools change the name "a[0]" into "a_0_" to follow the rule: “no special character”. Meanwhile, name mapping plays an important role in verification tools because good name mapping can help verification tools efficiently and correctly verify designs. Although different stages, tools, and settings adopt different rules of name changing, there are always some simple ways to map the changed names back to the original names, and humans can easily tell the mapping rules/relations between the original names and the changed names. Nevertheless, it is difficult for machines/tools to solve the mapping ‘automatically’.

In this contest, we formulate a problem of **program-building for name mapping**. Contestants shall write a program that accepts a given set of mapping relations and generate a Python script. Then, the Python script can generate the same mapping result. The smaller size of the generated script is the better in this problem.

II. Background

In the implementation flow, a design is continuously optimized from one stage to another stage. During these stages, the tools would change names of design components, such as modules, instances, pins/ports, sequential elements, and nets for satisfying the implementation rules; some simple examples are string extension, symbol transformation, dimension transformation, and name mangling [1]. For formal equivalence checking or Engineering Change Order [2][3], name mapping is important to identify the design intent from the final circuit.

Name mapping itself is an interesting topic. Usually, humans can easily recognize the mapping rules/relations between the original names and the changed names. However, it is hard for machines/programs to solve the mapping automatically. Take Figure 1 as an example, humans can easily recognize the mapping relations “dog<->0”, “cat<->1”, and “no special character” behind the sets of names for *Example. 1*, and easily recognize the relations “binary to decimal” and “special character to '_'” for *Example 2*. However, machines/programs may need to do some complicated handling on the names for mapping. For example, the program would need to do string partition with the symbols ' [', '\', '_', symbol transformation like '[', ']' to '_', token recognition and string manipulation like "dog" to "0", and dimension transformation/binary-to-decimal like {1, 1} to 3.

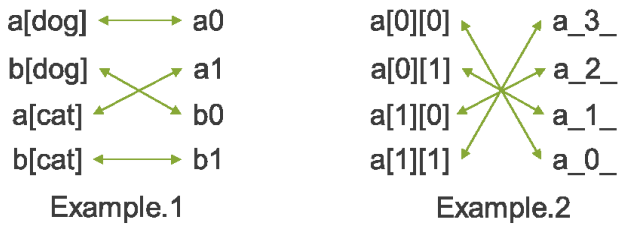

Figure 1. Examples of name mapping.

As designs become more complicated, and as EDA tools keep advancing, more and more name changing rules would be involved in the implementation processes. Then, the name mapping problem becomes more and more complicated and challenging. Therefore, in this contest, we formulate a problem of **program-building for name mapping** to stimulate academic ideas for solving name mapping problems. In particular, we encourage contestants to apply AI techniques to solve this problem.

III. Contest Objective

The objective of this contest is to develop a smart and automatic program for building name mapping program/script. In this contest, we provide industrial name mapping cases to evaluate contestants' programs. With these cases, we look forward to innovative approaches that can be utilized in industrial tools.

IV. Problem Formulation and Input/Output Format

Given a set of mapping relations MR_{given} = {"G1"<->"R1", "G2"<->"R2", …, "GN"<->"RN"} between the first set of names {"G1", "G2", … , "GN"} and the second set of names {"R1", "R2", … , "RN"}, your main program must output a **Python script** that can accept two sets of names and output a mapping result MR_{out} such that $MR_{out} \equiv MR_{given}$, which means the mapping relations in MR_{out} are the same as that in MR_{given}. Figure 2 shows an example of the problem of program-building for name mapping.

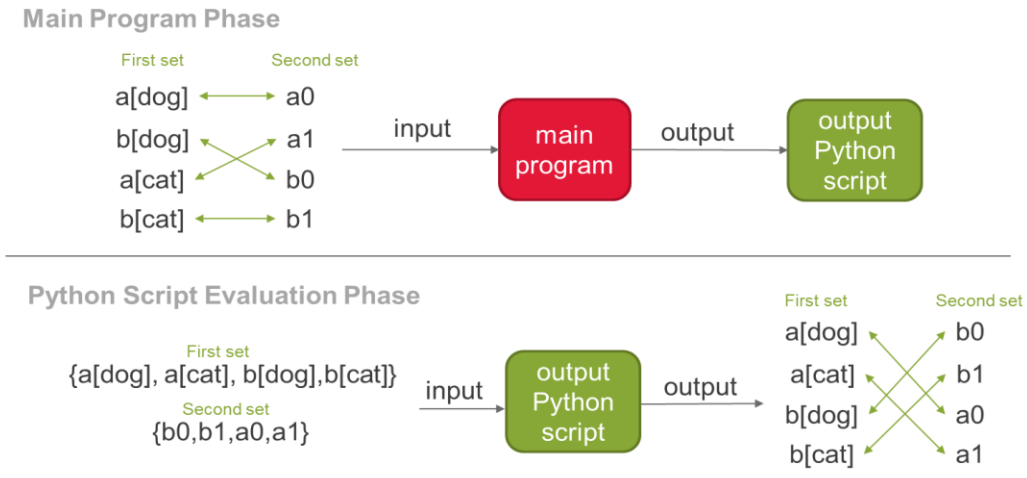

Figure 2. Program-building for name mapping for Example. 1.

Program Requirement:

Main program:

The requested program must be run on a Linux system. The time limit of running each testcase is 1800 seconds. Parallel computation with multiple threads or processes is not allowed. The executable file should be named "*nmpgen*" and accept two arguments:

./nmpgen <map_in.json> <python_script.py>

<*map_in.json*> is an input file describing the mapping result of two sets of names.
<*python_file.py*> is an output file and is a Python script in Python 3.4.0.

Input Format

Input mapping file <*map_in.json*> uses the format of JSON object to describe the mapping pairs:

```
{
  "G1" : "R1",
  "G2" : "R2",
  "G3" : "R3",
  ...
  "GN" : "RN"
}
```

The above JSON file means the name `"G1"` in the first set of names maps to the name `"R1"` in the second set of names, `"G2"` maps to `"R2"`, and so on. Only 1-to-1 mapping will appear.

Output Format

The Python script <*python_script.py*> must be a valid Python 3 script.

Python script:

The generated Python script <*python_script.py*> by the main program must follow the format of Python 3.4.0. The python script can only use official packages. The time limit of running the Python script is 900 seconds. The python script should accept two arguments:

python3 <python_script.py> <name_in.json> <map_out.json>

<*name_in.json*> is an input file describing two sets of names.
<*map_out.json*> is an output file describing the same mapping pairs as that in <*map_in.json*>.

Input Format

<*name_in.json*> uses the format of JSON array.

```
[
   ["G3", "G1", "G2", … , "GN"],
   ["R2", "R3", "R1", … , "RN"]
]
```

The above JSON file means the first set of names is `{"G3", "G1", "G2", …, "GN"}`, and the second set of names is `{"R2", "R3", "R1", …, "RN"}`.

The sizes of these two arrays are the same.

Output Format

Output mapping file <*map_out.json*> also uses the format of JSON object:

```
{
  "G2" : "R2",
  "G1" : "R1",
  "G3" : "R3",
  ...
  "GN" : "RN"
}
```

The above JSON file means `"G1"` maps to `"R1"`, `"G2"` maps to `"R2"`, and so on. The first set of names in <*name_in.json*> must be at the "key" positions (left) in the JSON object. and the second set of names in <*name_in.json*> must be at the "value" positions (right) in the JSON object.

We strongly recommend contestants to use Python JSON package [4] for dealing with the JSON format.

Example

Given the mapping file *map_in.json*:

```
{
  "a[dog]" : "a0",
  "b[dog]" : "b0",
  "a[cat]" : "a1",
  "b[cat]" : "b1",
  "a[0][0]" : "a_0_",
  "a[0][1]" : "a_1_",
  "a[1][0]" : "a_2_",
  "a[1][1]" : "a_3_",
}
```

map_in.json

We run the program with

```
./nmpgen map_in.json map.py
```

Then we evaluate the Python script *map.py:*

python3 map.py names.json map_out.json

```
[
  ["a[dog]", "a[cat]", "a[0][0]", "a[0][1]", "b[dog]", "b[cat]", "a[1][1]",
"a[1][0]" ] ,
  ["a0", "a1", "b0", "b1", "a_3_", "a_2_", "a_0_", "a_1_"]
]
```

names.json

```
{
  "a[0][0]" : "a_0_",
  "a[0][1]" : "a_1_",
  "a[1][0]" : "a_2_",
  "a[1][1]" : "a_3_",
  "a[dog]" : "a0",
  "b[dog]" : "b0",
  "a[cat]" : "a1",
  "b[cat]" : "b1"
}
```

map_out.json:

We check if *map_out.jso*n is the same mapping pairs as the given *map_in.json*.

In this example, the mapping pairs in *map_out.json* are correct as that in *map_in.json*, so the size of the script will be compared to other teams.

V. Evaluation Method

For each case, the result will be evaluated by the following criteria:

1. Correctness: The main program and Python script must be executed without crash. The main program and Python script must follow the requirement mentioned in Section IV. The generated python must follow Python 3.4.0 format and can only use official packages. The generated python must output exactly correct mapping file that follows the output format mentioned in Section IV. Any violation gets score of 0 for that testcase.
2. Time limit: The main program must finish within 1800 seconds, and the Python script must finish within 900 seconds; otherwise, the team gets score of 0 for that testcase.
3. Scoring according to the rank: The teams that pass the above correctness and time limit checking get their scores by their ranks for that testcase. The teams with the rank 1~6 will get scores of {10, 7, 5, 4, 3, 2}, respectively. The remaining teams get a score of 1. Teams are ranked based on the following criteria:
   a. We rank teams according to size of Python script.
   b. The size of the Python script is defined as **the characters in the codes except for the space character**. The smaller is better.
   c. If the size ties, we rank them according to the elapsed time of the learning program. The faster is better.

VI. Testcase

Several testcases will be announced soon.

However, please note that these public testcases will not be included in the final qualification. All testcases in the final qualification are hidden testcases.

VII. Python Script Example

Input:

```
[["G3","G1","G2"],["R1","R2","R3"]]
```

Output:

```
{"G3": "R3", "G1": "R1", "G2": "R2"}
```

Script 1:

```
==================================================================
import sys,json
i=json.load(open(sys.argv[1]))
json.dump(dict(zip(i[0],[x.replace('G','R') for x in i[0]])),open(sys.argv[2],'w'))
==================================================================
```

Script 2:

```
==================================================================
import sys,json
i=json.load(open(sys.argv[1]))
json.dump(dict(zip(sorted(i[0]),sorted(i[1]))),open(sys.argv[2],'w'))
==================================================================
```

VIII. Reference

[1] Name mangling, https://en.wikipedia.org/wiki/Name_mangling.

[2] Engineering Change Order (ECO), https://en.wikipedia.org/wiki/Engineering_change_order.

[3] Confromal ECO Designer, https://www.cadence.com/content/cadence-www/global/zh_TW/home/tools/digital-design-and-signoff/functional-eco/conformal-eco-designer.html.

[4] Python JSON encoder and decoder, https://docs.python.org/3/library/json.html.