# Documentation of the work:

## Algorithm or my approach in detail:

-This algorithm calculates the probability of the number of times a condition can be validated based on the size of given variable that is there in the “code” this variable has been passed as an input to the function(compute) as well.

- **Identifying Related Nodes**: The code identifies all nodes connected to the selected "code" variable. It also checks the type of statements they represent using string matching and examines the bit size in the graph.
- **Assessing Relationships**: After identifying these connected nodes (e.g., [41, 61, 81]), the script evaluates their relationships. Instead of immediately calculating probabilities, it checks if any of these nodes is a subset of others in a straightforward manner (one-to-one). If this is the case, it multiplies the probabilities; otherwise, it adds them.
- **Checking Source Code**: The evaluation of relationships includes comparing the "source code" If the source code does not match, it's considered a false match; otherwise, it's deemed true.
- **Parent-Child Relationship**: All these components have a parent switch block. With more time, it's possible to explore this relationship further. This might involve backtracking and examining the situation from the perspective of child nodes.
- **Custom Inputs**: The script is designed to accommodate custom inputs. Users can specify the variable name they want to analyze.

Sample lockvo.sv file contents:

```
if(code == 8'haa) begin -

            state_t <= 2'b01;

        end else begin

            state_t <= 2'b00;

        end

        2'b01   :

            if(code == 8'hbb) begin ==bb

                state_t <= 2'b10

            end else begin -

                state_t <= 2'b01;

            end
```

```
2'b10   :
  if(code == 8'hcc) begin
    state_t <= 2'b11;
  end else begin
    state_t <= 2'b10;
  end
  default : unlocked_t <= 1'b1;
```

**My code does the following:**

-loads the JSON file

-it computes the adjacency graph tog et all the neighbors for each and every node

-it checks for the input parameters that is given as input==TRUE or False(in compute function)

-given the list of Input parameters, it is upto us to select a given variable who's probability needs to be calculated.

My code performs the following tasks:

1. **Loading Data**: It begins by loading data from a JSON file.
2. **Building an Adjacency Graph**: The code constructs a graph to understand the relationships between different parts of the data.
3. **Handling Input Parameters**: It checks if the input parameter is 'True' or 'False' in the 'compute' function.
4. **Variable Selection**: Users can choose a specific variable for probability calculation.

In the 'compute' function:

- **Identifying Related Nodes**: The code finds all nodes connected to the selected "code" variable and checks the type of statements they represent using string matching. It also looks at the bit size in the graph.
- **Assessing Relationships**: After identifying these connected nodes (e.g., [41, 61, 81]), it takes an extra step. Instead of immediately calculating probabilities, it evaluates the relationships between these variables. It checks if any of them is a subset of others in a straightforward manner (one-to-one). If this is the case, it multiplies the probabilities; otherwise, it adds them.
- **Checking Source Code**: The assessment of relationships includes comparing the "source_code." If the source code does not match, it's considered a false match; otherwise, it's deemed true. If this is true there is multiplication of the probability as it is dependent groups else there is addition

- **Parent-Child Relationship**: All these components have a parent switch block. Given more time, there's potential to explore this parent-child relationship further. This might involve backtracking and examining the situation from the perspective of child nodes.
- **Custom Inputs**: The code is designed to accommodate custom inputs by allowing users to specify the variable name they want to analyze.