Cevat Aykan Sevinc sevinc@kth.se
Cemal Arda Kizilkaya cakiz@kth.se

Source code: https://github.com/cevataykans/cfl-model-checking-flow-graph

DD2372 Lab 2 Report

1. Construction of the Flow Graph, DFA, and Their Product CFG

Firstly, in order to represent the flow graph, we created a class called FG and used two hashmaps. The first one is a mapping between function names and their set of edge transitions, where edge transitions are stored in a special data structure holding the beginning and ending nodes of the transitions, namely the NodePair class. The second one is a mapping between the method names and the types of nodes in their flow graphs, where method nodes and their types are stored as a hashmap, where node type is the key and the set of nodes is the value.

In order to represent the DFA, we used the Automaton class that was provided to us in the first lab, with an additional constructor which takes the file name and constructs the automaton according to the .spec file.

Lastly, to represent the product of the DFA and the flow graph, we created a class called CFG, which uses two data structures to represent the product CFG. The first one is a mapping between the variables in the grammar and their sets of productions, where each production is represented as a list whose items are the variables or terminals that can be the yield of the variable. In order to distinguish between the variables and terminals, we have embedded the variables in [ ]'s to make it easier for emptiness tests. The other data structure is a mapping between the variables in the grammars and the set of rules where they appear, which will be used for emptiness tests later on.

2. Calls to Functions Without Definitions in the Flow Graph

Unfortunately, our program requires each function to have a definition on the flow graph. But this can be quickly solved by introducing a fake simulation. If there is a function call that was not specified in the flow graph, our program can check if that function exists in the graph, and immediately return if it does not exist, to simulate a successful execution. In other words, we can create an entry node and a return node for this function call where the entry and return node is connected with an epsilon transition. Consequently, when this function is called, it will try to mimic a successful execution of this program, without knowing what is actually happening inside the function. This may be problematic, as the unspecified function can call other

functions with different behaviours. This solution just saves the algorithm from failing. Therefore, we decided to not to include this functionality and require the user to add each possible requirement for the flow graph.

3. Handling of Underspecified Transitions in DFA

Since a DFA is required to specify a transition for each symbol (in our case, function names) from every state, we may experience some problems when we have some transitions on some methods not being specified. To handle this issue, we can perform a check on the DFA to see if every state has as many transitions as the number of total methods found in the .spec file. If we find out some transitions are missing, we can simply raise an error before proceeding, thus avoiding any problems with the execution.

4. Implementation of Counterexample Search

First, the resulting grammar is checked for emptiness using the algorithm defined in the textbook. If the grammar was found to be non-empty, the “useless” variables in the grammar are deleted and the counterexample search is carried out as follows:

1. First, we push the start variable to the stack.
2. Then, until the stack is empty:
    a. We pop the stack and check if it's a terminal.
    b. If it’s a terminal and it’s not an epsilon, we add it to the counterexample.
    c. If it’s not a terminal, then we try to find an unvisited production from the popped variable and mark these productions as visited, and push the variables in the productions to the stack in the reverse order, so that we preserve the production’s variable ordering.

3. When the stack is empty, we simply print the terminals we’ve obtained by applying the previous step.

We are marking productions as visited because there can be infinitely many counterexamples. By visiting each production once, we are avoiding infinite loops.

5. Meaning of Epsilons

The epsilons in the counterexample obtained denote the internal transitions of the method, rather than calling another function inside. This is somewhat analogous

to having an if statement in the program, and the program takes the branch that stays in the current function, rather than making a call and transitioning to another function.

6. Effects of Input Size on Performance

The change in the input size (nodes in the flow graph, number of methods, number of states in the DFA, etc.) is correlated with the number of productions we obtain for the grammar being constructed (e.g.. the part requiring all state sequences Q^4). Thus, even a slight increase in the input size can have a huge impact on performance.

7. Possible Optimizations

We believe that the design of the current program has been carried out in full detail with the efficiency of emptiness test and counterexample search being the top priority in the design process. The bottleneck of the performance is because of the cross product algorithm. This algorithm explodes productions which are mostly useless. Thus, we can not think of any further improvements in terms of overcoming this bottleneck as we believe the rest of the program has efficient usage of data structures and algorithms.