Automatic Verification of Program Correctness

Anshul Gupta[1], **Anup Deshmukh**[1]

[1]International Institute of Information Technology- Bangalore

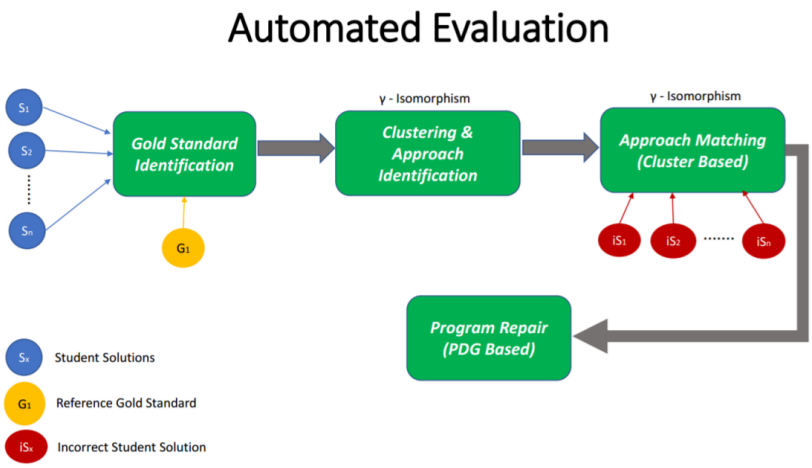
Anshulaalok.gupta@iiitb.org, Deshmukh.Anand@iiitb.org

Abstract. *With the advent of Massive Open Online Courses (MOOCs), it is becoming increasingly important to automate the correction of submitted answers. For obvious reasons Verification of program correctness is costly because human testers will have to manually write test inputs and description of expected test results. There exist many state of art techniques which simply check for complete correctness of a program against a set of test cases. It is equally important to find the distance of how far are these submitted solutions are from the correct solution. Hence, we tried developing an automated approach to this problem using static and dynamic analysis.*

1. Overall goal and our problem

The goal of this project is to develop a tool which identifies the correct submission.

- The tool will categorise the correct submissions into a set of distinct correct 'approaches'
- Compute the 'distance' of the submissions from all the correct submissions, and will apportion marks to submission based on its semantic distance.



Given a gold standard solution and a student solution, our aim was to convert the student solution to the gold standard in the minimal number of changes. Beacuse changes needed will in turn give us the method of grading.

2. Approach 1: Sketch Tool

The Sketch synthesis system allows programmers to write complex routines while leaving fragments of the code unspecified; the contents of these ”holes”in the program is derived automatically by a synthesis engine. In this approach our goal was to identify a set of syntactic changes that can turn a program that is incorrect with respect to a given specification into a correct one. Before going into this approach it is important to understand how distances between two programs are defined as.

- Syntactic distances: Number of edits to the program
  - Boolean Expression distance (tracks if expressions equal or not)
  - Expression Size distance (tracks size of repaired expression)
- Semantic distances: Number of changes to the behaviour of a program with respect to a given set of tests
  - Concrete Execution distance (compares both locations and variables)
  - Value Execution distance (compares only variable values)
  - Location Execution distance (compares only locations)

Programs are written in the Sketch language. These programs are partial programs which in turn have holes. The holes in the language are specified by ’??’. Sketch tries to find values of holes based on the given constraints (output values to be satisfied) and completes the program. If these constraints are not satisfied or cannot be satisfied then Sketch throws error. Basically, specifications constraint the program behavior. Program behaviour can be specified in the following ways.

- A complete formal specification as a formula in some logic (say, first-order). If we want the program P to add 2 to its input, we might write down the logical formula $\forall$ x. $P(x) = x + 2$.
- A set of input/output pairs, as examples of what the program should do. So for the program that subtracts 5 from its input, we might provide a list of pairs like (1, -4), (10, 5), ...
- Demonstrations of how the program should compute its output are similar to input/output pairs, but might also provide some intermediate steps of the computation to show how to transform input to output.
- A reference implementation to compare against.

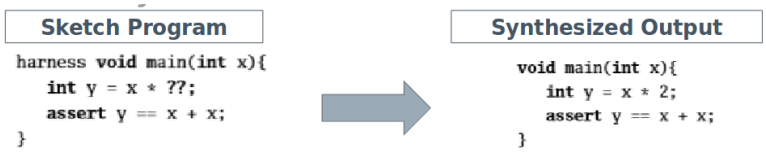


Figura 1. Sketch Synthesizer

In deductive program synthesis technique we deduce an implementation based on the specification and a set of logical axioms while in inductive program synthesis we apply an iterative search technique. Sketch uses Counter Example Guided Inductive Synthesis (CEGIS) algorithm.

We start with some specification of the desired program. A synthesiser produces a candidate program that might satisfy the specification, and then a verifier decides whether that candidate really does satisfy the specification. If so, we're done! If not, the verifier closes the loop by providing some sort of feedback to the synthesizer, which it will use to guide its search for new candidate programs.

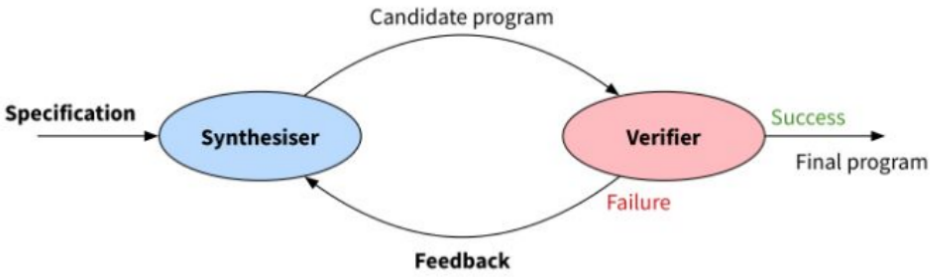


Figura 2. Counter Example Guided Inductive Synthesis

In the algorithm below, Φ_i is the set of all controls which satisfy the specification for the input states $E = \sigma_0$, . . . , σ_{i-1} .$The control \phi_i$ is a candidate selected non-deterministically from Φ_i , and it constitutes the result of the inductive synthesis, as it is guaranteed to be correct for all inputs in E. The state σ_i is an input which exposes an error in the candidate program represented by ϕ_i. The initial control set Φ_0 is initialized to Φ , the set of all controls, while σ_0 is initialized to a random initial state.

Basic Terminologies:

- ϕ_i is the initial set of candidate solutions.
- ϕ_j is the set of constrained solutions which are satisfied.
- Denotation function: C[[P]] [σ_{in}, ϕ_i] [σ_{out} , ϕ_j]

Algorithm 1 (CEGIS Algorithm) .

$\sigma_0 := \sigma_{\text{random}}$
$\Phi_0 = \Phi$
$i := 0$
do
  $i = i + 1$

  def Φ_i s.t. $\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_{i-1}, \Phi_{i-1}) = (\sigma', \Phi_i)$
  if $\Phi_i = \emptyset$ **then return** UNSAT_SKETCH
  def $\phi_i \in \Phi_i$
  } Inductive Synthesis Phase

  def σ_i s.t. $\mathcal{C}[\![P]\!]^{\tau_\emptyset}(\sigma_i, \{\phi_i\}) = (\sigma', \emptyset)$
  } Validation Phase

while $\sigma_i \neq$ **null**
return $PE(P, \phi_i)$

Figura 3. Formalization of Algorithm

3. Approach 2: PDG (Program Dependence Graph)

Given the PDGs of a gold standard solution and a student solution, our aim was to convert the student solution PDG to the gold standard PDG in the minimal number of changes. Our proposed approaches are discussed below,

Approach 2.1

- Traverse the tree using BFS/ DFS
- At every node, compare node type. Change accordingly
- At every node, check children. “Add” or “delete” edges accordingly
- Remove nodes with no incoming edges.

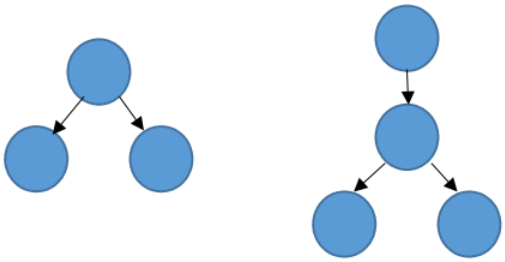


Figura 4. Problem with the approach 2.1

Approach 2.2

- Traverse the tree using BFS/ DFS
- At every node, compare node type. “Add” accordingly.
- If added, keep pointer of program node fixed and update the pointer of the GS node.
- Else, check children. “Add” or “delete” edges accordingly.

Figura 5. Problem with the approach 2.2

Approach 2.3

Direct comparison on the basis of node ids.

Very fragile code that is highly dependent on assignment of node ids.

Figura 6. Problem with the approach 2.3

4. Approach 3: PDG Subproblem

In this approach our aim was to find the largest common subgraph between the PDG of the Gold standard solution and the PDG of the student solution.

4.1. Ullman's Algo for Subgraph Isomorphism

Basic idea: There is some formal similarity between the problems of finding whether two graphs are related and finding whether two patterns conserve spatial relationships. For example,, Chemical compund is subcompound of another can be again viewwed as a similar problem. Original propesed work by ulllman is discussed below,

Basic terminologies:

- G1 = (V1, E1): number of points and lines are p1 and q1
- G2 = (V2, E2): number of points and lines are p2 and q2
- The adjacency matrices of G1 and G2 are A = $[a_{ij}]$ and B = $[b_{ij}]$

Defined Matrices:

- Matrix $M^{'}$: (p1 * p2)
  1, If j^{th} point in G2 corresponds to the i^{th}point in G1, in the graph isomorphism.
  0, otherwise
- Matrix M^{0} : (p1 * p2)
  1, If the degree of the j^{th} point of G2 is greater than or equal to the degree of the i^{th} point of G1,
  0, otherwise.

Matrices $M^{'}$ are generated by systematically changing to 0 all but one of the 1's in each of the rows of M^{0} , subject to the definitory condition that no column of a matrix$M^{'}$ may contain more than one '1'. For each such matrix M' the algorithm tests for Isomorphism by applying following,

$$C = [c_{ij}\ ] = M^{'}(M^{'}B)^{T}\ ,$$

where T denotes transposition. If it is true that,

$$(\ V_iV_j\ )\ (a_{ij} = 1) \Rightarrow (c_{ij} = 1),$$

If the above condition holds then we can conclude that the Matrix M' was assigned correctly.

4.2. A Backtrack Procedure for Isomorphism of Directed Graphs

In the proposed approach authors have set up a reference K-formula of one of the digraphs, and then they found all K-formulas representing the other that have the same pattern. Each such correspondence defines an isomorphism. This will be done in stages, by comparing K-formulas of substructures. The final task is then to read off the two node correspondance.

Definitions:

- A directed graph or digraph is the ordered pair D = (A, P), where A is a set and P is a relation in A. Members of A are called nodes and members of P are arcs of the digraphs
- Arc (a_i , a_j) is said to originate from node a_i and to terminate at node a_j
- If D = (A, P) is a digraph, and B is a subset of A, then $D^{'}$ = B, (B $\times$ B) $\cap$ P is a subdigraph of D. If Q is a subset of P, then $D^{''}$ = A, Q is a partial digraph of D.

K-formulas

The proposed algorithm is based on the linear notation for digraphs. The given notation of K-formulas is: An arc (a, b) is represented by *ab where

'*' is called as the K-operator
'*ab' is called as a K- formula.

A K-formula that represents n arcs originating from a given node in a digraph consists of n K-operators, followed by the name of the given node, followed in turn by names of the n nodes at which the arcs terminate. For example,

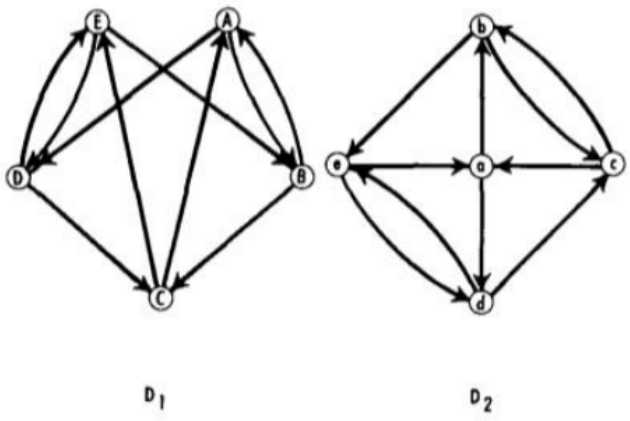


Figura 7. Example directed graph

The K- formulas for each node for graph D1 can be given by,

node A: **ABD
node B: **BAC
node C: **CAE
node D: **DCE
node E: **EBD

K-formulas may be combined. For an instance in the above example We replace the B in **ABD by **BAC to get **A**BACD, which represents all arcs originating from nodes A and B. And hence any digraph can be represented by K-formula. K-formula that represents the entire digraph D1:

AB**C**E**DCEBAAD

Let D = (A, P) be a digraph

1) For every isolated node a ε A write the K-formula a.

*2) For every node b from which originate arcs (b, t_1), (b, t_2), ... ,(b, t_k) write the K-formula **...*bt_1t_2 .. t_k, where k K-operators precede the b.*

3) Combine the K-formulas according to the following substiution rule If there exmts a K-formula of a node and there exmts another K-formula in which the name of the node appears, substitute the K-formula of the node for thin name Apply the substitution rule until it can no longer be applied

4 (Check step.) Denote the K-formulas produced in Step 3 by f_1, f_2, ... , f_n and the leading node of any f_i by a_i If some f_i contains as a subformula a K-formula of node b in which a_i occurs, and the b occurs in one of f_1, ... , f_{i-1} ,f_{i+1}, .. , f_n . extract the K-formula of b from f_i . inserting b in its place, substitute what now remains of f_i into this formula, and return to Step 3. Otherwise stop.

Figura 8. Generating K-formulas

Isomorphism of Digraphs: In the proposed approach authors have set up a reference K-formula of one of the digraphs, and then they found all K-formulas representing the other that have the same pattern. Each such correspondence defines an isomorphism. This will be done in stages, by comparing K-formulas of substructures.

1. Let the reference formula be:
*A**B**C**E**DCEBAAD (K-formula of D1) .. (1)
K- formula of all the nodes of digraph D2 are:
*abd, **bce, **cab, **dce, **ead .. (2)

2. Then we will look for K-formulas that will match subformula **DCE from
*A**B**C**E**DCEBAAD .. (3)

3. Then we try to produce K-formulas that have the same pattern as subformula
EDCEB from
*A**B**C**E**DCEBAAD .. (4)
K- formulas that can be generated from (2) that has the same pattern as (3) are,
cbeca, **b**cabe, **e**dcea and **d**eadc

The same iterative provedure will result in all the isomorphic digraphs to (1). The final task is to read off the two node correspondance. Since all permutations of the node names are considered in the matching process, the procedure finds all Isomorphisms.

5. Our Naive Algorithm

The algorithms discussd above while elegent, provide optimizations that are probably useful only in the case of very large graphs. Hence we propose our own naive algorithm:

```
MaxSubgraphVertex(v,V)
If v ∈ V,
        For every v' ∈ Adj(v)
                Return ( v ∪ MaxSubgraph(v',V) )
Else
        Return ϕ

MaxSubgraph(V_gs,V)
For every v ∈ V_gs
        Return max ( length ( MaxSubgraphVertex(v,V) ) )
```

MaxsubgraphVertex finds the node that exactly matches the parameter 'v' in the set of vertices 'V'. We then apply the same algorithm to each of its adjacent nodes and return the largest set of vertices which exactly match between the subgraph associated with v, and V.

MaxSubgraph applies MaxSubgraphVertex to each of the nodes of the first graph 'V_gs' and returns the largest set of matching vertices.

6. Libraries used

java.util.ArrayList - Datastructure used for storing nodes and edges
import java.nio.file.Files;
import java.nio.file.Paths;
import java.nio.charset.Charset;
import java.io.IOException;

7. References

- [1] "An Algorithm for Subgraph Isomorphism", J.R. Ullman, National Physical Laboratory, Teddington, Middlesex, England, Journal of the Association for Computing Machinery, VoI 23, No 1, January 1976

- [2] "A Backtrack Procedure for Isomorphism of Directed Graphs", A. T. BERZTISS, University of Pittsburgh, Pittsburgh, Pensylvania, Journal of the Association for Computing Machinery, Vol. 20, No 3, July 1973

- [3] "Program Synthesis by Sketching", Armando Solar-Lezama, Texas AM University 2003
- [4] "Frama-C". frama-c.com. Retrieved 2016-11-05.