ECE750T28 Project: Synthesizing Boolean Expressions Based On Several Constraints Using Counterexample Guided Inductive Synthesis Concepts

Mohammad Robati Shirzad (Student ID: 20963513) (mrobatis@uwaterloo.ca)
Rahul Punchhi (Student ID: 20569992) (rpunchhi@uwaterloo.ca)

***Abstract*— A Program Synthesis tool automates the writing of computer programs based on a set of specifications and constraints provided to the tool. In this paper, we outline the development of a Boolean expression synthesis tool based upon the JS-Buxter tool developed by author Mohammad Robati Shirzad. This automated program repair tool was developed for resolving JavaScript bugs. It included support for the IF-CC bug fix pattern, which focused on bugs related to if-condition statements. The JS-Buxter tool used a "Brute-Force" approach, but it was clear that there could be a way to resolve such bugs more efficiently. Since if-condition statements are activated based on a Boolean result of some given variable(s), a Boolean expression synthesis tool would be helpful in improving performance for the IF-CC bug fix pattern. This project is focused on our development of such a Boolean expression synthesis tool. We provide background information for the software and theoretical concepts we used in development of our Boolean Expression Synthesis tool. These concepts are (in part) based upon the Counterexample Guided Inductive Synthesis (CEGIS) approach. We developed two approaches for program synthesis: "Feedback Only", and "Feedback + Initial Probing". We sought to establish that these approaches achieved Boolean program synthesis much faster than a "Brute-Force" Approach. All three approaches would take a set of variables and a set of constraints as input. We collected data for all three approaches, for different levels of expression depth. We analyzed the results of this data, and determined that these feedback-based approaches achieved Boolean expression synthesis significantly faster (more than 10 times faster for large inputs) than the "Brute-Force" approach.**

I. INTRODUCTION

Program Synthesis is the process of automatically writing a computer program based on the syntax of a specific programming language which satisfies certain predefined specifications. The problem of program synthesis is considered to be a long sought-after achievement of Computer Science research (p. 3) [13]. There have been many approaches suggested by researchers for synthesizing a program. In the late 1960's and early 1970's, researchers tried to leverage the power of automatic theorem provers for automatic program writing (p. 4) [13] [1] [2] [3]. Another approach was described by [13], in which the provided high-level formal specification would be reduced to a low-level program underlying a programming language after several iterations of transformation (p. 4) [13] [4]. A more recent approach for program synthesis is Syntax-Guided Program Synthesis, pioneered by the Sketching [7] approach (p. 4) [13]. In this approach, the user provides a skeleton of the desired program (with holes in it) alongside the specification to the synthesis tool. Defining a skeleton for the desired program can drastically reduce the search space, resulting in a more efficient synthesis tool. Solar-Lezama also introduced Counterexample Guided Inductive Synthesis (CEGIS) as a strategy to efficiently fill the gaps inside a template program (p. 7) [26]. This was done by providing a synthesizer with feedback regarding the root of incorrectness of a previously synthesized program.

In our tool, instead of focusing on synthesizing general purpose functions, we have only focused on automatically generating Boolean expressions. Boolean expressions can be found at the heart of any computer program, irrespective of programming language. These expressions repeatedly appear in control flow statements as well as different types of program loops. Therefore, given that Boolean expressions are so crucial to common elements of any program, and that program synthesis itself can be a complex problem, the problem of Boolean expression synthesis is non-trivial.

In this paper, we address the problem of synthesizing Boolean expressions based on a given set of input variables and constraints. We sought to develop a tool that could synthesize such expressions faster than a naive "Brute-Force" approach.

An efficient Boolean expression synthesizer can en-

hance the performance of automated program repair tools. Automated program repair tools try to locate bug locations in the source code and then fix them, possibly without human intervention, in a manner such that the modified program meets the desired specifications (p. 57) [19]. A powerful automated program repair system has many useful applications (p. 63-64) [19] (p. 16) [13] [18] [11] [8] [16] [9] and can significantly simplify the process of debugging, resulting in a reduction in software development cost(s).

Pan et al. analyzed seven Java open source projects, and calculated the frequency of occurrence of 27 predefined bug fix patterns (p. 312) [6]. Their results showed that IF-CC (change in if-condition) accounts for 5.6–18.6% of the total bugs found (p. 312) [6]. This supports the notion that errors in if-conditions are one of the most common programming mistakes.

Since if-conditions are essentially Boolean expressions, an efficient Boolean expression synthesizer could enhance the performance of a program-repair algorithm. This could ultimately optimize the function of a program repair system.

The problem of Boolean Synthesis involves two inputs. The first input is a set of variables. In our implementation, these variables can be integers or Boolean values. The second input is a set of constraints that must be applied to these input variables. Based on these inputs, the resulting output must be a synthesized Boolean expression that satisfies all the given constraints (if possible). Our approach to Boolean Synthesis uses concepts from CounterExample Guided Inductive Synthesis (CEGIS) [17] [21].

In overview, we developed two solutions to Boolean expression synthesis: "Feedback-driven" and "Feedback + Initial Probing". These solutions used concepts from CEGIS to synthesize Boolean expressions faster than a naive "Brute-Force" approach.

The programming language and software tools used in this project are discussed in Section II. Section III describes the project problem, the concepts of program synthesis, CEGIS, and previously completed work. Section IV summarizes the problem-solving approach employed by our project solution implementation. It also describes the algorithm, experimental benchmarks, and testing setup. Section V summarizes the results of our implementation based on our defined benchmarks and testing parameters. We provide analysis of our data in this section. Section VI summarizes the cited papers that we used as part of our project implementation and research. In Section VII, we summarize the implications of our result. This section also discusses the impact and novelty of our project. Lastly, in Section VIII, we discuss the applications of our project, and ways in which it could be improved.

II. Methods and Materials

We selected Python as the programming language for development. Python was used in the development of JS-Buxter by Mohammad Robati Shirzad. Our Boolean Synthesis implementation was partially based on this prior work project, so continuing to use Python reduced development overhead.

The Z3 solver was used as part of our implementation. The Z3 SMT solver is an open source theorem prover [27]. Z3 is described as "...an efficient SMT solver with specialized algorithms for solving background theories" [22].

Our implementation also uses the itertools module, specifically its combinations function [23]. The function takes as input an iterable value and an integer r [23]. The output of the combination function is the sub-sequences of the iterable value, each of which is of length r [23]. This was used in our "Brute-Force" synthesis approach.

Our implementation uses the yacc module from the ply module as part of our parser implementation. The yacc module is included in order to "...recognize language syntax that has been specified in the form of a context free grammar" [25]. This was used as part of building our parser.

III. Background

A. Boolean Expression Synthesis

Program synthesis is a process of automatically generating a program that will satisfy a given set of specified requirements (p. 3) [13]. Boolean Expression Synthesis is a simpler subset of this synthesis problem.

The focus of our project is to synthesize Boolean expressions given the following inputs:

1) A finite set of variables (integer and/or Boolean)
2) A finite set of constraints

The Boolean expression that is outputted from these given arguments must satisfy the given constraints using the explicitly defined variables. Fig. 1 gives an example of how inputs can be defined.

An acceptable result to these sample inputs would be the following Boolean expression:

$$a == b \;||\; b == c \;||\; a == c$$

```
"variables": ["a", "b", "c"],
"constraints": [
    {"mappings": [2, 1, 3], "evaluation": false},
    {"mappings": [1, 1, 2], "evaluation": true},
    {"mappings": [1, 2, 1], "evaluation": true},
]
```

Fig. 1. A sample of input [20]

This example was originally included as part of our project proposal submission. Our goal was to develop an approach to Boolean expression synthesis that would be faster than a "Brute-Force" approach. In order to accomplish this, we used concepts from the Counterexample Guided Inductive Synthesis (CEGIS) approach.

B. CEGIS

1) The CEGIS approach: We use concepts from the CEGIS algorithm as part of our implementation of Boolean expression synthesis. CEGIS is defined as the Counterexample-Guided Inductive Synthesis approach for program synthesis (p. 272) [17]. CEGIS has two phases: synthesis and verification (p. 272) [17].

The synthesis phase uses a program specification σ to generate candidate program $P*$ (p. 272) [17]. This candidate program $P*$ satisfies $\sigma(P*, x)$ for the subset of all inputs, denoted as x_{inputs} (p. 272) [17]. The candidate $P*$ will then be sent to the verification phase.

The verification phase receives this candidate program $P*$ to ensure that it satisfies $\sigma(P*, x)$ for all possible inputs, not just a given subset (p. 272) [17]. To achieve this, this phase verifies that $\neg\sigma(P*, x)$ is UNSAT (p. 273) [17]. If it holds that $\neg\sigma(P*, x)$ is unsatisfiable, then $\forall x.\neg\sigma(P*, x)$ holds, and the candidate solution $P*$ satisfies $\sigma(P*, x)$ for all possible inputs and CEGIS terminates (p. 273) [17]. If, however, there is a possible satisfiable counterexample c, it is returned back to the synthesis phase as part of the input, and this process repeats (p. 273) [17]. Fig. 2 illustrates the CEGIS process.

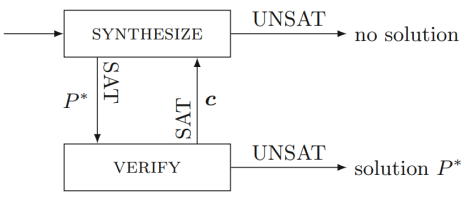

Fig. 2. CEGIS block diagram (p. 273) [17]

The concept of using a feedback loop during the CEGIS process would be a great improvement over a "Brute-Force" synthesis approach. Using feedback could help reduce the number of possible candidate solutions to be checked in a synthesis process. When considering the metric of performance time for a given input, a synthesis algorithm that uses verification feedback could significantly reduce synthesis run-time in comparison to "Brute-Force" algorithms.

C. Prior Work

This Boolean synthesis project was based upon prior work by Mohammad Robati Shirzad. This prior work was the "JS-Buxter" tool, written in Python, which would repair buggy JavaScript files [24]. In particular, the IF-CC ("If Change Condition") bug-fix pattern of JS-Buxter [24] formed a basis for this Boolean Synthesis project. This bug-fix pattern applied to instances of incorrect if-conditions within an input JavaScript file.

IV. Solution Description and Implementation

A. Our Approach

A naive approach to look for the appropriate Boolean expression can be broken down into two steps. First, one synthesizes all possible expressions. Second, one iterates through all expressions until finding the expression which satisfies all constraints. A Boolean expression is essentially a string of symbols which have drawn from the set $U = V \cup C \cup L$. V is the set of all defined variables appearing in our constraints list. C and L are the sets of comparison and logical operators that we are allowed to use in our synthesis process, respectively. Since our algorithm for generating feedback to cut branches of the search tree relies on the semantics of the logical operators that we can use, our tool only accounts for $L \subseteq \{\text{and, or}\}$ where $L \neq \phi$. The "and" and "or" logical operators are expressive enough to build all other possible semantics.

Parentheses are symbols that are used in Boolean expressions to override the predefined order of evaluation of the logical operators. However, we have not incorporated the ability of using parentheses in the synthesis process. We have assumed fixed positions for parenthesization, which is to encapsulate each two rightmost expressions with parentheses, and that results in a right-associative Boolean expression. Assuming the right-associative parenthesization is crucial for our feedback-generation algorithm. It should be noted that if we consider only right-associative Boolean expressions with fixed positions for variables, we will miss the

opportunity of checking many Boolean expressions. To solve this issue, we will synthesize Boolean expressions through several iterations. In each iteration, all synthesized expressions are in a right-associative structure, but the order of chosen variables differs from one iteration to another. Our final synthesized Boolean expression must be based on the syntax of the SMT-LIB2 language [12], as we want to verify it using the Z3 solver. An example of a valid Boolean expression would be:

$$((< a\, b) \text{ AND } ((\text{distinct } a\, c) \text{ OR } (\leq a\, c)))$$

The terms a,b, and c are integer variables.

B. Encoding

Throughout the search, we keep track of two lists:

- choices
- id

The **choices** list indicates the order of the current chosen variables via their corresponding indices. The **id** list has the order of comparison and logical operators encoded in it via their corresponding indices.

Since we only synthesize right-associative expressions, we do not need to worry about encoding parentheses in our settings. To give an example, given following sets:

$$V = \{a, b, c\}$$
$$C = \{<, >, =, \text{distinct}, \leq, \geq\}$$
$$L = \{\text{and, or}\}$$

If the current state of the choices and id lists are [0,1,0,2,1,2] and [0,0,3,1,4], it would indicate that our currently synthesized Boolean expression is:

$$((a < b) \text{ AND } ((\text{ distinct } a\, c) \text{ OR } (\leq a\, c)))$$

Considering only right-associative expressions helps us to elegantly represent all possible expressions via a tree.

C. Our Synthesis Tool

In general, our tool is comprised of three different components: Synthesis, Verification, and Feedback Generation.

1) Synthesis: In the synthesis module, we maintain our position in the synthesis tree via keeping track of **choices** and **id** lists through all iterations. Based on the current state of the **choices** and **id** lists, we synthesize the next Boolean expression and input it to the verification component.

2) Verification: In this component we check if the current synthesized Boolean expression satisfies all the constraints. If it does, then we will output the current expression as our final answer. We use an instance of the Solver class in the Z3 library to carry out this verification step. If the solver outputs UNSAT, it means that we have to signal the feedback generation component to inspect the root of the problem in the expression and provide useful information for further synthesis.

3) Feedback Generation: The CEGIS strategy helps the synthesizer module to synthesize better programs by conducting a dialogue between the synthesizer and verification modules. This dialogue works as follows: Each time the synthesizer module creates a program and sends it to the verification module, the verification modules searches for a counterexample in which the program fails to meet its constraints. Abate et al. (p. 270) [17] added more power to this strategy by incorporating an extra theory solver. This theory solver allows the synthesizer to ask more general questions, and therefore synthesize new programs more efficiently. Although our synthesis tool cannot entirely adhere to the CEGIS strategy, we have extracted some ideas from it to design our feedback generator module.

Here we define two invariants about the right-associative Boolean expressions which yields an algorithm for generating feedback for the synthesis module:

Invariant 1:

If we have synthesized a Boolean expression which fails on a constraint that must be evaluated to true, then there are two potential roots of the expression's problem. The issue could lie in the leftmost false sub-expression which has been joined to its right-hand side. Alternatively, the root may be the last sub-expression that appears in the whole Boolean expression.

Similarly, we can define a complement version of the invariant above:

Invariant 2

If we have synthesized a Boolean expression which fails on a constraint that must be evaluated to false, then there are two potential roots of the expression's problem. The root of the problem could be the leftmost true sub-expression which has been disjuncted to its right-hand side. Alternatively, the root of the problem may be the last sub-expression that appears in the whole Boolean expression.

Our feedback generation component is comprised of two algorithms: INSPECT and DEDUCT. These two algorithms work together based on invariants 1 and 2

in order to inform the synthesis component of which branches can be cut off and where it can jump forward to, ignoring all the leaves in-between. These algorithms are described in the subsection titled *Code Discussion*.

4) Initial Probing: Up until now, whenever we want to search for a desired Boolean expression, we start off from the leftmost leaf in the synthesis tree. This is done for both the plain brute-force search or feedback-driven approach. As shown in Fig. 3, perhaps the area possessing the most answers is in a distance from the leftmost node. Consequently, we have developed an algorithm to detect an area of the tree which is more likely to contain answers.

The core idea of our algorithm is to perform a local optimization over the sub-expressions that can be synthesized based on set of comparisons C, set of logical operators L, and expression depth D. This approach will select each logical comparison operator that satisfies most constraints (if used in the expression).

We take the sample synthesis tree in Fig. 3, and apply the algorithms discussed. We assume the following:

- $V = a, b, c$
- $C = <, >$
- $L =$ and, or
- choices $= [0, 1, 0, 2]$
- $D = 1$

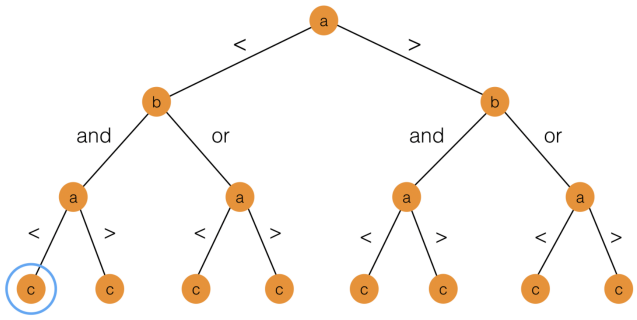

Fig. 3. Sample Synthesis Tree

Looking at Fig. 3, in a naive brute-force search, we would start off from the leftmost leaf $((a < b)$ and $(a < c))$ and check all the possible expressions until we reach an expression that satisfies our constraints. Specifically, in the example of Fig. 3, we have to iterate through six expression until we reach a success node $((a > b)$ and $(a > c))$. Fig. 4 illustrates this approach.

Unlike the brute-force search, the feedback-driven approach will provide us with information about which leaves we can ignore along our search by cutting tree

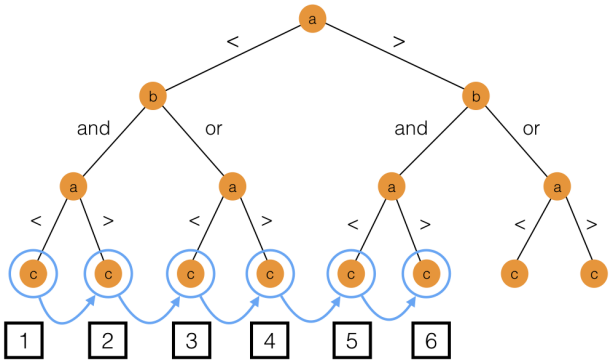

Fig. 4. Naive Brute-Force applied to Sample Synthesis Tree

branches. Getting back to our example in Fig. 3, after verifying the leftmost leaf in the tree, the feedback generation component will tell us that we can ignore other leaves in the subtree $((a < b)$ and $\square)$ (based on the first invariant). Therefore, we jump forward to the leaf representing the expression $((a < b)$ or $(a < c))$. Similarly, after verifying this leaf, the feedback generation component will again tell us that we can ignore other leaves in the subtree $((a < b)$ or $\square)$ (based on the second invariant). Therefore, we jump to the next available leaf, which is $((a > b)$ and $(a < c))$. Overall, by the time we reach the success node, we will have only checked four nodes. This difference between the number of checked leaves in a brute-force search and feedback-driven approach would be more significant for a larger problem size. This feedback-driven approach is shown in detail in Fig. 5.

Now let us incorporate the initial probing algorithm to our search. At the first level, our algorithm tries to select the first expression between $((a < b)$ and $\square)$, $((a < b)$ or $\square)$, $((a > b)$ and $\square)$, and $((a > b)$ or $\square)$. The purpose of this selection is to choose the highest-scoring expression. Based on the constraints, the scores would be 0,0,3,3, respectively. Therefore the first highest score belongs to the expression $((a > b)$ and $\square)$. If we keep on applying the methods until we reach the last level, we would reach the expression $((a > b)$ and $(a < c))$, which satisfies our constraints. In other words, in this specific sample synthesis tree, if we use the initial probing algorithm to select the first leaf to start off from, we would only check one node, which is the starting node itself. The Feedback and Initial Probing approach is illustrated in Fig. 6.

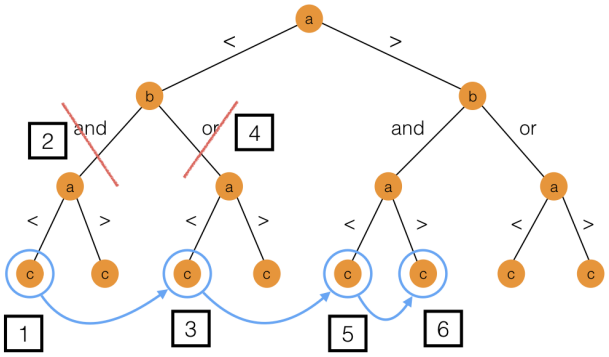

Fig. 5. Feedback-driven Approach applied to Sample Synthesis Tree

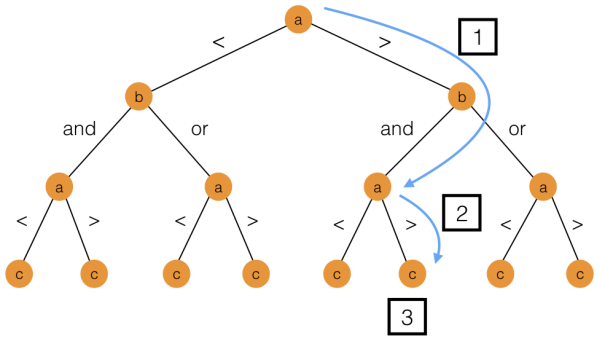

Fig. 6. Feedback and Initial Probing Approach applied to Sample Synthesis Tree

5) Our Search Algorithm vs. CDCL: In this subsection, we mainly derive our understanding of CDCL from ECE750T28 lecture notes and [10]. CDCL [5] is an efficient algorithm for solving SAT problem. Although CDCL algorithm and our algorithm are solving different problems, there are certain similarities between them. In general, the CDCL algorithm iteratively assigns Boolean values to the Boolean variables present in a given SAT expression. If one assignment produces an UNSAT clause in the expression, a conflict analysis function will be invoked to inspect the cause of the unsatisfiabilty. As a result, the conflict analysis function will append a learn clause to the clause list. This decreases the search space (i.e. by ignoring certain assignments in the rest of the search). We believe that our feedback generation component resembles the conflict analysis function as it provides feedback (similar to learn clause in CDCL) for the search module to shrink the search space.

Moreover, CDCL uses a branching subroutine to select the best variable for next assignment. This subroutine is somehow similar to the initial probing algorithm that we use to find the best starting point to begin our search.

CDCL differs from our tool's algorithm approach in that it assigns variables in an incremental fashion. In our tool, at any stage, we have a complete expression to verify or draw information from. There are two reasons for this approach in our tool. Firstly, it gives us the possibility to implement our approach on top of a naive brute-force search. Secondly, our approach yields a perfect opportunity for parallelization. This is discussed further in Section VIII.

6) Code Discussion: Two functions are used as part of our synthesizer: DEDUCT (see Algorithm 1) and INSPECT (see Algorithm 2). These functions are located within the `synthesizer.py` file of our implementation source code.

The DEDUCT function recursively seeks the last sub-expression within a given search-tree. It leverages the INSPECT function to achieve this purpose. The INSPECT function leverages a solver to determine the satisfiability of a given input expression.

The functions DEDUCT and INSPECT work together to set the correct value for the global variable *cutLevel*. The *cutLevel* variable will then be used in the Synthesis module. We only call the function DEDUCT when the current expression cannot satisfy a given constraint. Since *expr* is right-associative, in the DEDUCT function (Line (2)), we check if the root of the problem is the left-hand side of the *expr*. If so, then we jump to Line (6) and *cutLevel* is set to 1. If not, then we must advance to the right-hand side and recursively call DEDUCT on the right-hand side. Note that before the recursive call, we must check if we have reached the last sub-expression (DEDUCT - Line (4)). If this condition holds true, then we return *True*, which means that no feedback can be generated for the current expression.

In function INSPECT, we check the situation of the input expression (the left hand side) against the unsatisfied constraint. As mentioned before, there are two cases that can produce unsatisfied constraints, first (Line (5)) is when *expr* is not consistent with the constraint and the logical operator is "And", and second (Line (8)) is when *expr* is consistent with the constraint and the logical operator is "Or".

The INITIALPROBING function (see Algorithm 3) is located within the `bruteforce.py` file of our implementation source code. This function seeks to find a specific area of a given synthesis tree that could contain a satisfactory expression. This is meant to be an improvement over the conventional approach of always starting searches from the leftmost leaf. The *gen(op, exp1, exp2)* function will build a prefix Boolean string which is recognizable in SMT-LIB2 language.

Algorithm 1 DEDUCT Function
Input: Expression (expr)
Output: Boolean Value

```
1: function DEDUCT(expr)
2:     if INSPECT(expr.LHS, expr.logicalOperator) then
3:                         ▷ Below, we have reached the last sub expression
4:         if expr.RHS does not contain logical operators then
5:             return True
6:         return DEDUCT(expr.RHS)
7:     ▷ In above line, we cannot cut the tree, so we advance to further sub-expressions
8:     return False
```

Algorithm 2 INSPECT Function
Inputs: Expression (expr), LogicalOperator (logOp)
Output: Boolean Value

```
1: function INSPECT(expr, logOp)
2:     solver ← SOLVER(expr, constraints)
3:     result = solver.check()
4:     cutLevel ← cutLevel + 1                          ▷ global variable
5:     if logOp = "And" and result = "UNSAT" then
6:                                                      ▷ invariant 1
7:         return False
8:     if logOp = "Or" and result = "SAT" then
9:                                                      ▷ invariant 2
10:        return False
11:    return True
```

Algorithm 3 INITIALPROBING Function
Inputs: choices, LogicalOperator
Output: id

```
1: function INITIALPROBING(choices)
2:     id ← 0
3:     for choice in choices do
4:         bestScore ← 0
5:         for op in C do
6:             expr ← gen(and, gen(op, choice[0], choice[1]), true)
7:             score ← getScore(expr)
8:             if score > bestScore then
9:                 bestScore ← score
10:                                                  ▷ encode expr in id
11:            expr ← gen(or, gen(op, choice[0], choice[1]), false)
12:            score ← getScore(expr)
13:            if score > bestScore then
14:                bestScore ← score
15:                                                  ▷ encode expr in id
16:    return id
```

V. EXPERIMENTAL EVALUATION AND ANALYSIS

A. Evaluation Approach

The input to our tool must follow the grammar described in Fig. 3, which has some similarity to the SMT-LIB2 language. In general, the input is comprised of two parts: *variable declaration* and *constraints list*.

In the variable declaration, variables can be of type Integer or Boolean. Each constraint in the constraints list is partitioned into two sections. The first section is a set of assertions which describe the boundaries we put on our variables. In the second section, we can determine how we want our synthesized expression to be evaluated when given this constraint. Fig. 7 shows an example of a variable declaration and assignment used in evaluation.

```
(declare-fun i2 () Int)
(declare-fun i1 () Int)
(declare-fun i4 () Int)
(declare-fun b2 () Bool)
(declare-fun b1 () Bool)
;

(assert (< i2 1))
(assert (<= i1 3))
(assert (<= i4 7))
(assert (= b2 true))
(assert (= b1 true))
(assert (= synth false));
```

Fig. 7. Example of Evaluation Test Data

We developed a Python script that could randomly generate 30 datasets (formatted as text files). Each dataset consisted of a declaration of Integer and/or Boolean variables, followed by multiple assignments. As part of generating these 30 datasets, our script would define an expression "depth" value. In our data collection approach, we selected depth values of 2, 3, and 4.

B. Experimental Setup

We performed our data collection on a 2016 MacBook Pro laptop with a 2.9GHz dual-core Intel Core i5 processor. This machine has a turbo boost clock speed of up to 3.3GHz, with 4MB shared L3 cache. This machine also possesses 8GB of 2133MHz LPDDR3 on-board memory. The laptop's OS is MacOS Mojave 10.14.6.

We compared our approaches of "Feedback Only" and "Feedback + Initial Probing" (also written as "Feedback and Initial Probing") against a baseline approach of "Brute Force". This was done to establish

whether our Feedback-based approaches indeed could outperform the most simplistic solution approach. We used the metric of synthesis time, measured in seconds.

C. Results and Analysis

After testing our tool against our benchmark, we realized that the "Feedback only" approach, regardless of the chosen depth for synthesis, will always give us a better execution time than the "Brute Force" approach. Moreover, we observed that the "Feedback + Initial Probing" approach shows a significantly better performance over more complicated syntheses in comparison to the other two approaches. For the most complex inputs, across depth values, "Feedback + Initial Probing" would always perform synthesis at least 10 seconds faster than the other approaches.

However, in smaller problem sizes, the "Feedback only" approach for some instances gives better results than the "Feedback + Initial Probing" approach. There are two reasons for that. First is that the initial probing approach has an overhead of efficiency *O(depth)* to find the best starting leaf. Additionally, in our initial probing approach, there is no mechanism to prevent falling into the local optimum and missing the best answer. Therefore, practically, our greedy algorithm for finding the best starting point can give us a weak leaf that can impose a longer search time. Lastly, our results showed that when the constraints are spanning over a narrower domain (i.e. assigning a value to variables instead of defining a valid interval for them), "Feedback + Initial Probing" is by far the superior approach.

Fig. 8 and Fig. 9 depict the results for a depth-value of 2. Fig. 8 shows that the Brute Force approach of program synthesis has the highest maximum for synthesis time. It also has the highest median time. The Feedback Only approach has the lowest median time. It also has the largest disparity between minimum and maximum. The approach of Feedback and Initial Probing has a longer median time than Feedback Only, but the maximum time is noticeably smaller than Feedback Only. Fig. 9 shows that as compared to Brute Force, the other Feedback-type approaches perform synthesis considerably faster. This is especially noticeable where the Brute Force approach takes the longest. It should also be noted that there is a point where the Feedback Only approach consistently takes longer than the Feedback and Initial Probing approach.

Fig. 10 and Fig. 11 depict the results for a depth-value of 3. Fig. 10 shows that the Brute Force approach has the largest median synthesis time. The Feedback

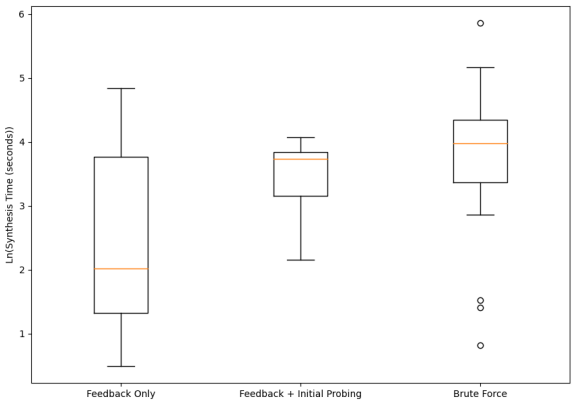

Fig. 8. Box Plot (Depth 2)

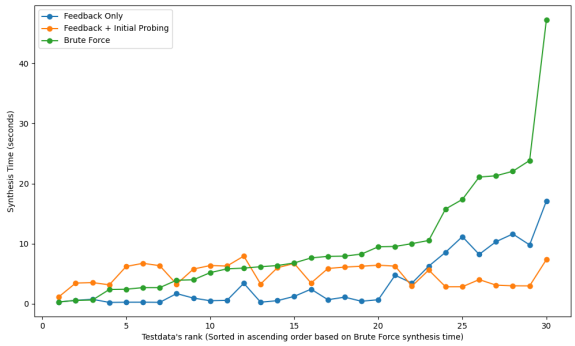

Fig. 9. Line Plot (Depth 2)

Only approach has the lowest median synthesis time. For the depth-value of 3, the Feedback and Initial Probing has a much smaller difference between its minimum and maximum, when compared to the other two approaches. Fig. 11 supports this observation: The Feedback and Initial Probing data line (orange) fluctuates much less than the data line (blue) for the Feedback Only approach.

Fig. 12 and Fig. 13 depict the results for a depth-value of 4. Fig. 12 shows similar results as previous box plots. Brute Force has the largest median synthesis time, and Feedback Only has the smallest median synthesis time. At the depth-value of 4, the Feedback Only approach has a considerably larger interquartile range in comparison to the other approaches. Also, although the Feedback and Initial Probing approach has the smallest difference between maximum and minimum synthesis times, it also appears to have more outlier data.

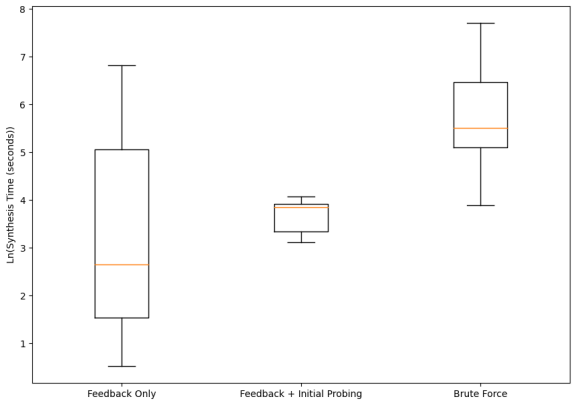

Fig. 10. Box Plot (Depth 3)

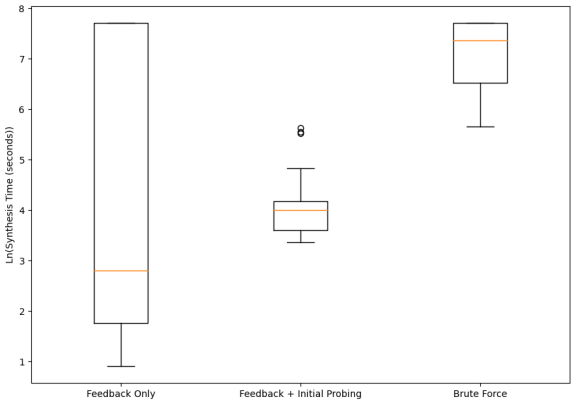

Fig. 12. Box Plot (Depth 4)

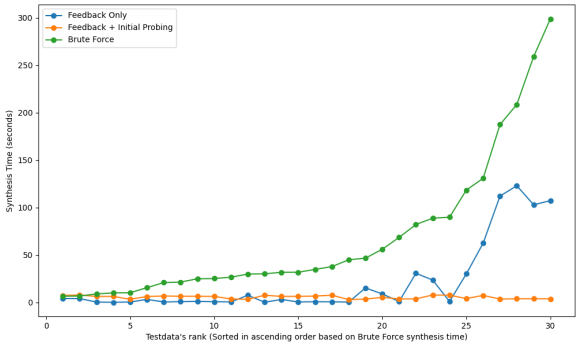

Fig. 11. Line Plot (Depth 3)

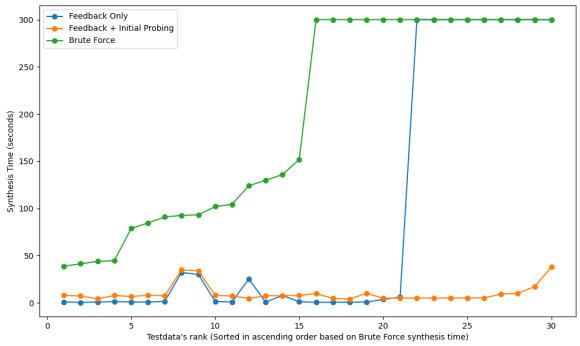

Fig. 13. Line Plot (Depth 4)

Fig. 13 shows that the Brute Force approach consistently has a larger synthesis time in comparison to the other two approaches. At a certain point of input complexity, the Feedback Only approach has a massive increase in synthesis time. The Feedback and Initial Probing approach does not show this sudden dramatic increase in synthesis time. This explains why this particular approach does not have such a disparity between its maximum and minimum in Fig. 12.

VI. Related Work

This section summarizes research papers that are related to the concepts relevant to our Boolean expression synthesis tool.

We compared our Feedback-related Boolean expression synthesis solutions against the Brute-Force approach. The related work discussed in this section pertain to the concepts used in the development of our synthesis tool, and alternative (non-CEGIS) approaches to expression synthesis.

The paper titled *Counterexample Guided Inductive Synthesis Modulo Theories* by Alessandro Abate et al. focuses on program synthesis using the CEGIS($\mathcal{T}$) approach [17]. This paper also describes basic concepts of CEGIS. The CEGIS($\mathcal{T}$) approach "...combines the strengths of a counterexample-guided inductive synthesizer with those of a theory solver, exploring the solution space more efficiently without relying on user guidance" (p. 270) [17]. The sections of this paper centering around CEGIS were useful in the development of our program synthesis tool. We used the concepts of feedback and theory solvers used in CEGIS to help inform the design of our synthesis tool.

The paper titled *The Sketching Approach to Program Synthesis* by Armando Solar-Lezama [26]. This paper describes sketching as a way for developers to use partial programs to describe their desired implementation strategy, leaving low-level implementation details for automated program synthesis (p. 1) [26]. Solar-Lezama

describes a "SKETCH synthesis engine" (p. 1) [26]. This tool derives missing details from a given sketch in order to output a program that satisfies a given set of correctness criteria (p. 1) [26]. This synthesis tool also uses CEGIS concepts in its approach, just as our tool does. The sketching strategy is discussed in further detail in Section VIII. This approach is listed as a potential means for improving our existing synthesis tool.

The paper titled *Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs* by Jifeng Xuan et al. [15] proposes the NOPOL approach for "automatic repair of buggy conditional statements (i.e., if-then-else statements)" (p. 34) [15]. This work deals with if-conditions, which are (in essence) Boolean expressions. NOPOL uses "angelic fix localization" (p. 34) [15] to determined the expected values of conditions within a program. NOPOL then collects variable data (values, data types, etc.) and encodes the data into an SMT problem instance to be solved (p. 34) [15]. This related work makes no mention of CEGIS, but it offers an interesting alternative approach to solving if-condition bugs. Unlike our tool, NOPOL uses pure constraint programming. It elegantly defines several constraints alongside the synthesis information, and inputs all this information into a SMT solver. NOPOL then translates back the result into the desired language. However, using pure constraint programming has made the tool unable to obtain an answer for many test cases (p. 44) [15].

The paper titled *Precise Condition Synthesis for Program Repair* by Yingfei Xiong et al. [14] describes an implementation of a Java program repair system (p. 417) [14]. This system, called ACS (Accurate Condition Synthesis), achieves condition synthesis in two steps: variable selection, followed by predicate selection (p. 417) [14]. For example, the steps to synthesize the condition IF($a > 10$) would be as follows (p. 417) [14]:

1) Variable Selection: a
2) Predicate Selection: > 10

Yingfei Xiong et al. found that their ACS system was able to successfully repair program defects with a relatively high level (78.3%) of precision (p. 416) [14]. This related work can be used to synthesize if-conditions, hence its overlap with our work in Boolean expression synthesis. This paper does not explicitly use CEGIS, making it another potential alternative approach to expression synthesis.

VII. Conclusions

In conclusion, our Boolean expression synthesis tool offers significant improvement in synthesis time in comparison to a "Brute-Force" approach. Both the "Feedback Only" and "Feedback and Initial Probing" approaches tended to perform Boolean expression synthesis in a shorter time than the "Brute-Force" approach. The "Feedback and Initial Probing", in particular, has consistently had the shortest synthesis time across all expression depth values.

Our data collection and analysis implies that our Boolean expression synthesis tool could have a positive impact in the area of IF-CC bug repair. A tool that can help offer repairs in such a common type of bug would offer novelty in automated program repair. Our software tool could open a new line of research into IF-CC bug repair. Research could be focused on determining faster synthesis patterns that would lead to more efficient tools for IF-CC bug repair.

VIII. Future Work

Future work for Boolean expression synthesis should focus on sketching and parallelization. Both of these concepts could help speed up the synthesis time of our tool.

Sketching is a localized program synthesis strategy that we could implement to improve our Boolean expression synthesis tool (p. 1) [26]. In our existing synthesis tool implementation, specifications (variable declarations and constraints) are given to the tool as input. The sketching approach would allow the input to include a *sketch* template, defined as "...a partial program that encodes the structure of a solution while leaving its low-level details unspecified" (p. 1) [26]. By combining our existing specifications with a program sketch skeleton, a user can give a clearer description of how they want the program to be structured. This clearer template could significantly reduce the search space, resulting in a more efficient synthesis time.

A sketching framework can be easily integrated into our synthesis tool, making it even a more efficient. Through this sketching framework, we can determine which parts of the expression must be searched for (the holes in the expression).

Looking back at Fig. 3, it is evident that independent searches can be carried out over different sections of the tree. This indicates the potential for parallelization in our algorithm. To parallelize the search, we would first partition the tree into several sections. The number of sections should be equal to the number of processing

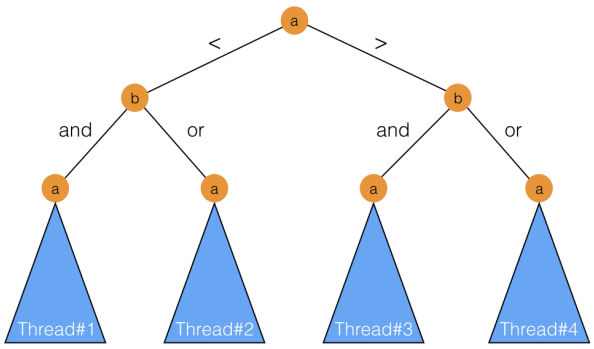

Fig. 14. Parallelization of Search

cores (of any granularity) available. It would be better if the number of processing cores that we allocate is divisible by $|C|$ (the size of the set of logical operators). Fig. 14 shows the original sample synthesis tree (see Fig. 3) broken into separate sections to be handled by different concurrent threads.

Due to the nature of the feedback-driven approach, the search time of the sections would likely be different from one another. This imposes the need for a synchronization algorithm to re-partition the tree. Parallelization could impose significant overhead on our existing algorithm. Therefore, it would be better to reserve this approach for larger inputs.

IX. Acknowledgements

We wish to thank Professor Patrick Lam for his feedback on how to improve the structure and writing of our report.

References

[1] Cordell Green. "APPLICATION OF THEOREM PROVING TO PROBLEM SOLVING". In: *Proceedings of the First International Joint Conference on Artificial Intelligence*. url: https://www.ijcai.org/Proceedings/69/Papers/023.pdf. 1969, pp. 219–239.

[2] Richard J. Waldinger and Richard C. T. Lee. "PROW: A STEP TOWARD AUTOMATIC PROGRAM WRITING". In: *Proceedings of the First International Joint Conference on Artificial Intelligence*. url: https://www.ijcai.org/Proceedings/69/Papers/024.pdf. 1969, pp. 241–252.

[3] Zohar Manna and Richard J. Waldinger. "Toward Automatic Program Synthesis". In: *Commun. ACM* 14.3 (Mar. 1971), pp. 151–165. ISSN: 0001-0782. DOI: 10.1145/362566.362568. URL: https://doi.org/10.1145/362566.362568.

[4] Zohar Manna and Richard Waldinger. "KNOWLEDGE AND REASONING IN PROGRAM SYNTHESIS". In: *Proceedings of the Fourth International Joint Conference on Artificial Intelligence*. url: https://www.ijcai.org/Proceedings/75/Papers/041.pdf. 1975, pp. 288–295.

[5] J.P. Marques Silva and K.A. Sakallah. "GRASP-A new search algorithm for satisfiability". In: *Proceedings of International Conference on Computer Aided Design*. 1996, pp. 220–227. DOI: 10.1109/ICCAD.1996.569607.

[6] Kai Pan, Sunghun Kim, and E. James Whitehead. "Toward an understanding of bug fix patterns". In: *Empirical Software Engineering* 14 (2008), pp. 286–315.

[7] Armando Solar-Lezama. "Program Synthesis by Sketching". link: https://people.csail.mit.edu/asolar/papers/thesis.pdf. PhD thesis. University of California, Berkeley, 2008.

[8] Rishabh Singh, Sumit Gulwani, and Armando Solar-Lezama. "Automated Feedback Generation for Introductory Programming Assignments". In: *SIGPLAN Not.* 48.6 (June 2013), pp. 15–26. ISSN: 0362-1340. DOI: 10.1145/2499370.2462195. URL: https://doi.org/10.1145/2499370.2462195.

[9] Luca Della Toffola, Michael Pradel, and Thomas R. Gross. "Performance Problems You Can Fix: A Dynamic Analysis of Memoization Opportunities". In: *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. OOPSLA 2015. Pittsburgh, PA, USA: Association for Computing Machinery, 2015, pp. 607–622. ISBN: 9781450336895. DOI: 10.1145/2814270.2814290. URL: https://doi.org/10.1145/2814270.2814290.

[10] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View*. Springer, Berlin, Heidelberg, 2016.

[11] Sergey Mechtaev, Jooyong Yi, and Abhik Roychoudhury. "Angelix: Scalable Multiline Program Patch Synthesis via Symbolic Analysis". In: *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. 2016, pp. 691–701. DOI: 10.1145/2884781.2884807.

[12] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard Version 2.6*. url: https://smtlib.cs.uiowa.edu/papers/smt-lib-reference-v2.6-r2017-07-18.pdf. July 2017.

[13] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*. Vol. 4. NOW, Aug. 2017, pp. 1–119. URL: https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[14] Yingfei Xiong et al. "Precise Condition Synthesis for Program Repair". In: *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. 2017, pp. 416–426. DOI: 10.1109/ICSE.2017.45.

[15] Jifeng Xuan et al. "Nopol: Automatic Repair of Conditional Statement Bugs in Java Programs". In: *IEEE Transactions on Software Engineering* 43.1 (2017), pp. 34–55. DOI: 10.1109/TSE.2016.2560811.

[16] Jooyong Yi et al. "A Feasibility Study of Using Automated Program Repair for Introductory Programming Assignments". In: *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ESEC/FSE 2017. Paderborn, Germany: Association for Computing Machinery, 2017, pp. 740–751. ISBN: 9781450351058. DOI: 10.1145/3106237.3106262. URL: https://doi.org/10.1145/3106237.3106262.

[17] Alessandro Abate et al. "Counterexample Guided Inductive Synthesis Modulo Theories". In: *Computer Aided Verification*. Ed. by Hana Chockler and Georg Weissenbacher. Cham: Springer International Publishing, 2018, pp. 270–288. ISBN: 978-3-319-96145-3.

[18] Simon Urli et al. "How to Design a Program Repair Bot? Insights from the Repairnator Project". In: *Proceedings of the 40th International Conference on Software Engineering: Software Engineering in Practice*. ICSE-SEIP '18. Gothenburg, Sweden: Association for Computing Machinery, 2018, pp. 95–104. ISBN: 9781450356596. DOI: 10.1145/3183519.3183540. URL: https://doi.org/10.1145/3183519.3183540.

[19] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. "Automated Program Repair". In: *Commun. ACM* 62.12 (Nov. 2019), pp. 56–65. ISSN: 0001-0782. DOI: 10.1145/3318162. URL: https://doi.org/10.1145/3318162.

[20] Mohammad Robati Shirzad. *Synthesizing a boolean expression based on several constraints*. StackOverflow Link: https://stackoverflow.com/questions/67924604/synthesizing-a-boolean-expression-based-on-several-constraints. 2021.

[21] Alessandro Abate et al. *CEGIS(T): CounterExample Guided Inductive Synthesis modulo Theories*. Presentation Link: https://polgreen.github.io/pdfs/cegist.pdf.

[22] Nikolaj Bjørner et al. *Programming Z3*. http://theory.stanford.edu/~nikolaj/programmingz3.html.

[23] *itertools.combinations(iterable ,r)*. https://docs.python.org/3/library/itertools.html#itertools.combinations.

[24] *JS-Buxter*. https://github.com/mohrobati/JS-Buxter.

[25] *PLY (Python Lex-Yacc)*. https://www.dabeaz.com/ply/ply.html.

[26] Armando Solar-Lezama. *The Sketching Approach to Program Synthesis*. pp. 1-10
URL: https://people.csail.mit.edu/asolar/papers/Solar-Lezama09.pdf.

[27] *Z3*. https://github.com/Z3Prover/z3.