Implementing Brahma: Synthesis of Loop-Free Programs in Python

Meyssam Rostamzadeh
Tehran Institute for Advanced Studies

Amirhossein Farahani
Tehran Institute for Advanced Studies

***Abstract*—Brahma [1] is a component-based synthesis tool that synthesizes programs in the bit-vector domain by leveraging the power of SMT solvers. Its synthesis process follows a counterexample guided iterative paradigm that in each step, first generates a partially correct program, then refines it.**

Since the authors didn't provide the implementation of their tool Brahma, in our contribution, we implemented its python version using Z3 as its SMT solver. By running 25 examples from the Brahma paper, we compared our results to those from the baseline [1] and an implementation of the baseline in the rust language [2]. The results were promising as they were comparable to the baseline's and the rust implementation's results.[1]

***Index Terms*—Component-based Synthesis, Z3 Solver, CEGIS**

I. Introduction

The composition concept is utilized in many areas of computer engineering ranging from upgrading programming languages - from simple to object-oriented ones - to system verification. Without modular-based concepts such as function calling or inheritance, programming would be much more difficult for programmers. Also, system verification would not be scalable if verifiers could not verify a system by verifying its components. This concept even structures one of the two main paradigms of program synthesis; bottom-up synthesis. Bottom-up synthesis generates sections of the program and merges them until the whole program is constructed.

The applications of composition can be leveraged in more specific domains of program synthesis, such as component-based synthesis. Component-based synthesis utilizes universal components - the most frequently used components of all programs in the specified domain - to generate a desired program. Automating this process has some advantages; first the synthesized program needs no verification as it is correct by construction. Second, the synthesized program will be optimal in terms of using the fewest possible number of components. Third, this automation can benefit developers a lot as it eliminates the need for finding the exact subset of the components needed to synthesize the desired program.

The baseline tool, which its creators call Brahma, synthesizes bit-vector programs by taking a specification of the desired program - a high level specification of the relationship between the inputs of the program and their corresponding output - and an over-approximation of its components. The program components are objectified by assigning a variable to each of their respective inputs and output. They are further objectified by assigning each of them location variables and relating each of them to their corresponding ordinary variables using constraints. By constructing some constraints that assert the necessary relationships between the parts of the components and the inputs and outputs of the program, Brahma can generate a program that satisfies the given constraints; thus generating candidate programs that can later be tested in terms of their generalizability. This test is done by a counterexample guided iterative process. In this process, first a synthesizer generates a program that maps a subset of inputs to their respective outputs. However, it may fail to map a particular subset. Later on, a verifier - similar to the mentioned synthesizer - verifies if the candidate program can be generalized to the entire set of inputs; if it fails, the verifier gives a counterexample to the mentioned synthesizer and the process repeats. After some iterations, the tool either finds the desired program or fails to do so. Each iteration of the process is done by utilizing a SMT solver to solve the mentioned constraints.

Brahma synthesizes bit-vector programs and provides good capabilities to developers of bit-vector programs. Bit-vector programs are hard to develop, even for experts in programming. For example consider a bit-vector program that takes as input two integers and outputs their average, and ceils it afterwards. This program may seem easy to implement; however, as it can cause runtime overflow, the developer should be cautious during development. Bit-vector programs are also beneficial as they substitute the ordinary programs that can cause problems, such as overflow. For example consider the below program:

$$(a + b)/2 \tag{1}$$

It takes two integers as inputs and outputs their average. This program can obviously cause overflow in the summation process. Its safe bit-vector version is:

$$(a|b) - ((a \oplus b) >> 1) \tag{2}$$

This tool also leverages the constraint-based search method that outperforms other search strategies - such as exhaustive enumerative search methods - in terms of the time it takes to find the solution. For example, finding a solution for a synthesis problem that uses n components costs $O(2^n)$

[1]The Github Repo is available at: https://github.com/meyssamrostamzadeh/LoopFreeSynthesis/

using enumerative search. Reducing the synthesis problem to a constraint-based search can put the burden on SAT or SMT solvers' shoulders, which have turned to powerful tools, especially in recent years.

In our work, we have implemented a python version of the Brahma tool and compared our experiments' results to the baseline and the rust language implementation results. Our work can be summarized by these items:

- Since there was no acceptable Python implementation of the tool Brahma [1], we implemented it using the Z3 SMT solver framework in Python. We obtained results after running 25 examples from the $ExperimentalResults$ section of the baseline. We compared our results with the baseline results and results from a rust language implementation [2].
- We resolved some of the issues and shortcomings of the baseline paper in our implementation.

In the second section, we provide a brief overview of the formulation method and the CEGIS algorithm used in the Brahma tool. In the third section, we summarize some of the functionalities we modularized into our code. Lastly, in the fourth section, we present our results and their comparison to the baseline and the rust implementation mentioned earlier.

II. Brahma

A. Logical Formulation

In this part, we explain how Brahma transforms the component-based synthesis problem into a constraint-based search that leverages the power of SMT solvers.

We partition this section into four parts, each explaining how each component of the below formulation is constructed:

$$\Psi_{wfp}(L) \wedge (\Psi_{lib}(T) \wedge \Psi_{conn}(\vec{I}, O, T, L) \implies \Psi_{spec}(\vec{I}, O))$$

We will explain each of these components with a bit-vector program that takes as input a bit-vector and turns off its rightmost 1 bit in the output.

1) Well-formedness Constraints: Well-formedness constraints - denoted with $\Psi_{wfp}(L)$ - assure that the produced programs are well-formed; having properties such as consistency, acyclicity and location-boundedness. For the aforementioned example, these constraints are compacted in the formula below:

$$\Psi_{wfp} := \Psi_{cons} \wedge \Psi_{acyc} \wedge \bigwedge_{x \in P} (0 \leq x \leq 2) \wedge \bigwedge_{x \in R} (1 \leq x \leq 2)$$

$$where\ \Psi_{cons} := (l_{O_1} \neq l_{O_2})$$

$$and\ \Psi_{acyc} := (l_{I_1} < l_{O_1}) \wedge (l_{I_2} < l_{O_2}) \wedge (l_{I_2'} < l_{O_2})$$

Where l_{O_1} and l_{O_2} are the locations of the two components used in the program and l_{I_1}, l_{I_2} and $l_{I_2'}$ are the locations from which the inputs of the components are taken from.

The consistency constraints - denoted by Ψ_{cons} - assure that we will not have two or more components on one line of the program. The acyclicity constraints - denoted by Ψ_{acyc} - assure the loop-freeness of the generated program.

2) Components' Logical Formulation: Library constraints - denoted with $\Psi_{lib}(T)$ - are comprised of the components' formulations. For the aforementioned bit-vector example, the components are formulated as below:

$$\Phi_1(I_1, O_1) := O_1 = (I_1 - 1)$$

$$\Phi_2(I_2, I_2', O_2) := O_2 = (I_2 \wedge I_2')$$

Where I_1, I_2 and I_2' are the components' input values and O_1 and O_2 are the components' output values.

3) Connectivity Constraints: Connectivity constraints - denoted with $\Psi_{conn}(\vec{I}, O, T, L)$ - assure that if some of the components' input values are equal to some other component's input or output - or the inputs or the output of the program - their corresponding locations become equal too. These constraints are summarized below:

$$\Psi_{conn} := \bigwedge_{x,y \in P \cup R \cup \vec{I} \cup \{O\}} (l_x = l_y \implies x = y)$$

Where l_x and l_y are the locations of the program's elements and x and y are their respective values. These elements are the input and output of the program, inputs of the components and their outputs.

4) Program Specification: The program specification constraints - denoted with $\Psi_{spec}(\vec{I}, O)$ - summarize the behaviour of the desired program in the form of a first-order logic formulation. For the aforementioned bit-vector example, this formulation is as follows:

$$\Psi_{spec}(I, O) := \bigwedge_{t=1}^{b} ((I_{[t]} = 1 \wedge \bigwedge_{j=t+1}^{b} I_{[j]} = 0) \implies A)$$

$$where\ A := (O_{[t]} = 0 \wedge \bigwedge_{j \neq t} O_{[j]} = I_{[j]})$$

Where b is the number of bits of our bit-vectors and I and O are the bit-vectors of the program's input and output.

B. CEGIS-based algorithm

In this part, we explain how Brahma exploits the formulation we constructed in the previous part in a counterexample guided synthesis manner.

The aforementioned method can be summarized in the two subprocesses below:

- First, the SMT solver solves the formulation we generated in the previous part of this section. In each step, the formulation will only have to be satisfied with respect to a subset of the inputs called S. S is initially empty and in each iteration, $Verifier$ adds a counterexample - an inconsistent input - to it. This subprocess generates a candidate program that maps a subset of inputs to their corresponding outputs. This subset is a superset of S. We call this subprocess $Synthesizer$.

- Second, the SMT solver will have to solve the formulation with respect to the locations - actually the candidate program - $Synthesizer$ found in the previous subprocess. If it returns SAT for all inputs (and their respective outputs), the candidate program is our desired program and the tool displays it. However, if it returns UNSAT for an input, the input is added to the S set as a counterexample. We call this subprocess $Verifier$.

These two subprocesses continue until either a candidate program satisfies all the inputs and gets verified by the $Verifier$ or no candidate program passes the verification test and the tool returns no program.

III. The Implementation

This section is dedicated to the path we have taken in implementing [1]. In the first subsection, we explain the manual implementation of the first order logic formulation of the first example of the 25 examples of the baseline. In the second subsection, we briefly explain the efforts and difficulties in the implementation process of the CEGIS-based algorithm of the baseline.

A. The Initial Implementation

For the first step of our journey, it seemed appropriate to manually construct the first order logic formulation - explained in detail in part A of the second section - of an example from the baseline paper. For the example part, the most appropriate choice was to take the first example from the 25 examples in the $ExperimentalResults$ section of the baseline paper. It was a beneficial experiment and was a significant step as it proved the authenticity of the formulation the Brahma creators proposed.

Furthermore, we noticed that this simple and fully manual implementation of the first example needs no counterexample guided procedure. It was because when you assign its input and output variables to a pair of I/O values, it produces the correct program in just one run. We repeated the same procedure on the seventh example and got the same result. The respective codes are in the GitHub repo.

B. CEGIS Implementation

In the second step, it was crucial to address some of the bold shortcomings of the baseline. We encountered four issues in this implementation phase:

- We knew that to find a candidate program using SMT solvers, we needed to generate the $\Psi_{spec}(\vec{I}, O)$ part of the formula automatically. However, the baseline did not mention anything about the specification they took from the user in their experiments; they left this part vague.
- They have also not specified their intent by saying that the user must provide the synthesizer with an overapproximation of the required components of the desired program. This overapproximation could mean any subset of the all possible components of the bit-vector program space.
- In the CEGIS-based algorithm - explained in detail in the second part of the previous section - the $Verifier$ adds counterexamples to the S set at each step. At the next step, the $Synthesizer$ should find a candidate program that satisfies all the counterexamples. This is an implementation difficulty as there is just one set of value variables that can correspond to just one pair of I/O values:

$$(\vec{I}, O) \iff \forall_{1 \leq i \leq n} (\vec{I_i}, O_i)$$

  Where n is the number of components. The problem is that in the formula from which the $Synthesizer$ must produce a candidate program, we cannot assign more than one value to the $(\vec{I}, O)$ pair.
- Brahma does not just take the program specification from the user; it also takes an overapproximation of the required components. There was a sense of unfairness in this expectation, and we began to think about how we can handle it.

After a long discussion, we came to a conclusion about the first problem. We chose to use the lambda expression available in the Python language. This way, both implementation and user interaction would be easier. As an example, the following expression would hold for the "turning off the rightmost 1 bit example":

$$lambda \;\; y, x : y == (x \wedge (x - 1))$$

For both the second and fourth problems, we decided not to ask the user to give our synthesizer any component. This way the user would not have to traverse through all the components to find the appropriate ones for his/her program. Instead, we first exploited the entire set of components for each example. However, this led us to some runtime difficulties; the tool timed out on some examples, especially the last ones. As a result, we created and exploited 11 subsets of the components set. These subsets help $Synthesizer$ find more feasible candidates at each step, leading the tool to converge to the desired program faster. After utilizing these subsets in the $\Psi_{lib}(T)$ part of the formulation, the time out problem was resolved.

For the third problem, we decided to create an entire new set of value variables each time a counterexample is added to the S set:

$$\forall_{1 \leq i \leq m} (\; (\vec{I_i}, O_i) \iff \forall_{1 \leq j \leq n} (\vec{I_{i,j}}, O_{i,j}) \;)$$

Where m is the size of S set and n is the number of all the components exploited. This way, we can have more than one I/O value pair and the $Synthesizer$ will be able to find candidate programs that satisfy all the counterexamples - the I/O values in the S set - at once. Note that the location variables are shared between these m formulations, as the $Synthesizer$ is obligated to find one candidate program.

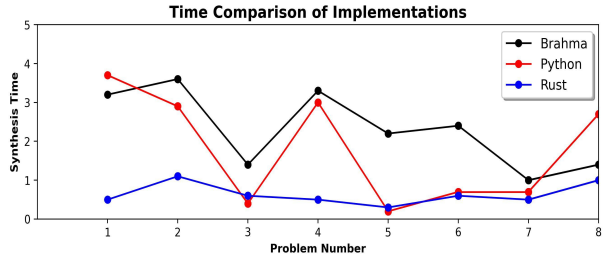

Fig. 1. Our results' comparison with [1] and [2] - specified with $Brahma$ and $Rust$ in the legends of the chart - in the first 8 experiments. These results confirm that our implementation is comparable to the Rust implementation and to the baseline's. The times are in seconds.

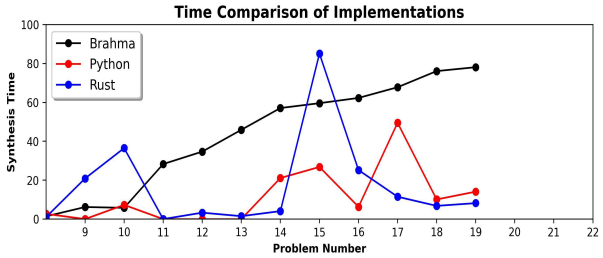

Fig. 2. Our results' comparison with [1] and [2] in the experiments 9-19. Note that in some experiments, our implementation has provided the best results.

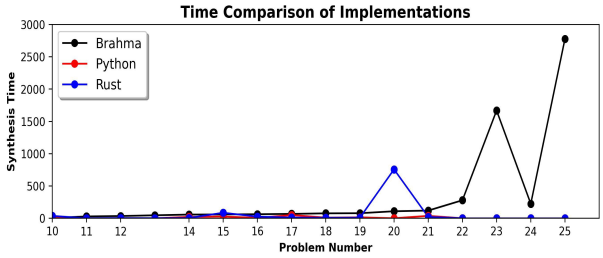

Fig. 3. Our results' comparison with [1] and [2] in the experiments 20-25. Note that in the last four experiments, both we and the rust implementation have failed to synthesize the respective programs.

IV. Experiments and Results

We have turned the 25 examples from the $Experimental\ Results$ section of the baseline paper into their lambda version using Python's built-in functions and Z3's functions. However, we were unable to do it on the 18th example as it had $Reduction\ OR$ component, which was not found in both Python's and Z3's functions. We did our experiments using 11 manually-made library sets. Nine of them were utilized explicitly for their unique examples and 3 of them were exploited for the rest of the examples.

Figure 1 illustrates the results of running the first 8 examples from the aforementioned examples. It also compares them to their respective results from the baseline's and the Rust implementation's. From the figure, we can observe that both the results from our implementation and Rust's are almost always better than the reported results from the baseline's. This is realizable as the SMT solvers and computational resources were much weaker when the baseline paper was published. We can also observe that although in some examples, we have obtained weaker results than Rust's, our results are slightly better than Rust's in some examples.

Our results from experiments 9-19 are more interesting. Figure 2 illustrates those results and their comparison to their respective results from the baseline's and the Rust's. We can observe that in all examples (except one), our implementation provides better results than the baseline's. Also, we can observe that in 7 out of 11 examples, we have provided better results than Rust's.

And finally, from the figure 3 we can see that we have achieved better results than both other implementations. However, even by setting a 3600 second timeout limit, our synthesizer could not synthesize the programs from examples 22-25. [2] authors also reported their tool's crash on some of these examples; their tool also could not synthesize these examples. This can be a valid reason to suspect the Brahma's results for these examples.

From all these experiments, it can be concluded that our implementation satisfies the prerequisites of a successful implementation as it can synthesize 20 out of 25 examples from the baseline's benchmark. Further comments and details about the implementation are available in the GitHub repo.

V. Conclusions

Brahma is a component-based synthesizer that uses the power of SAT and SMT solvers to utilize and arrange the given components which leads to the construction of a program. We have implemented a Python version of this tool and compared it with the baseline and a Rust implementation. We illustrated our implementation's performance by designing an experiment that compares the speed of our implementation in synthesizing the 25 examples from the baseline paper's benchmark to the speed of the baseline's and Rust's synthesis for those examples. The results were promising as they were either close to or better than the baseline's and Rust's in more than half of the examples.

References

[1] Gulwani, S., Jha, S., Tiwari, A., and Venkatesan, R. (2011). Synthesis of loop-free programs. ACM SIGPLAN Notices, 46(6), 62-73.
[2] https://github.com/fitzgen/synth-loop-free-prog