Evolving Design Patterns for Program Synthesis

Fabian Radomski
Supervisor: Sebastijan Dumančić,
EEMCS, Delft University of Technology, The Netherlands

June 19, 2022

A Dissertation Submitted to EEMCS faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

Evolving Design Patterns for Program Synthesis

Fabian Radomski
Supervisor: Sebastijan Dumančić
EEMCS, Delft University of Technology, The Netherlands

Abstract

Design pattern provide an abstraction that the program synthesis algorithm can use in order to find programs easier. However, coming up with them is difficult as they are domain-specific. This paper showcases a novel approach to creating design patterns through the means of genetic algorithms. Results are showing that while in the robot domain the average time is slightly better, string domain shows that using the patterns actually lowers the accuracy and average time.

1 Introduction

Program synthesis is the task of creating programs from a specification. This specification can come in different forms - one example being a collection of input-output relations in the case of inductive program synthesis (IPS)[1]. There are many use cases where this technique would be effective, especially in the field of software engineering. It can involve a lot of tedious work that the machine could potentially do instead of the developer who would only provide the examples. Even though easy synthesis of any program is not yet possible, there have been successful commercial applications, such as the FlashFill feature in Excel.

One of the main challenges of program synthesis is the searching for the program that satisfies the specification in the space of programs. The difficulty comes from the fact that the space is of infinite size because there exist infinitely many programs to check. Additionally, as the task becomes more complex and needs bigger programs, the search space grows in an exponential manner. Fortunately, there are ways to reduce the size of it.

Design patterns are programs with parameters that can be used as abstractions when searching for a program to make the synthesis easier [2]. They could be likened to software design patterns which help developers to apply recurring and reusable solutions to common problems in software design [3]. To illustrate them more concretely, consider an example robot-planning task illustrated in Figure 1. In this domain the robot starts in a grid represented by the input state in Figure 1a. It can move step by step in all 4 directions, pick up or drop the ball in order to leave the grid exactly as in the output state shown in Figure 1b. A straightforward program that solves

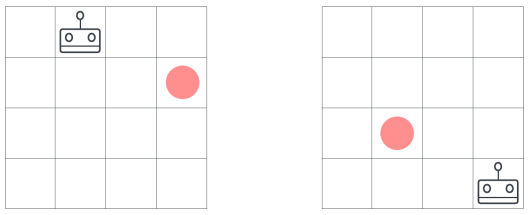
(a) Input state (b) Output state

Figure 1: The input and output states of the robot-planning domain showing a 4x4 grid with the robot and ball locations indicated by the corresponding icons. The task is to find a sequence of moves for the robot to move itself and the ball as indicated in the output.

this particular example could look like the one in Figure 2 (left). It is built out of 11 expressions and there is a certain pattern: the robot moves twice in either the left or the right direction and then moves down. We can use this observation and introduce a design pattern like the following:

```
DoubleAndDown(Expression):
    Expression
    Expression
    MoveDown
```

and by implementing it in a program showcased in Figure 2 (right), the program size has been reduced to only 5 (compound) expressions. This illustrates that appropriate design patterns allow the synthesiser to make leaps in the program space to find the programs in less time and more easily.

But how does one create these design patterns? It appears finding a set of design patterns that really make synthesis more efficient is not an easy task. In the worst scenario, it would require a deep understanding of the domain and manual combination of different expressions in an attempt to create successful patterns. However, it is promising to look into computational methods that will try to generate them in an automatic way instead.

The goal of this research is to examine the use of Genetic Algorithms (GA) in finding design patterns. GA is a method for obtaining high quality solutions to search problems inspired by the natural selection process and evolution [4]. The idea behind GA is to move from one population of candidate

```
MoveRight
MoveRight
MoveDown
Grab
MoveLeft
MoveLeft
MoveDown
Drop
MoveRight
MoveRight
MoveDown
```

```
DoubleAndDown(MoveRight)
Grab
DoubleAndDown(MoveLeft)
Drop
DoubleAndDown(MoveRight)
```

Figure 2: Example program for the aforementioned task (left) and a program that uses the DoubleAndDown(Expression) design pattern (right)

solutions to the next one by using operators inspired by genetics: selection, crossover and mutation. It has proven to be effective in many different scenarios like predicting protein structure, scheduling problems, or even program synthesis itself [5].

I claim that the design patterns generated with Genetic Algorithms are going to increase the accuracy of program synthesis, compared to using no design patterns. To this end, I will answer these research questions:

1. Can the design patterns evolved with GA increase the accuracy of program synthesis?
2. How to define genetic operators that lead to better design patterns in future generations?
3. What is the optimal fitness function that evaluates the performance of a chromosome?

In this paper I will describe the following contributions:

- The implementation of the Genetic Algorithm for evolving design pattern with its operators is described
- The design patterns are evaluated in two domains: robot-planning and string manipulation.

2 Related work

This section will introduce the necessary background information and work that is related to this research. First two subsections present an explanation of the contributions of a team of students who did research on search procedures of program synthesis in the second quarter of academic year 2021/2022 as it is the code- and knowledge base for this research. The repository with the code can be found under [6]. Lastly, an overview of genetic algorithms is given.

Domain Specific Language

A *program* is a concrete description of how to solve a certain task. This requires a notation, a programming language, that defines the atomic expressions (from now on called tokens) and how to compose them in order to describe computation [1]. In this implementation, the programs are structured as a list of tokens. The interpreter sequentially applies each token to the input environment. General-purpose programming languages like C++, Java, or Go have a lot of syntax elements that are not all useful for the particular task at hand. For this reason, it is typical to use a **Domain Specific Language (DSL)** which is a narrow, specialized language which contains a limited set of tokens that are appropriate for the domain. As an example consider the tokens and the program in Figure 2.

In terms of the effect, there are three types of tokens: transition, boolean, and control. Transition tokens act on a given state and result in another state. Boolean tokens return true or false depending on the state of the environment. Finally, control tokens influence the control flow of the program. There are two of them: a while loop construct Loop and a conditional construct If. The Loop token takes a condition and a body. When applied, the body will be executed as long as the condition yields true. The If token takes a single boolean and two sequences of tokens. On application, it evaluates the boolean and based on whether it is true or false, applies the first or the second branch respectively.

Program Synthesis

Program synthesis is the approach of generating programs from a specification. In the implemented *Inductive Program Synthesis* system the specification comes in the form of examples. Each example consists of an input state and an output state. The state is represented as an environment object and holds important properties.

Program synthesis can be perceived as a search problem. The space of the programs is defined by all of the programs that can be described by the DSL [1]. Then, the synthesiser will try to find a program that matches the examples as closely as possible. Rather than using a binary decision to decide if a program entails the examples or not, a distance function is used to decide how far the output is from the example output [7]. The code implements 6 different search procedures: Brute, Vlute, Monte Carlo Tree Search, VanillaGP,

Three different domains have been implemented, each with its environment, DSL, and distance functions.

Robot planning has already been touched upon in the introduction. The state contains the size of the square grid and the x- and y-coordinates of the ball and the robot. The transition and boolean tokens for the domain are [MoveUp, MoveDown, MoveLeft, MoveRight, Grab,Drop] and [AtLeft,NotAtLeft, AtRight, NotAtRight, AtTop, NotAtTop, AtBottom, NotAtBottom]

In the **string transformation** domain the objective of the synthesis is a program that takes a string and changes it according to an arbitrary rule; for instance capitalizing each letter or extracting just the numbers. The input state is the initial string and the index of the string where the cursor is located. The output state is the transformed string. These are the transition tokens: [MoveRight, MoveLeft, MakeUppercase,MakeLowercase, Drop]. It also has these boolean tokens: [AtEnd, NotAtEnd, AtStart, NotAtStart, IsLetter, IsNotLetter, IsUppercase, IsNotUppercase,IsLowercase, IsNotLowercase, IsNumber, IsNotNumber, IsSpace, IsNotSpace].

Genetic Algorithms

Genetic Algorithm is a heuristic often used in optimization or search problems [8]. It bases on the idea of ”survival of the fittest". The principle behind is to evolve a population of solutions with the aim of obtaining better solutions in next generations.

Each of the solutions is called a chromosome. A chromosome is conceptually divided into genes. The values that a single gene can take are called alleles. In the simplest and typical representation, a chromosome could be a bit string where each allele can be a 0 or a 1 [4]. The fitness function associates a value to each chromosome that represents how good is the solution to the problem.

A genetic algorithm works by mimicking evolution by means of these operators:

- *Selection* operator chooses chromosomes from population that will be reproduced. On average, the individuals with the highest fitness will be selected.
- *Crossover* operator swaps subparts of two chromosomes, thus producing 2 new child chromosomes. It imitates the recombination of chromosomes, which is the key to creating better offspring according to [8].
- *Mutation* operator randomly changes genes of a chromosome with some probability.

These steps are repeatedly applied to each generation, finishing when a stopping condition is met (e.g. target fitness or number of generations).

3 Methodology

This section will lay the grounds to the problem of evolving design patterns. First, the implementation of design patterns is described. Then, there is a description of the GA approach along with the reasoning behind the choices.

3.1 Design patterns

The design patterns are implemented in resemblance to programming language functions with a single parameter. As elements of the language in the existing system are represented by tokens, I decided to keep it this way. Similarly to *InventedTokens*, the design pattern, implemented in the *PatternToken* class, has a list of tokens, the difference is that each token can either be a transition token or a *FunctionVariableToken*. The class also holds a reference to a parameter token, although this only serves as a building block for extending design patterns to multiple parameters. *PatternApplicationToken* contains the design pattern itself and a single argument transition token. When it is applied, the interpreter will go over the patterns and replace all occurences of the variable token with the argument token. In order to be used by the search procedures, these design patterns applications are instantiated with each of the transition tokens as an argument and added to the set of invented tokens.

3.2 Genetic Algorithm

Chromosome encoding

The goal of this genetic algorithm is to find a set of design patterns. There are two possible chromosome designs that have been considered. The first design was to code a single design pattern as a chromosome. This way the resulting design patterns would consist of a number of chromosomes with the highest fitness from the very last generation of chromosomes. Unfortunately, this makes it difficult to calculate fitness as a single design pattern doesn't have big of an impact on the synthesis. The second design expressed the chromosome as a list of design patterns. This gives a direct result as a single chromosome with the highest fitness. This approach was found to be beneficial for a number of reasons. First of all as the goal is to generate a set of design patterns, this would give one that takes into account that different patterns might be useful for different tasks and therefore complement each other. It also enables an easier evaluation when computing the fitness of a chromosome.

Fitness function

The fitness function is used to calculate how fit an individual solution is. This fitness reflects how well an IPS system that uses a set of design patterns performs. The process of evaluation of a chromosome starts with performing the synthesis and gathering the results. The synthesiser runs a suite of tasks (each consisting of one or more examples) and for each of them generates a program. Every program is evaluated and returns a number of different statistics, most relevant for this case being the number of correct examples and execution time.

The formula for the fitness function that was designed is shown in the Equation 1, where c_{avg} is the average of the fraction of correctly solved examples over all tasks and t_avg is the average execution time over all tasks.

$$fitness = \begin{cases} c_{avg} \cdot \frac{1}{t_{avg}} & \text{if } t_{avg} \neq 0 \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

Another version of the fitness function was considered where instead of the fraction of correctly solved examples, it used the average profit over tasks defined as $p_{avg} = 1 - cost_{avg}$. This idea , however, did not come to fruition as the average cost is different across domains (string transformation counts the cost differently) and distance functions.

Population

Once we have a way to encode a chromosome, we need to generate an initial population of them. It consists of a number of chromosomes. To produce the much wanted diversity, each chromosome is generated randomly, such that it contains from two to five design pattern. To limit their complexity, each one of them has a body length between two and six with one to six parameter occurrences.

The size of the population p, denoted as s_p is a parameter that should be chosen, such that it explores a vast area of the search space. However, the bigger the population, the more computationally expensive the evolution is going to be. For this reason, after manual testing and checking standards mentioned in [4] this number was chosen to be between 50 and 100.

Genetic operators

The selection operator is responsible for choosing the pairs of chromosomes for reproduction. The operator that has been designed first selects a number of solutions, denoted e, with the highest fitness, it is usually a fraction of all chromosomes: $e = f * s_p$ where $f \in [0.1, 0.2]$. They are preserved and

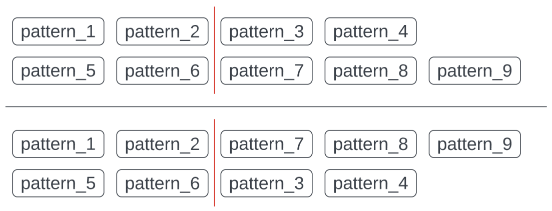

Figure 3: A point of crossover is chosen within the smaller chromosome (top). The subparts of the chromosomes are exchanged around that point, producing two new chromosomes(bottom).

added straight to the new population. Then, it selects pairs from the whole generation with roulette wheel sampling [9], which is a method that associates probability of being that is proportionate to the fitness of a chromosome and applies the next operators: crossover and mutation until there is $s_p - e$ offspring in total.

The crossover operator implements a simple one-point crossover. It chooses an index in the pattern with a smaller number of patterns and swaps the subarrays, pictured in Figure 3. Note that this method will not result in too long or too short chromosomes.

The mutation operator has had 3 possible implementations under consideration. The first one added a random design pattern to a chromosome. The second one deleted a random design pattern from the chromosome under mutation. The last one replaces a random pattern with a new, random pattern. To simplify the development and possible problems with size of the chromosome with the first two, only the third one stayed in the final version.

4 Experimental work and results

This section will focus on the setup of the experiments and what results have been collected. It will also mention how certain parameters were picked in order to be used for the final evaluation.

4.1 Crossover and mutation probabilities

Crossover and mutation probabilities dictate how often, statistically, the respective operators are applied. The genetic algorithm have been run on 50 test cases of the robot domain for a certain range of values five times to pick those, which show the highest average fitness. All of the experiments used the A* search algorithm and the RobotOptimizedSteps search setting.

In the Table 1 are the results for the mutation probabilities, $p_c = 0.85$.

In the Table 2 are the results for the mutation probabilities, $p_m = 0.003$ achieving the best score by a small margin.

4.2 Robot planning

The robot domain has been evaluated on a test set containing 250 test cases, from all complexities, all tasks and trials 1-5.

p_c	$avg(f_{max})$
0.60	40.982
0.65	41.763
0.70	41.513
0.75	41.035
0.80	41.435
0.85	42.783
0.90	42.006

Table 1: Averaged maximum fitness for different crossover probabilities.

p_m	$avg(f_{max})$
0.001	46.642
0.003	47.137
0.005	43.780
0.010	43.044

Table 2: Averaged maximum fitness for different mutation probabilities.

To evaluate the genetic algorithm it has been run with different maximum search times. Figure 4 shows a graph of the average fitness of a population in consequent generations for 4 values of search times. As seen in the graph, the average fitness rises over generations but it reaches its peak around 15th generation.

For search time of 0.1s, the best pattern achieves average of correct test cases of 1.0 and time average of 0.01496s. To compare, the default synthesis, meaning no patterns available, for the same search time achieved the same average of correct test cases and a time average of 0.01543s. The best performing set of design patterns is

- MoveDown, FunctionVariableToken, MoveLeft
- FunctionVariableToken, FunctionVariableToken, FunctionVariableToken
- MoveUp, MoveLeft, Drop, FunctionVariableToken, Grab
- MoveDown, FunctionVariableToken, FunctionVariableToken, FunctionVariableToken, FunctionVariableToken, FunctionVariableToken

Last thing to check is the index of the generation where the best chromosome was found. It turns out that it was the consistently very first generation.

4.3 String transformation

The results for string transformation are not as promising. Same type of evaluation has been performed as on the robot domain, shown in Figure 5. The resulting fitness however is only lower at 0.6711 compared to using no design patterns with fitness of 0.6956.

5 Conclusions/Future Work

The question posed in this report was whether the design patterns evolved with Genetic Algorithms can improve the accuracy of program synthesis. The experiments show a slight

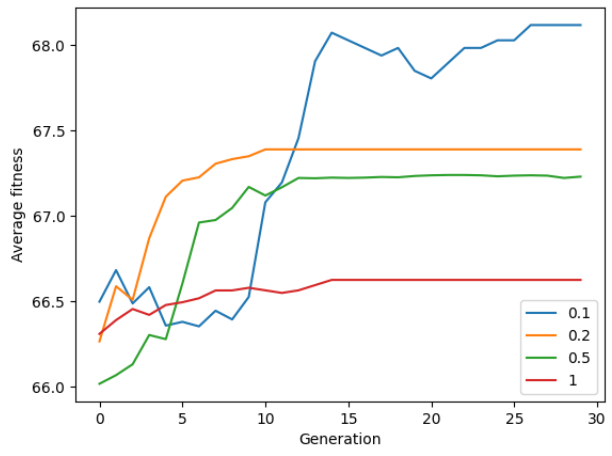

Figure 4: Average fitness across generations in the robot domain, shown for different maximum search times in seconds.

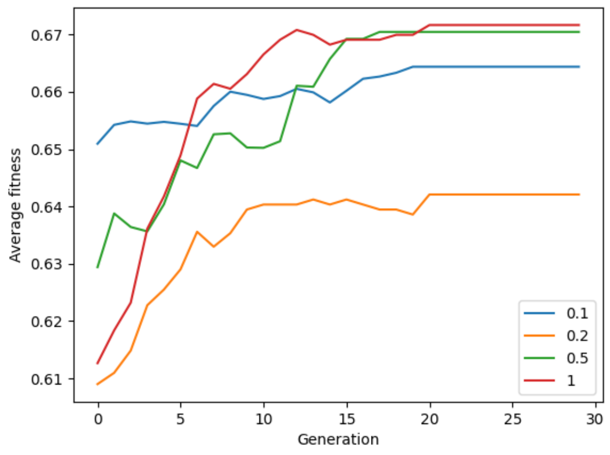

Figure 5: Average fitness across generations in the robot domain, shown for different maximum search times in seconds.

increase of fitness in the robot planning domain, compared to using no design patterns. However, this difference is not really significant and stems from a decrease of average execution time and not the improved accuracy. The string transformation domain shows a decrease in both average accuracy and average time.

Generating the best chromosome in the very first population is also a sign that this approach is rather inefficient. This means that instead of the GA approach the solution could have been found in one bigger set of random solutions and that crossover and mutation operators do not serve any help in evolving a solution towards a global optimum.

As per possible improvements, there is quite a lot of things that could be researched in future work and would've definitely been done if the time had allowed.

The first improvement would be regarding the design pattern implementation itself. Right now, there is a single parameter and that leaves the number of generated design pattern application tokens quite small. Enabling more parameters would help because each new parameter increases the number of possibilities by factor of the number of transition tokens. Another possible performance boost could be achieved by not using the application token and simply reusing the already implemented invented token. This would require substituting the variable tokens directly, but in the long run it would reduce checking each token if it is a variable token during application.

6 Responsible Research

The current state of development of computer science and computer technology in the world allows us to speak of a new, completely different level, or even a virtual world, in which different laws and mechanisms operate just as in the real world. These virtual laws, like the laws of the real world, require their discovery through a multitude of researches and subsequent researches in order to refine them and search for ever new applications. It is commendable to engage in such research because it allows us to broaden the horizon of the virtual world in order to improve the functioning of the real world, which includes the virtual world of computers. For these worlds are closely intertwined. [10]

For this reason, researchers should be aware of scientific ethics, which puts honesty and sincerity first in the conduct of studies and the recording of results. If one manipulates observations or fails to pay attention to their accuracy, there is a certain discrepancy, even if these movements do not change the hypothesis, because if the next person repeats the research process step by step and receives different results, then the question arises which study was carried out correctly and whether we can still follow the thesis that the study was supposed to prove or disprove.

The second point arising from scientific ethics is the expectation of positive results. The point is that some scientists, if you can call them that at this point, dazzled by potential fame and recognition, are able to build themselves an experiment so that they get the results they expect or confirm their hypothesis, and so become discoverers of a law or a method. This is often not reflected in the real function of the virtual world. They forget that negative research results are also necessary in the world of science, because they prove that some methods are wrong and must be ruled out. This helps to carry out further studies, as fewer hypotheses or methods for starting the experiments allow them to be carried out more reliably and easily.

The problems described above, which arise at the level of proof of hypotheses and methods by scientific research, i.e. failure to adhere to scientific principles, are primarily due to the desire of researchers to seek their own glory, and not only in the field of computer science. Scientists must, above all, keep an eye on the welfare of science, its truthfulness and its excellence. Not the money, the fame, the famous name, but the knowledge that, thanks to our work, science has taken a further step in its development, not necessarily by proven new theses, but also by refuted hypotheses. But in a world of selfishness, hatred, lack of peace and the desire to claim everything for oneself at the expense of others, scientific ethics is falling out of the foreground. It is no longer a question of the common good, but of the individual. This attitude must be combated by showing the world the beauty of science, which is true and reliable. May the development of science not be a means of earning money, and above all not a source of good from which everyone can draw, and therefore everyone should contribute something of their own. It is up to us to decide in which direction computerization will develop in the coming years. Let's make sure she turns to the truth.

References

[1] Armando Solar-Lezama. Introduction to program synthesis, 2018.

[2] Sebastijan Dumancic, Tias Guns, and Andrew Cropper. Knowledge refactoring for inductive program synthesis, 2021.

[3] Erich Gamma, Richard Helm, Ralph Johnson, Ralph E Johnson, John Vlissides, et al. *Design patterns: elements of reusable object-oriented software*. Pearson Deutschland GmbH, 1995.

[4] Melanie Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[5] Farhad Azimzade and Sebastijan Dumančić. Vanillagp: Genetic algorithm for inductive program synthesis, 2022.

[6] Farhad Azimzade, Bas Jenneboer, Nadia Matulewicz, Stef Rasing, and Victor van Wieringen. Bep_project_synthesis. https://github.com/AlgTUDelft/BEP_project_synthesis, 2022.

[7] Andrew Cropper and Sebastijan Dumančic. Learning large logic programs by going beyond entailment. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, pages 2073–2079. International Joint Conferences on Artificial Intelligence Organization, 7 2020. Main track.

[8] SN Sivanandam and SN Deepa. Genetic algorithms. In *Introduction to genetic algorithms*, pages 15–37. Springer, 2008.

[9] DE Goldberg. 1989,'genetic algorithms in search, optimization, and machine learning', addison-wesley. *Reading*.

[10] Nikolaos Efthymiou, Lucas Kroes, Michał Okoń, Fabian Radomski, and Philip Tempelman. Evolving program synthesis. https://github.com/FabianRadomski/EvolvingProgramSynthesisers,2022, 2022.