applied sciences

MDPI

Article

From Requirements to Source Code: Evolution of Behavioral Programs

Roy Poliansky [1], Moshe Sipper [2] and Achiya Elyasaf [1,*]

[1] Department of Software and Information Systems Engineering, Ben-Gurion University, Beer-Sheva 8410501, Israel; roypoli@post.bgu.ac.il
[2] Department of Computer Science, Ben-Gurion University, Beer-Sheva 8410501, Israel; sipper@bgu.ac.il
* Correspondence: achiya@bgu.ac.il

Abstract: Automatically generating executable code has a long history of arguably modest success, mostly limited to the generation of small programs of up to 200 lines of code, and genetic improvement of existing code. We present the use of genetic programming (GP) in conjunction with context-oriented behavioral programming (COBP), the latter being a programming paradigm with unique characteristics that facilitate automatic coding. COBP models a program as a set of behavioral threads (b-threads), each aligned to a single behavior or requirement of the system. To evolve behavioral programs we design viable and effective genetic operators, a genetic representation, and evaluation methods. The simplicity of the COBP paradigm, its straightforward syntax, the ability to use verification and formal-method techniques to validate program correctness, and a program comprising small independent chunks all allow us to effectively generate behavioral programs using GP. To demonstrate our approach we evolve *complete* programs *from scratch* of a highly competent O player for the game of tic-tac-toe. The evolved programs are well structured, consisting of multiple, explainable modules that specify the different behavioral aspects of the program and are similar to our handcrafted program. To validate the correctness of our individuals, we utilize the mathematical characteristics of COBP to analyze program behavior under all possible execution paths. Our analysis of an evolved program proved that it plays as expected more than 99% of the times.

Keywords: genetic programming; behavioral programming; code generation

Citation: Poliansky, R.; Sipper, M.; Elyasaf, A. From Requirements to Source Code: Evolution of Behavioral Programs. *Appl. Sci.* **2022**, *12*, 1587. https://doi.org/10.3390/app12031587

Academic Editor: Mauro Castelli

Received: 18 December 2021
Accepted: 29 January 2022
Published: 2 February 2022

1. Introduction

Search-based software engineering (SBSE) applies search-based techniques to software engineering tasks, of which an important one is the improvement and generation of software. This task is difficult to achieve due to several reasons, including the complexity of modern programming languages, the difficulty in evaluating the generated software, and the linearity of code—a small change might "break" the entire program's behavior. To overcome these difficulties we propose the use of genetic programming (GP) to evolve *behavioral programs*.

Context-oriented behavioral programming (COBP) [1] is a software development and modeling paradigm designed to allow users to program reactive systems in a natural and intuitive manner that is aligned with how humans perceive the system requirements. A behavioral program consists of a set of context-dependent scenarios (that state what to do) and anti-scenarios (that state what not to do), which are interwoven at run-time to generate a combined reactive system. Each scenario and anti-scenario is specified as a sequential thread of execution that isolates a specific aspect of the system behavior, preferably an individual requirement; it is referred to as a *b-thread*. An application-agnostic execution mechanism repeatedly collects these scenarios, chooses actions that are consistent with all the scenarios, executes them, and continuously informs them of each selection. We elaborate on COBP in Section 3.

A behavioral program has unique characteristics that makes it an excellent candidate for code evolution:

- Repetitive structure, which allows for a simple representation of a program as an abstract syntax tree (AST).
- Small independent components that are easier to change and adjust, since in many cases breaking one b-thread has little effect on the other b-threads.
- Synthesis, formal reasoning, and verification algorithms can be used to evaluate a generated program's performance.

In this paper we demonstrate how COBP programs can be evolved through GP, showcasing the benefits of COBP by focusing on the game of tic-tac-toe. Specifically, we evolve *from scratch* a highly competent O player specified as a set of b-threads. Creating such a player requires the specification of different behaviors, such as place third "O" when possible, block "X" from winning, etc. We will show evolution created from scratch a complete program that consists of different modules that specify these behaviors. Most work on code evolution focuses on genetic improvements of existing code and code evolution of small modules, such as methods (elaborated in Section 2). Thus, while tic-tac-toe may be seen as a toy problem, creating a complete program from scratch is complicated. We are aware of no attempt at generating from scratch a complete program for tic-tac-toe.

As we will show, our evolved individuals have the repetitive structure of behavioral programs. Most modules of our best individual are self-explanatory and adhere to a known strategy for the game.

For evolving COBP programs, we design the following: a domain-independent grammar for behavioral programs, augmented with terminals for tic-tac-toe; type-based, COBP-oriented genetic operators; and a fitness function that uses a behavioral program's flow of events.

The contributions of our work are as follows:

1. We provide a methodology and tools for evolving complete programs from scratch.
2. We design a domain-independent grammar for behavioral programs.
3. We design domain-independent, type-based, COBP-oriented genetic operators.
4. We evolve a highly competent player for tic-tac-toe.
5. Our evolved programs are well structured, consisting of multiple modules that are explainable and similar to our handcrafted program.

We first present a short survey on the field of SBSE and code-generation in Section 2, followed by a short primer on behavioral programming in Section 3. We then explain our method in Section 4 in detail, and present our results in Section 5. Finally, we present concluding remarks in Section 6.

2. Previous Work

The earliest example of code generation is probably that of [2], who evolved small multiplication functions using two specially designed languages. Koza notably took this idea several steps forward [3]. A more recent example is that of Orlov and Sipper [4,5], who were able to generate Java functions by evolving Java bytecode directly. Despite these examples and others, automated programming has never managed to achieve a substantial breakthrough. In fact, Dijkstra stated that automated programming was a contradiction in terms [6]:

> ...computing science is——and will always be——concerned with the interplay between mechanized and human symbol manipulation, usually referred to as "computing" and "programming", respectively. An immediate benefit of this insight is that it reveals "automatic programming" as a contradiction in terms.

Petke et al. [7] explains that many influential authors regard the challenge of automated programming to be simply unattainable, and thus more recent research focuses on automated testing and genetic improvements of software, rather than complete automated code generation.

Genetic improvement (GI) uses GP and other automated search methods to improve upon existing programs. The benefit of improving existing code instead of generating it

from scratch is clear: GI starts with an existing base and as such involves a far more focused search space. GI touches on a number of different topics such as program transformation, program synthesis, GP, testing, and SBSE [7].

GI uses the fact that naturally occurring code is very repetitive [8,9], and as such the practical space of naturally occurring code is much smaller than its theoretical space. We examined several core GI papers (defined as such by Petke et al. [7]) to understand how these semi-generated programs can be represented, evaluated, and evolved.

2.1. Representation

There are number of ways to represent programs for GI, such as source code, bytecode, and the most common way—abstract syntax trees (ASTs). ASTs are a very natural approach to represent programs for use with GP techniques, since they are already in tree form. ASTs can also be applied to many different types of languages, including C [10,11], Java [11], Python [12], and b-programs. When introducing *GenProg*, a tool for automatic bug repair, Le Goues et al. [13] represented individuals as patches, a sequence of edit operations for an AST, a solution that drastically improves space complexity for large programs.

Another viable representation is binary or bytecode representation, examples of which include Schulte et al. [14], who repaired defects in ARM, x86 assembly, and ELF binaries, and Orlov and Sipper [4,5], who evolved Java bytecode. This representation offers several benefits, stated by Schulte et al. [15], including potentially being applicable to many programming languages, a small set of instructions, and simple syntax; these benefits also apply to b-programs.

Evolving b-programs combines the benefits of both approaches: We employ the natural and convenient representation of ASTs, while still maintaining benefits gained from a binary representation, since the paradigm has been implemented in many underlying languages, including Java [16], JavaScript [1], C [17], LSC [18], and more.

Petke et al. [7] noted that a natural question in GI is concerned with translating code for parallel processing, and cited early work on *Paragen* [19,20]. It is worth noting that COBP is inherently parallel—b-threads run internal logic in parallel [16].

2.2. Evaluation

Most GI works use testing as the main fitness criterion, with the simplest way to calculate fitness based on the number of total tests passed [21,22], or using a weighting of tests [23]. Arcuri and Yao [24] evaluated fitness using a distance function based on a formal software specification. They also introduced the process of co-evolving test cases, and generating unit tests that pass on the original program but fail on the modified ones.

Overall, it is apparent that testing is the most dominant evaluation parameter for fitness calculation; however, with COBP, formal methods and verification techniques can also be used to evaluate the correctness of b-programs, due to the mathematical properties of COBP semantics.

3. A Short Primer on Context-Oriented Behavioral Programming

In context-oriented behavioral programming, requirements are implemented *independently* from each other. Each requirement is encapsulated by a behavioral thread, or *b-thread*. A collection of such b-threads forms a *b-program*. The COBP paradigm has emerged from the behavioral programming paradigm [16], extending it with context idioms that enable the definition of context-dependent requirements. In this paper we use a JavaScript implementation of COBP, called COBPjs (https://github.com/bThink-BGU/BPjs-Context, accessed on 28 January 2022). The code in this paper can be found at https://github.com/RoyPoli99/BPCodeGenerator, accessed on 28 January 2022.

To render these ideas more concrete we demonstrate the paradigm on the game of tic-tac-toe. There are several reasons for choosing this game:

- While simple to play, a tic-tac-toe strategy is far from simple to evolve. The strategy for the game has eight different behavioral aspects (elaborated below) and learning *all* of them *from scratch* is a complex task.
- The game rules include scenarios and anti-scenarios that demonstrate the *power and simplicity* of the paradigm in specifying reactive systems.
- The number of possible game traces is relatively small, allowing us to *formally prove* the quality of our solution.

We begin with a behavioral program for the game that specifies only the requirements of the game's rules. The requirements are as follows:

1. A cell can be marked only once.
2. X and O take alternate turns playing, with X starting.
3. The first player to obtain three marks in a line is the winner.
4. If no player wins and all nine squares are marked, a tie is declared.
5. The game ends upon a tie or a win.

We note that the context of Requirement 1 is a cell and the context of Requirement 3 is a line. To define the behavior of these requirements, we first need to define the system context, as presented in Listing 1. The code begins (lines 1–16) with a definition of the contextual data as a set of entities, each having a unique identifier (the first parameter of the `ctx.Entity` function), a type (second parameter), and data (third parameter). In addition, the context definition also includes two named functions—`Cell.Any` and `Line.Any` (lines 18–28). These functions are used to query the contextual data and return only the relevant entities. In this example, the queries always return the same answer since the context does not change throughout the lifetime of the game (e.g., cells and lines are not added/removed during the game). Nevertheless, COBP also supports dynamic changes to the context. To support such dynamics, the queries can be defined as functions rather than a predefined array of entities (see [1] for further information).

Listing 1. The context definition in COBPjs for the game of tic-tac-toe.

```
1  // Populate context with all cells
2  for (let i = 0; i < 3; i++)
3    for (let j = 0; j < 3; j++)
4      ctx.populate(ctx.Entity('cell('+i+','+j')', 'cell', {at: {i: i, j: j}}))
5
6  // Populate context with all lines
7  for (let i = 0; i < 3; i++) {
8    ctx.populate(ctx.Entity('row('+i+')', 'line',
9      { cells: [{ i: i, j: 0 }, { i: i, j: 1 }, { i: i, j: 2 }] }))
10   ctx.populate(ctx.Entity('col('+i+')', 'line',
11     { cells: [{ i: 0, j: i }, { i: 1, j: i }, { i: 2, j: i }] }))
12 }
13 ctx.populate(ctx.Entity('diag(0)', 'line',
14   { cells: [{ i: 0, j: 0 }, { i: 1, j: 1 }, { i: 2, j: 2 }] }))
15 ctx.populate(ctx.Entity('diag(1)', 'line',
16   { cells: [{ i: 2, j: 0 }, { i: 1, j: 1 }, { i: 0, j: 2 }] }))
17
18 // Register queries on the contextual data
19 ctx.registerQuery("Cell.Any", [
20     ctx.Entity('cell(0,0)', 'cell', {at: {i:0, j:0}}),
21     \dots
22     ctx.Entity('cell(2,2)', 'cell', {at: {i:2, j:2}})
23 ])
24 ctx.registerQuery("Line.Any", [
25     ctx.Entity('line(0)', 'line', {cells: [{i:0,j:0},{i:0,j:1},{i:0,j:2}]}),
26     \dots
27     ctx.Entity('line(7)', 'line', {cells: [{i:0,j:2},{i:1,j:1},{i:2,j:0}]})
28 ]
```

Now that we have the context definition, we can define the rule requirements, including the context-dependent requirements, as presented in Listing 2. We begin with a syntax explanation, followed by an an outline of the execution semantics (the full semantics are defined in [1]).

B-threads in COBP interact through events; thus, we begin with the definition of the possible events in the game (lines 1–6). X and O are functions that takes a `cell` entity and return an event with the name X or O, respectively, and the given cell as the event's data. A set of events can be defined using an array of events (e.g., line 14), or using a function that takes an event and returns true if the event is in the set (i.e., `AnyX` and `AnyO` in lines 9–10). Another way to define event sets is using `bp.not(e)` and `bp.all` functions (i.e., lines 39 and 53). The first returns true for events that are not e, and the latter for all events.

Listing 2. Tic-tac-toe rules, specified in COBPjs.

```
1  // Define events:
2  const X = cell => Event('X',cell)
3  const O = cell => Event('O',cell)
4  const XWin = Event('XWin')
5  const OWin = Event('OWin')
6  const Tie = Event('Tie')
7
8  // Define event sets
9  const AnyX = EventSet('AnyX', event => event.name === 'X')
10 const AnyO = EventSet('AnyO', event => event.name === 'O')
11
12 // Req. 1: A cell can be marked only once.
13 bthread("Do not mark cell twice", "Cell.Any", function(cell) {
14   sync({waitFor: [X(cell.at), O(cell.at)]})
15   sync({  block: [X(cell.at), O(cell.at)]})
16 })
17
18 // Req. 2: X and O play in turns; X starts.
19 bthread("Enforce turns", function () {
20   while (true) {
21     sync({waitFor: AnyX, block: AnyO})
22     sync({waitFor: AnyO, block: AnyX})
23   }
24 })
25
26 // Req. 3: The first player to get three marks in a line is the winner.
27 bthread("Detect X win", "Line.Any", function (l) {
28   sync({waitFor: [X(l.cells[0]),X(l.cells[1]),X(l.cells[2])]})
29   sync({waitFor: [X(l.cells[0]),X(l.cells[1]),X(l.cells[2])]})
30   sync({waitFor: [X(l.cells[0]),X(l.cells[1]),X(l.cells[2])]})
31   sync({request: XWin, block: bp.not(XWin)})
32 });
33
34 // Req. 3: The first player to get three marks in a line is the winner.
35 bthread("Detect O win", "Line.Any", function (l) {
36   sync({waitFor: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]})
37   sync({waitFor: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]})
38   sync({waitFor: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]})
39   sync({request: OWin, block: bp.not(OWin)})
40 })
41
42 // Req. 4: If no player won and all nine squares are marked, declare a tie.
43 bthread("Detect tie", function () {
44   for(let i = 0; i < 9; i++)
45     sync({waitFor: [AnyX, AnyO]})
```

```
46    sync(
47      {request: Tie, waitFor: [XWin, OWin], block: bp.not([XWin, OWin, Tie])})
48  })
49
50  // Req. 5: The game ends upon a tie or a win.
51  bthread("End game", function () {
52    sync({waitFor: [XWin, OWin, Tie]})
53    sync({  block: bp.all()})
54  })
```

The first b-thread (lines 1–5) specifies the first requirement: a cell can be marked only once. The context of the requirement is a cell, therefore the b-thread is bound to the `Cell.Any` query. As a result, for each answer to this query (i.e., for each `cell` entity), the execution mechanism of COBPjs will spawn and execute a live-copy of this behavior with the specific answer (i.e., `cell`) given as a parameter. The behavior itself (lines 3–4) waits for X or O events with this `cell` and then blocks any attempt to place another X or O there. The second b-thread (lines 7–13) defines the "turns" requirement, repeatedly blocking O while X is playing, and vice versa. This b-thread is not bound to any query since the requirement is not context-dependent. Similarly, the following b-threads handle winning and tie situations.

We note that the code in Listing 2 only imposes a correct behavior according to the game rules. When no one requests X or O events the game does not start. To start, we can create random X and O players by binding an additional b-thread to the `Cell.Any` query that has a single line: `sync({request: [X(cell), O(cell)]})`.

Although b-threads are independent from each other, they still need a way to synchronize with other b-threads. In COBP, whenever b-threads reach a sync statement, they submit their statement—what they *request*, *waitFor*, or *block*—to an application-agnostic central event arbiter, and halt. The arbiter selects an event that was requested but not blocked, and resumes all b-threads that are waiting for, or requested, the selected event, until the next synchronization statement. This simple protocol generates a cohesive behavior that is consistent with all b-threads. Notably, the protocol does not dictate the order in which actions are performed (e.g., given the random X and O player, the first X may be placed at any of the nine cells). To create a highly competent player for the game, which does not lose, and wins when possible, we need additional *strategy* b-threads that will prioritize the possible actions.

3.1. Hand-Crafted Strategy B-Threads for Tic-Tac-Toe

In the sections ahead we will use GP for evolving strategy b-threads for tic-tac-toe. Herein, we present a hand-crafted version for these b-threads, toward which end we begin with a description of Newell and Simon's strategy [25], consisting of eight rules with diminishing priorities, meaning that at each turn the players should play the first available rule:

1. **Win:** If the player has two tokens in a line, then the player should place the last cell.
2. **Block:** If the opponent has two tokens in a line, then the player should place the last cell.
3. **Fork:** Cause a scenario where the player has two ways to win (two non-blocked lines with two tokens).
4. **Blocking an opponent's fork:** If there is only one possible fork for the opponent, the player should block it. Otherwise, the player should block all forks in any way that simultaneously allows them to make two-in-a-row. Otherwise, the player should make a two-in-a-row to force the opponent into defending, as long as it does not result in their producing a fork.
5. **Center:** A player marks the center.
6. **Opposite corner:** If the opponent is in the corner, the player plays the opposite corner.
7. **Empty corner:** The player plays in a corner square.

8. **Empty side:** The player plays in a middle square on any of the four sides.

Listing 3. Strategy b-threads for an O player.

```
// Strategy 1: Win
ctx.bthread("Add third O", "Line.Any", function (l) {
  sync({waitFor: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]})
  sync({waitFor: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]})
  sync({request: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]}, 8)
})

// Strategy 2: Block
ctx.bthread("Prevent third X", "Line.Any", function (l) {
  sync({waitFor: [X(l.cells[0]),X(l.cells[1]),X(l.cells[2])]})
  sync({waitFor: [X(l.cells[0]),X(l.cells[1]),X(l.cells[2])]})
  sync({request: [O(l.cells[0]),O(l.cells[1]),O(l.cells[2])]}, 7)
})

// Strategy 3: Create a fork opportunity
ctx.bthread("Cause fork", 'Fork.Any', function (f) {
  sync({ waitFor: [O(f.cells[0]), O(f.cells[1])] })
  sync({ waitFor: [O(f.cells[0]), O(f.cells[1])] })
  sync({ request: O(f.cells[2]) }, 6)
})

// Strategy 4: Block fork
ctx.bthread("Block fork", 'Fork.Any', function (f) {
  sync({ waitFor: [X(f.cells[0]), X(f.cells[1])] })
  sync({ waitFor: [X(f.cells[0]), X(f.cells[1])] })
  sync({ request: O(f.cells[2]) }, 5)
})

// Strategy 5: Center
ctx.bthread("Center", "Cell.Center", function (c) {
  sync({request: O(c.at)}, 4)
})

// Strategy 6: Opposite corner
ctx.bthread("Corner", "Cell.Corner", function (c) {
  sync({request: O(opposite(c.at))}, 3)
})

// Strategy 7: Empty corner
ctx.bthread("Corner", "Cell.Corner", function (c) {
  sync({request: O(c.at)}, 2)
})

// Strategy 8: Sides
ctx.bthread("Sides", "Cell.Sides", function (c) {
  sync({request: O(c.at)}, 1)
})
```

Listing 3 presents the code for a highly competent O player. The numbers at the end of the sync statements represent the priority of the events. The selected event will have the highest priority. Listing 3 also introduces new queries: `Fork.Any`, `Cell.Center`, `Cell.Sides`, and `Cell.Corners`. We do not elaborate the definition of these queries, though evolution will find them, as shown below. In Section 5 we compare the evolved strategy b-threads to these hand-crafted ones.

We note that most rules are well defined and their implementation is thus straightforward. Nevertheless, the two fork rules (3 and 4) are unclear and the implementation is thus not aligned to them. While in the next section we will evolve the entire strategy b-threads, we note that in complex domains users may add hand-crafted strategy b-threads to the base individuals' programs, and learn others.

3.2. COBP Characteristics

The code in the above listings reveals some of the unique characteristics of behavioral programs.

First, we observe the repetitive structure of many b-threads, with each having roughly the same body. This structure allows for a simple representation of a program as an abstract syntax tree (AST).

Second, because each b-thread defines a single aspect of the system's behavior, the behavioral programs are self explanatory. As we will show in Section 5, the evolved programs share this same characteristic, resulting in simple and explainable programs generated by evolution.

Third, COBP's execution mechanism is based on the mathematically rigorous nature of the COBP semantics. These semantics afford the application of formal methods and verification algorithms to verify the correctness of a program, i.e., that it behaves correctly for any input. Here, we do not use this property for evaluating the individuals during evolution, though we will use it for validating the quality of the evolved individuals.

4. Method

In this section we describe our experimental setup for evolving behavioral programs. Aside from the fitness function and some parameters, the approach is domain-independent and may be applied to any behavioral program. The algorithm requires baseline b-threads (here, the entities and the rule b-threads from Listings 1 and 2), a fitness function, and terminals for the grammar (elaborated below). The algorithm outputs a set of b-threads that together with the baseline b-thread construct a complete, modular behavioral program. In our case, we want the algorithm to generate the strategy b-threads that constitute, along with the baseline b-threads, a program of a highly competent O player.

4.1. Grammar

We use grammar-based GP to evolve the program. The grammar is mostly domain-independent (except for the terminal set) and utilizes the repetitive nature of a behavioral program.

The structure of a general behavioral program is illustrated in Figure 1. The program consists of contextual data, represented as a set of entities, queries on these entities, and context-dependent behaviors, each bound to a single query. For example, in Listing 2, there are two types of entities—`cell` and `line`. The queries are `Cell.Any` and `Line.Any`, and each of the context-dependent b-threads is bound to one of these queries. We recall that for generating strategy b-threads, additional queries and behaviors are required (`Cell.Sides`, `Cell.Corners`, etc.).

In the following experiments we want evolution to find both the b-threads and all context queries, including `Cell.Any` and `Line.Any`. We note that a context query is essentially defined as a set of entities. For example, `Cell.Any` is a set of nine sets, each containing a single `cell` entity. Similarly, `Line.Any` is a set of eight sets (three rows, three columns, and two diagonals), each containing three `cell` entities in a line. Thus, the queries in our grammar are parameterized by the size of the internal sets.

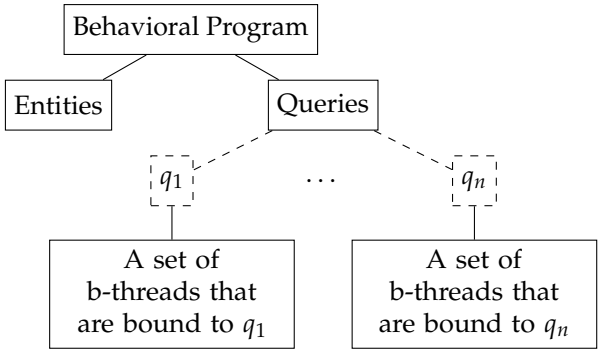

Figure 1. Structure of a general COBP program.

The grammar is presented in Listing 4; it is divided into two parts: a general grammar for any COBP programs, and specific terminals for tic-tac-toe. The general grammar can be generalized even more, e.g., by allowing multiple requests or by allowing *waitFor* and *request* in the same synchronization statement. Nevertheless, this grammar is general enough for our task and, from our experience with COBP, it is popular in programs as well. The specific grammar for tic-tac-toe is also general, in the sense that it only defines the entities and events that were defined by the rule b-threads. In addition, this part of the grammar configures the maximal possible priority and the maximal number of queries, behaviors per query, and entities. These values act as estimations that help evolution by placing boundaries on the size of the individuals.

Listing 4. Grammar for evolving strategy b-threads for tic-tac-toe.

```
// A general grammar for COBP programs:
BProgram → Query^{1...MaxQueries}
Query → i, Entities_i, Behavior_i^{1...MaxBehaviors}
Entities_i → Entity_i^{1...MaxEntities}
Behavior_i → Wait_i^{0...2}, Request_i
Wait_i → "sync({waitFor: [" + Event_i^{1...i} + "]})"
Request_i → "sync({request: [" + Event_i^{1...i} + "]}, " + Priority + ")"
Priority → 0 | 1 | ··· | MaxPriority

// Terminals for tic-tac-toe:
MaxQueries = 10
MaxBehaviors = 5
MaxEntities = 9
MaxPriority = 10
Entity_i → Cell^i
Cell → Position, Position
Position → 0 | 1 | 2
Event_i → XEvent_i | OEvent_i
XEvent_i → "X(" + Entities_i[⟨1 ... i⟩] + ")"
OEvent_i → "O(" + Entities_i[⟨1 ... i⟩] + ")"
```

4.2. Fitness Function

To evaluate an individual program we let it play 100 games as follows:

- A total of 25 games vs. a player that does not lose, where the generated program plays first.
- A total of 25 games vs. a player that does not lose, where the generated program plays second.
- A total of 25 games vs. a random player, where the generated program plays first.
- A total of 25 games vs. a random player, where the generated program plays second.

The player alternated between playing first and playing second in order to prevent evolution from succumbing to a bias, since the first player has an advantage because it has the opportunity to make more moves in a single game. Further, we alternated between players that do not lose and and random players because playing only against the first type of players did not allow the generated programs to learn how to win games. In addition, playing only against random players did not allow the generated programs to learn how to block the opponent from winning.

Given these 100 games, the fitness function was

$$f = 50 \cdot \phi\left(\frac{wins}{missed\_wins + wins}\right) + 50 \cdot \phi\left(\frac{blocks}{missed\_blocks + blocks}\right) - deadlocks$$

$$\phi(x) = \frac{1 - e^{-x}}{1 - e^{-1}}$$

$deadlocks$ = Number of times the individual did not request any action.

$wins/block$ = Number of wins/blocking opponent from winning.

$missed\_wins/missed\_blocks$ = Number of times that the individual missed an opportunity to win/block the opponent from winning.

The first two parts take into account how decisively the generated program won when given the opportunity, and how decisively it blocked the enemy from winning when possible. Additionally, to complete the game (i.e., reach a tie or a win), the individual must request for at least one event that is not blocked, whenever it is the turn of the O player. To address this, we added a penalty part to the fitness that reduces one point for each non-completed game. Finally, to push evolution further, we used fitness pressure (denoted by ϕ) for each of the two parts. Thus, the fitness values were in the range of $[-100, 100]$.

4.3. Genetic Operators

Our design of the genetic operators is based on two observations. First, generally speaking, b-threads may be correlated in several ways. For example, the strategy b-threads in Listing 3 have different priorities, thus defining an order for applying each strategy. As another example, consider the "turns" b-thread to be missing, in which case the program will not be able to generate correct game traces.

The second observation relates to the method for selecting the node for performing crossover or mutation. It makes no sense to perform crossover in the middle of a behavior, since it is a single logical unit with a high correlation between its different lines. Similarly, replacing an entire behavior will most likely result in a behavior that does not work well with the other behaviors. Therefore, uninformed, domain-independent genetic operators (e.g., single point crossover) yield poor results.

To keep the correlation between the different behaviors, we set the probabilities for applying mutation and crossover operators to be 70% and 5%, respectively. Thus, evolution was driven mostly by exploitation, rather then exploration.

For selecting the nodes for the operators, we grouped the nodes into four types, and defined a different probability for selecting each type, depending on the operator. The operators first selected a group type, and then uniformly selected a node in the group. The node types and the probabilities for selecting them are summarized in Table 1. These probabilities were reached through empirical testing.

Table 1. The probabilities for selecting nodes for mutation and crossover, grouped by node types.

What to Replace	Probability
Entire Query	0.05
Entire Behavior$_i$	0.15
A node under an Entities$_i$	0.4
A node under a Behavior$_i$	0.4

After selecting the node, the mutation operator replaces its sub-tree by growing another sub-tree instead, and the crossover operator replaces the sub-tree with a sub-tree from the second individual. Since the grammar is strongly typed, the crossover operator has additional constraints for selecting the node in the second individual.

4.4. "Smart" Test-Driven Genetic Operators

The first genetic operators served as a baseline for future enhancements and for presenting the capabilities of behavioral programming as a tool for evolving source code. In our second experiment, we proposed to evaluate tree nodes according to their contribution to the entire program—whether the contribution was "good" or "bad". Then, we used these evaluations for developing smart genetic operators, by selecting the nodes according to their evaluation values.

Our smart, test-driven mutation and crossover operators are rooted in the field of *fault diagnosis* [26]. Fault diagnosis is the process of determining the type, size, location, and time of detection of a fault. We considered "bad moves" the generated program performed as faults in the code. We then applied diagnosis techniques to find "faulty" nodes in our program. Additionally, we used the same strategy to find "good" components in the code. For each win, win miss, block, and block miss, we noted the last node that was involved, and calculated each component's score accordingly—wins and blocks increased the score, while win misses and block misses decreased the score. For example, if the last event was `OWin`, then the win counter will be increased for the following nodes: the *Behavior*$_i$ node that represents the b-thread that requested the last O event, the *Entity*$_i$ node of the behavior (i.e., the context of the b-thread), and the *Query* node that contains the entity and the behavior. We also added a small bonus for every move the component was involved in, in order to discourage "timid" components, which did nothing. The smart operators still selected the node type using the same probabilities, but the node inside the group was selected according to the component score, rather than uniformly, with contrastive formulas for mutation and recombination:

$$recombination\_score(node) = 2 \cdot wins - 2 \cdot win_v + blocks - block_v + 0.25 \cdot requests$$

$$mutation\_score(node) = -recombination\_score(node)$$

The reason for this change is that in recombination we provide a higher chance for exchanging good nodes among individuals, and in mutation we provide a higher chance for replacing bad nodes. To select the actual node we used the same proportionate-selection mechanism of roulette wheel. Thus, the node score acts as an estimation for the node quality, but other nodes still stand a chance of being selected. For the same reason, a score of zero or less was reset to one.

5. Results

We tested our approach both with the basic operators and with the smart ones—and compared the results. Further, we also examined the playing capabilities of the evolved programs. The evolutionary hyperparameters are summarized in Table 2. Notably, on a computer with an Intel i7 CPU, running 100 random games sequentially takes roughly three seconds and 500 MB. The memory footprint can be reduced to 4 MB at the expense of an additional second, by running the garbage collector after each run. Thus, while we

executed 3 M random games at each generation, complete runs with no parallelization took less than four hours.

Over several experiments the basic operators managed to achieve an individual with maximum fitness score of 94 out of 100; the best individuals from these experiments consistently appeared as early as generation 250. The smart operators managed to generate a better best-performing individual, with a fitness value of 97.5. It is worth noting that this specific individual missed a block only seven times, throughout all 100 games, and it did not miss a single win opportunity. Individuals with fitness above 95 consistently appeared as early as generation 150, nearly 100 generations faster than the basic mutation operators.

To compare the performance of the operators, we selected the best two individuals at generations 50, 150, and 300. We then applied the operators 1000 times on these individuals, cloning the original individuals before applying the operators. Table 3 shows the percentage each operator improved the fitness of the individual it was performed on, based on these 1000 runs. As results show, the smart operators were far more effective than the basic operators. The basic mutation performed worse almost always. The smart crossover outperformed the basic crossover in both parameters, except for generation 150, probably due to the fact that runs with the smart crossover had already reached a high fitness by this time.

Table 2. Evolutionary hyperparameters.

Representation	Grammar-based GP
Mutation	Grow sub-tree †
Recombination	Exchange of sub-trees †
Mutation probability	0.7
Recombination probability	0.05
Parent selection	Tournament with $k = 3$
Survivor selection	Generational replacement
Population size	100
Number of generations	300

† The selection of the node to mutate/exchange is described in Section 4.3.

Table 3. Results: Genetic operators applied 1000 times to best two individuals at generations 50, 150, and 300. %Improved: Percentage of 1000 times the operator improved fitness. %Improvement: Average improvement of those that were improved by operator, i.e., of *%Improved*.

	Generation	%Improved	%Improvement
Basic crossover	50	8.2%	0.57%
	150	37.0%	0.97%
	300	10.7%	0.41%
Basic mutation	50	1.9%	1.98%
	150	0.3%	2.45%
	300	0.0%	0.00%
Smart crossover	50	25.7%	1.03%
	150	28.0%	0.85%
	300	38.2%	0.99%
Smart mutation	50	29.4%	1.13%
	150	62.8%	1.83%
	300	10.7%	0.54%

5.1. Validation of the Evolved Players

To validate the correctness of our (O-playing) individuals, i.e., that they behave correctly for any opponent X player, we used the verification mechanism of COBPjs. Using this tool we mapped the state space of a game between two random players, thus generating all possible states of tic-tac-toe. Next, we traversed the graph and generated all possible

255,168 game traces (excluding symmetry). The first player wins 131,184 of these, the second player wins 77,904 traces, and the remaining 46,080 are a tie. To validate our individuals, we performed the same computation for our evolved players and compared the generated traces. Table 4 presents the results for our best individual of all runs. This individual reduces the number of times X wins to less than 1% and the number of missed possible wins to less than 0.01%. The results were similar no matter who started.

Table 4. Validation of top, evolved individual: Number of times X wins compared to all possible ways that X may win, and number of times O missed a possible win compared to the number of possible ways for O to win.

Starting Player	Times X Wins	Winning Misses
Exhaustive	0.03%	0%
Evolved	0.8%	0.01%

5.2. Individual Representation

Listing 5 presents the context definition and the strategy b-threads of the best individual of all runs. The individual is different from the handcrafted strategy b-threads (Listing 3), both in context and in behaviors; nevertheless, it ultimately exhibits similar behavior. Upon inspection of the evolved solution we found that part of the code is straightforward to understand. For example, the second b-thread, `bt_0_1`, waits for one X and then places an O on the same line. The context of this b-thread, `ctx1`, defines a list of pairs of cells, each pair consisting of two cells on the same line. This behavior resembles the behavior of our handcrafted "Prevent third X" b-thread (second b-thread of Listing 3), which waits for two Xs in a line before placing an O in the third cell. Some behaviors are covered by more than one behavior. For example, part of the possible forks are covered by a combination of `bt_0_1` and `bt_0_0`.

Listing 5. Best evolved individual.

```
ctx.registerQuery("ctx0", \dots)
ctx.registerQuery("ctx1", [
  ctx.Entity("ctx1_0", "ctx1", { cells: [{x: 1, y: 0}, {x: 0, y: 0}]}),
  ctx.Entity("ctx1_1", "ctx1", { cells: [{x: 0, y: 2}, {x: 2, y: 2}]})
  ctx.Entity("ctx1_2", "ctx1", { cells: [{x: 2, y: 2}, {x: 0, y: 2}]}),
  ctx.Entity("ctx1_3", "ctx1", { cells: [{x: 1, y: 2}, {x: 1, y: 0}]}),
  ctx.Entity("ctx1_4", "ctx1", { cells: [{x: 0, y: 2}, {x: 2, y: 0}]}),
  \dots
])
\dots
ctx.registerQuery("ctx3", \dots)

ctx.bthread('bt_0_0', 'ctx0', function (c) {
  sync({ waitFor: [O(c.cells[0]), X(c.cells[2]), X(c.cells[1])] })
  sync({ waitFor: [O(c.cells[3]), X(c.cells[4]), X(c.cells[5])] })
  sync({ request: O(c.cells[6]) }, 2)
})

ctx.bthread('bt_1_0', 'ctx1', function (c) {
  sync({ waitFor: X(c.cells[0]) })
  sync({ request: O(c.cells[1]) }, 9)
})

ctx.bthread('bt_2_0', 'ctx2', function (c) {
  sync({ request: O(c.cells[0]) }, 7)
})

```

```
ctx.bthread('bt_3_0', 'ctx3', function (c) {
  sync({ request: O(c.cells[1]) }, 10)
})

ctx.bthread('bt_3_1', 'ctx3', function (c) {
  sync({ waitFor: X(c.cells[2]) })
  sync({ waitFor: O(c.cells[3]) })
  sync({ request: O(c.cells[0]) }, 2)
})

ctx.bthread('bt_3_2', 'ctx3', function (c) {
  sync({ waitFor: [X(c.cells[1]), O(c.cells[1]), X(c.cells[2])] })
  sync({ request: [O(c.cells[1]), O(c.cells[2])] }, 9)
})
```

6. Concluding Remarks

We presented the use of genetic programming to evolve behavioral programs from scratch, performing an in-depth analysis, using as a case study the game of tic-tac-toe. We demonstrated the advantages of context-oriented behavioral programming (COBP) in general, and in particular as they pertain to amenability to evolution. To this end, we presented novel, domain-independent, grammar and genetic operators for behavioral programs. Using these grammar and genetic operators, we evolved highly competent players for tic-tac-toe.

To validate our findings, we utilized the mathematical characteristics of COBP to verify the correctness of evolved programs. Compared to all possible ways that a player can lose or miss an opportunity to win, our best evolved player reduced these cases to less than one percent.

Another strength of our approach is the ability to understand the generated programs. Due to the simple and repetitive structure of behavioral programs, we were able to observe that evolved individuals ultimately exhibited similar behavior to the handcrafted program.

While tic-tac-toe may seem like a toy problem, evolution of this magnitude has not been performed before for this domain. Evolving a complete program from scratch is a complex task that was made possible by the unique characteristics of COBP.

Given our study we believe that using GP, and evolution in general, to evolve behavioral programs is a viable research pursuit, well worth undertaking.

During our work, we identified some possible shortfalls of our approach that we wish to examine in future research:

- *Restricted context grammar:* The contexts (i.e., cell, line, fork) in tic-tac-toe are constant, in the sense that they do not change over time—cells are not dynamically added or removed. Dynamic contexts are important for many domains. In chess, for example, the "check" context needs to be activated whenever an opponent piece is threatening the king. While COBP supports dynamic context, the grammar defined here limits this ability. To support it, we will need to extend the grammar to allow more complex entities and queries. Relaxing the constraint of static context should be approached carefully, since it may dramatically increase the search space.
- *Evolving more-complex behavior:* In terms of computing resources (CPU and RAM), our approach is very efficient and scalable. Nevertheless, it may be less successful on larger problems with more-complex behaviors, such as chess or connect-four. For such problems, we will likely employ a hybrid approach, wherein programmers provide some behaviors and evolution is set to find others; programmers then refine evolution's solutions by adding handcrafted behaviors, after which evolution is launched—and so on. Such a methodology may constitute a natural relationship between programming and optimization.

- Finally, in this research we used game simulations for evaluating our individuals. Another interesting direction is to use a data-driven approach for evaluating individuals, without actually running them. For example, we may learn the rule b-threads, given a similar grammar and a labeled set of both possible and impossible game traces. The evaluation can be carried out by checking, on a training set, if the evolved program does not generate impossible traces and that all possible traces are indeed possible. This approach presents a complication because the grammar must include the block idiom as well, which forbids events from happening and may lead most games to deadlocks.

Author Contributions: R.P. designed and performed the experiments, and wrote the paper; M.S. did algorithm design and wrote the paper; A.E. designed the entire setup, did algorithm design, and wrote the paper. All authors have read and agreed to the published version of the manuscript.

Funding: This research was partially supported by "the Israeli Council for Higher Education (CHE) via Data Science Research Center, Ben-Gurion University of the Negev, Israel" and by the Israeli Smart Transportation Research Center (ISTRC).

Conflicts of Interest: The authors declare no conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

COBP	Context-oriented behavioral programming
GP	Genetic programming

References

1. Elyasaf, A. Context-Oriented Behavioral Programming. *Inf. Softw. Technol.* **2021**, *133*, 106504.
2. Cramer, N.L. A representation for the adaptive generation of simple sequential programs. In Proceedings of the International Conference on Genetic Algorithms and the Applications, Pittsburgh, PA, USA, 24–26 July 1985; pp. 183–187.
3. Koza, J.R. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*; The MIT Press: Cambridge, MA, USA, 1992.
4. Orlov, M.; Sipper, M. Genetic programming in the wild: Evolving unrestricted bytecode. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Montreal, QC, Canada, 8–12 July 2009; pp. 1043–1050.
5. Orlov, M.; Sipper, M. Flight of the FINCH through the Java wilderness. *IEEE Trans. Evol. Comput.* **2011**, *15*, 166–182.
6. Dijkstra, E.W. On the cruelty of really teaching computing science. *Commun. ACM* **1989**, *32*, 1398–1404.
7. Petke, J.; Haraldsson, S.O.; Harman, M.; Langdon, W.B.; White, D.R.; Woodward, J.R. Genetic improvement of software: A comprehensive survey. *IEEE Trans. Evol. Comput.* **2017**, *22*, 415–432.
8. Gabel, M.; Su, Z. A study of the uniqueness of source code. In Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering, Santa Fe, NM, USA, 7–11 November 2010; pp. 147–156.
9. Barr, E.T.; Brun, Y.; Devanbu, P.; Harman, M.; Sarro, F. The plastic surgery hypothesis. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, Hong Kong, China, 16–21 November 2014; pp. 306–317.
10. Forrest, S.; Nguyen, T.; Weimer, W.; Le Goues, C. A genetic programming approach to automated software repair. In Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, Montreal, QC, Canada, 8–12 July 2009; pp. 947–954.
11. Weimer, W.; Nguyen, T.; Le Goues, C.; Forrest, S. Automatically finding patches using genetic programming. In Proceedings of the 2009 IEEE 31st International Conference on Software Engineering, Vancouver, BC, Canada, 16–24 May 2009; pp. 364–374.
12. Ackling, T.; Alexander, B.; Grunert, I. Evolving patches for software repair. In Proceedings of the 13th Annual Conference on Genetic and Evolutionary Computation, Dublin, Ireland, 12–16 July 2011; pp. 1427–1434.
13. Le Goues, C.; Dewey-Vogt, M.; Forrest, S.; Weimer, W. A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each. In Proceedings of the 2012 34th International Conference on Software Engineering (ICSE), Zurich, Switzerland, 2–9 June 2012; pp. 3–13.
14. Schulte, E.; DiLorenzo, J.; Weimer, W.; Forrest, S. Automated repair of binary and assembly programs for cooperating embedded devices. *ACM SIGARCH Comput. Archit. News* **2013**, *41*, 317–328.
15. Schulte, E.; Forrest, S.; Weimer, W. Automated program repair through the evolution of assembly code. In Proceedings of the IEEE/ACM international conference on Automated Software Engineering, Antwerp, Belgium, 20–24 September 2010; pp. 313–316.
16. Harel, D.; Marron, A.; Weiss, G. Programming coordinated behavior in java. In Proceedings of the European Conference on Object-Oriented Programming, Maribor, Slovenia, 21–25 June 2010; Springer: Berlin/Heidelberg, Germany, 2010; pp. 250–274.
17. Shimony, B.; Nikolaidis, I. On coordination tools in the PicOS tuples system. In Proceedings of the 2nd Workshop on Software Engineering for Sensor Network Applications, Honolulu, HI, USA, 22 May 2011; pp. 19–24.

18. Elyasaf, A.; Marron, A.; Sturm, A.; Weiss, G. A Context-Based Behavioral Language for IoT. In *CEUR Workshop Proceedings*; Regina Hebig and Thorsten Berger, Ed.; CEUR-WS.org: Copenhagen, Denmark, 2018; Volume 2245, pp. 485–494.
19. Walsh, P.; Ryan, C. Automatic Conversion of Programs from Serial to Parallel using Genetic Programming-The Paragen System. Parallel Computing: State-of-the-Art and Perspectives, Proceedings of the conference ParCo 1995, Gent, Belgium, September 1995; D'Hollander, E.H.; Joubert, G.R.; Peters, F.J.; Trystram, D., Eds. Elsevier, 1995, Volume 11, *Advances in Parallel Computing*, pp. 415–422.
20. Walsh, P.; Ryan, C. Paragen: A novel technique for the autoparallelisation of sequential programs using gp. In Proceedings of the 1st Annual Conference on Genetic Programming, Stanford, CA, USA, 28–31 July 1996; pp. 406–409.
21. Barr, E.T.; Harman, M.; Jia, Y.; Marginean, A.; Petke, J. Automated software transplantation. In Proceedings of the 2015 International Symposium on Software Testing and Analysis, Baltimore, MD, USA, 13–17 July 2015; pp. 257–269.
22. Marginean, A.; Barr, E.T.; Harman, M.; Jia, Y. Automated transplantation of call graph and layout features into Kate. In Proceedings of the International Symposium on Search Based Software Engineering, Bergamo, Italy, 5–7 September 2015; Springer: Berlin/Heidelberg, Germany, 2015; pp. 262–268.
23. Le Goues, C. Automatic Program Repair Using Genetic Programming. Ph.D. Thesis, School of Engineering and Applied Science, University of Virginia, Charlottesville, VA, USA, 2013.
24. Arcuri, A.; Yao, X. A novel co-evolutionary approach to automatic software bug fixing. In Proceedings of the 2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence), Hong Kong, China, 1–6 June 2008; pp. 162–168.
25. Crowley, K.; Siegler, R.S. Flexible strategy use in young children's tic-tac-toe. *Cogn. Sci.* **1993**, *17*, 531–561.
26. Simani, S.; Fantuzzi, C.; Patton, R.J. Model-based fault diagnosis techniques. In *Model-based Fault Diagnosis in Dynamic Systems Using Identification Techniques*; Springer: Berlin/Heidelberg, Germany, 2003; pp. 19–60.