Review

Not peer-reviewed version

Review and Mapping of Search-Based Approaches for Program Synthesis

Takfarinas Saber [*] and Ning Tao

Posted Date: 24 March 2025

doi: 10.20944/preprints202503.1722.v1

Keywords: Program Synthesis; Automated Programming; Search-Based Algorithm; Heuristic; Metaheuristic; Survey

Article

Review and Mapping of Search-Based Approaches for Program Synthesis

Takfarinas Saber [1,*] and Ning Tao [2]

[1] Lero, Science Foundation Ireland Research Centre for Software, School of Computer Science, University of Galway;
[2] School of Computer Science, University College Dublin
[*] Correspondence: takfarinas.saber@universityofgalway.ie
[‡] These authors contributed equally to this work.

Abstract: Context: Program synthesis tools reduce software development costs by generating programs that perform tasks depicted by some specifications. Various methodologies have emerged for program synthesis, among which search-based algorithms have shown promising results. However, the proliferation of search-based program synthesis tools utilising diverse search algorithms and input types and targeting various programming tasks can overwhelm users seeking the most suitable tool. **Objective:** This paper contributes to the ongoing discourse by presenting a comprehensive review of search-based approaches employed for program synthesis. We aim to offer an understanding of the guiding principles of current methodologies by mapping them to the required type of user intent, the type of search algorithm, and the representation of the search space. Furthermore, we aim to map the diverse search algorithms to the type of code generation tasks at which they have shown success, which would serve as a guideline for applying search-based approaches for program synthesis. **Method:** We conducted a literature review of 67 academic papers on search-based program synthesis. **Results:** Through analysis, we identified and categorised the main techniques with their trends. We have also mapped and shed light on patterns connecting the problem, the representation and the search algorithm type. **Conclusion:** Our study summarises the field of search-based program synthesis and provides an entry point to the acumen and expertise of the search-based community on program synthesis.

Keywords: program synthesis; automated programming; search-based algorithm; heuristic; meta-heuristic; survey

1. Introduction

In this digital era, programming skills are increasingly vital for many jobs. As only a fraction of the labour force hones such skills, governments and companies are putting in place costly and time-consuming upskilling programs. Automated code generation based on user intent, known as program synthesis, is inching closer to becoming a viable alternative for organisations or individuals that strive to enhance their efficiency at minimum efforts and costs.

Extensive research has previously been conducted on program synthesis as a means to facilitate the work of programmers by providing them with a range of tools and techniques to generate programming code based on their high-level intentions. Recent studies have proposed many ways to tackle program synthesis tasks, including Machine Learning (ML) techniques, Large Language Models (LLMs), and search-based techniques. These algorithms vary in programming languages and target problems, from simple tasks (e.g., symbolic regression [1–7], string manipulation [8–11], and binary transformation [12–14]), to more complex challenges (e.g., robot path-finding [6,15], algebraic calculations [14,16–21], and intricate real-world programming problems). Particularly, Saha et al. [22] proposed an algorithm to generate an ML pipeline using a corpus and a human-written pipeline. Poliansky et al. [23] utilised Genetic Programming (GP) and context-oriented behavioural programming (COBP) in the Tic-Tac-Toe game. Beltramelli introduced pix2code [24], which leverages Convolutional Neural Networks (CNNs) to generate web development interface code (HTML/CSS) from screenshots of the graphical user interface. The AlphaCode developer team harnessed large-scale sampling and transformer language models to address previously unsolved competitive programming challenges [25].

Several methods have been proposed for program synthesis, each with its strengths and weaknesses. For instance, ML techniques and LLMs rely on a large corpus of training data. While they have been shown to successfully generate human-like programs, they often struggle to generate the correct code due to ambiguity in task specifications and complex programming syntax (e.g., generating known buggy code [26]). Moreover, there is a legitimate lack of trust in the generated code as it comes with documented risks (e.g., generating code with known and risky vulnerabilities [27,28]). The potential for LLM-generated bad code represents a large and growing risk to software and its stakeholders in general. Furthermore, their generative nature associated with the small size of their prompts limits their ability to fix erroneous codes with iterative LLM prompting [29,30].

Search-Based Program Synthesis (SBPS) algorithms, on the other hand, require less training data and can be constrained to search for programs that obey predefined specifications, such as grammar files [18,19]. However, SBPS approaches often struggle to scale to complex tasks and often yield programs with poor readability and clarity. To deal with the shortcomings of each field, recent works are attempting to combine LLM-based with SBPS techniques [18,31,32].

In this work, we reviewed the literature that utilizes Search-Based algorithms for program synthesis. After identifying and selecting the relevant works, we classify them along various dimensions, including search algorithm type, types of user intent, and representation type of the search space. Furthermore, we analyze the datasets and the type of target tasks addressed by each approach–shedding light on their efficiency and guiding their utilization by non-domain experts. Furthermore, we analyze the connections between these dimensions to gain insight into any patterns. Lastly, we concluded by summarizing the observed challenges in the SBPS field.

This study is of high and timely importance. As we witness a new research community (i.e., generative AI) taking on the automated code generation challenge and researchers from the search-based community investigating the best way to embed LLMs in their search techniques (e.g., [33–35]), it is important, now more than ever, to provide an entry point to the acumen and expertise of the search-based community to help build synergy for collaboration.

The rest of the paper is structured as follows: Section 2 provides an overview of the background and work related to our study. Section 3 offers a comprehensive exposition of the methodology and framework employed in conducting our review. Subsequently, in Section 4, we embark on a detailed examination and discussion of the review results, facilitating a comparative analysis of the identified research. In Section 5, we outline the challenges faced by the SBPS field. Finally, in Section 6, we conclude our work with the key takeaways and implications of our study while also delineating directions for prospective future research endeavours.

2. Background and Related Work

In this section, we describe the background of our study in two parts: Program Synthesis in Automated Code Generation and Search-Based Program Synthesis. We also discuss existing studies related to program synthesis.

2.1. Program Synthesis in Automated Code Generation

Automated code generation is a programming methodology that creates computer programs based on user specifications. This approach frequently employs artificial intelligence (AI) techniques to enhance productivity and reduce costs by replacing or assisting human developers in the manual composition of code segments. Its application spans various domains within computer science and software engineering, such as the development of Integrated Development Environments (IDEs), automatic program repair, and program synthesis.

Tools employed in automated code generation can be categorised into three primary groups based on their resulting code: code template generation, code repair, and program synthesis. Code template generation is commonly utilised within IDEs to assist users with pre-defined code templates, thereby simplifying the programming process. Code repair aims to finalise or augment provided code snippets. Program synthesis is a popular branch of automated code generation research aiming to produce code based on high-level user specifications. This category endeavours to formulate either complete programs or syntactically correct code segments. The nature of user specifications varies across different techniques, with search-based algorithms (e.g., evolutionary approaches) often leveraging input-output test cases and Language Models (LLMs) generating code based on prompts expressed in natural language.

2.2. Search-Based Program Synthesis

SBPS (subfield of, Search-Based Software Engineering), is a software development technique that uses search algorithms and optimisation approaches to address programming tasks in creating and maintaining software systems. Various search techniques can be applied to SBPS, while meta-heuristic search algorithms are the most popular. Meta-heuristic search algorithms include biological approaches like genetic algorithms, simulated annealing and swarm intelligence. It also includes non-biological algorithms such as Local and Tabu searches. This field focuses on analysing and enhancing vast solution spaces related to software engineering problems. SBPS treats software engineering problems as optimisation challenges that aim to find optimal or near-optimal solutions within the solution space. This approach enables the automation of certain software engineering tasks, enhances the efficiency of problem-solving processes, and provides insights into the trade-offs inherent in complex decision spaces. Applications of SBPS span a wide range of software engineering activities, including test case generation, code optimisation, requirements engineering, software maintenance, and project management.

2.3. Reviews Related to Search-Based Program Synthesis

Given its age and importance, automated code generation has garnered numerous surveys. Batouta et al. [36] performed a tertiary and systematic mapping review of research in automation and code generation. Therefore, we do not aim to discuss all related reviews. Instead, we attempt to highlight those that are the closest to ours.

Sobania et al. [37] surveyed the evolutionary program synthesis approaches from 2015 to 2020 that were specifically evaluated specifically on the PSB1 [38] benchmark dataset which enabled them to compare the performance of the different algorithms.

Olmo et al. [39] surveyed swarm-based automatic programming studies: i.e., studies at the intersection between program synthesis and the use of Swarm Intelligence as a search technique.

Bodik and Barbara [40] surveyed algorithmic program synthesis research as an introduction to a journal special issue and analysed the application of these technologies. The survey is highly informative. However, it is relatively high-level as it attempted to canvas the program synthesis field and provide an overview of it. Furthermore, the work is relatively dated. The authors divided their review into reactive synthesis (i.e., concerned with automata-theoretic techniques with an infinite input stream), and functional synthesis (i.e., produces programs consuming finite inputs).

Gulwani et al. [41] surveyed and provided an overview of state-of-art approaches to program synthesis. This work also aims to provide a general overview of program synthesis, its applications, common approaches (particularly enumeration search, constraint solving, stochastic search, and deduction-based programming by examples), and general principles of such approaches (e.g., bias, oracle-guided inductive search, and optimisation).

Alur et al. [42] is the closest work to our study. The authors discuss the use of search-based approaches to program synthesis. The authors focused specifically on syntax-guided synthesis (SyGuS) using four applications: synthesis from logical specifications, programming by examples, program transformation, and automatic inference of program invariants. Given the publication setting, (i.e., in a communication magazine), the authors did not delve deeper into the search approaches.

In our work, we review program synthesis based on different types of user intent. Furthermore, we survey all search-based program synthesis techniques without focusing on a particular one, delving into their representation of the search space and the types of code generation problems at which they are the most successful.

3. Methodology

This section details the three main steps of the study: (i) definition of research questions, (ii) paper search, and (iii) paper screening and selection.

3.1. Definition of Research Questions

Our review aims to identify the existing research on SBPS algorithms by reviewing published literature.

The Research Questions (RQs) formulated for this study are as follows:

- **RQ1**: What are the main techniques and trends in SBPS?
- **RQ2**: What are the guiding principles of SBPS algorithms?
  - **RQ2.1**: What type of user intent/input is used to guide the SBPS search process?
  - **RQ2.2**: What search space representation is used by SBPS algorithms?
- **RQ3**: What are the types of programming tasks targetted by each SBPS algorithm?

3.2. Search For Relevant Papers

Finding an accurate search string for our study proved challenging due to the vast array of subdomains within the automated programming field and large number of techniques under the umbrella of search-based techniques. Given this diversity and instead of naming all potential keywords (i.e., all synonyms of program synthesis and all search-based algorithms), we opted to start our search with a query that only includes the main terms in our study–albeit with a high chance of returning a large number of false positives.

To conduct our comprehensive review, we executed keyword-based queries on 24/03/2024 within the Scopus digital library. Scopus, renowned for its collection of peer-reviewed publications from top software engineering journals and conferences, indexes research papers from several esteemed sources including IEEE Xplore, ACM Digital Library, ScienceDirect (Elsevier), and Springer.

To search the Scopus digital library, we use the following search string:

Search Query

```
("program generation" OR "code generation" OR "program synthesis" OR "automatic programming" OR "automated programming")
AND
("search" OR "heuristic" OR "metaheuristic")
```

This approach facilitated the identification of relevant studies, providing a foundation for our exploration of the nuanced landscape of automated programming. The utilisation of such a methodologically robust strategy ensures the inclusion of diverse perspectives and insights from reputable sources, contributing to the scholarly rigour and validity of our study.

3.3. Paper Screening and Selection

We retrieved 1002 publications by running the search string. After the implementation of the search string and the establishment of clear inclusion and exclusion parameters (outlined in Table 1), a meticulous screening process was executed on the retrieved papers, delineated in Figure 1.

In the initial phase, 671 papers were excluded based on their titles and abstracts as they did not align with the pre-defined criteria. The subsequent phase involved an in-depth examination of the remaining 331 studies through a full-text reading process, culminating in the exclusion of an additional 167 papers. The resulting subset of 164 publications exhibited an exclusive focus on SBPS. Notably, 95 studies were omitted from this subset as (i) secondary studies, (ii) minor incremental improvements of a primary study or (iii) survey papers.

Our full process enabled us to identify 69 papers contributing to novel search-based program synthesis algorithms.

Table 1. inclusion and exclusion criteria

Criteria	Description
Inclusion (IC1)	The work focuses on program synthesis using a search-based algorithm.
Exclusion (EC1)	The work is not written in English.
Exclusion (EC2)	The work published before January 2013.
Exclusion (EC3)	The work is a secondary study.
Exclusion (EC4)	The work is a minor incremental improvement of the approach.

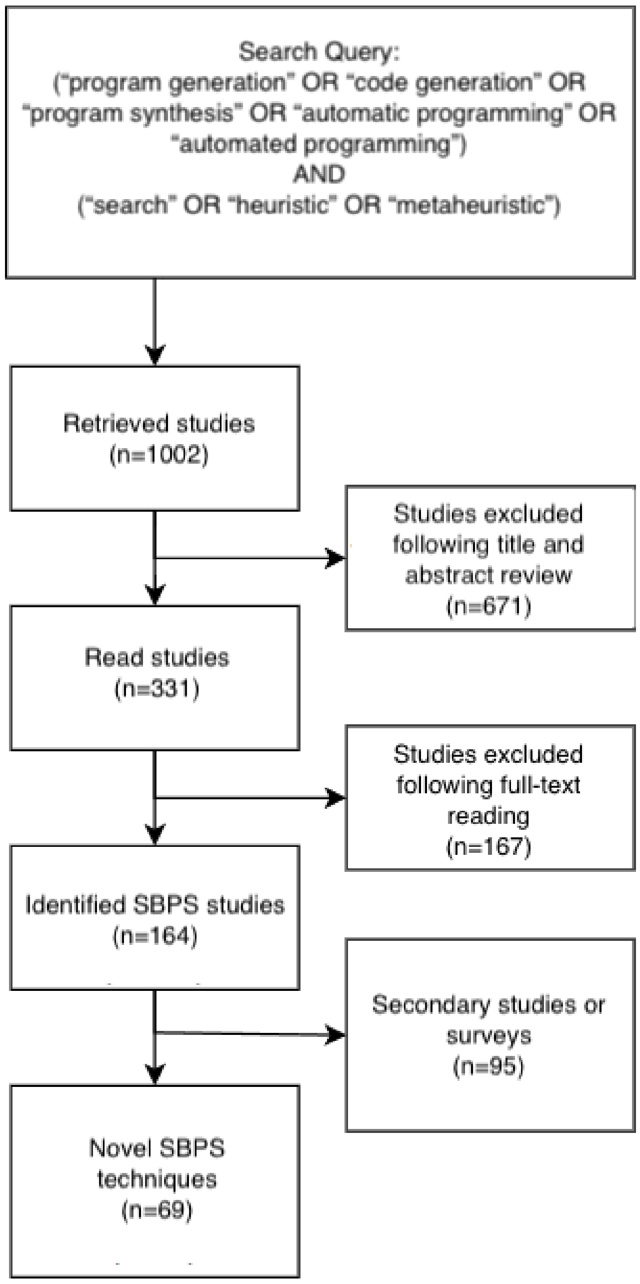
Search Query:
("program generation" OR "code generation" OR "program synthesis" OR "automatic programming" OR "automated programming")
AND
("search" OR "heuristic" OR "metaheuristic")
Retrieved studies (n=1002)
Studies excluded following title and abstract review (n=671)
Read studies (n=331)
Studies excluded following full-text reading (n=167)
Identified SBPS studies (n=164)
Secondary studies or surveys (n=95)
Novel SBPS techniques (n=69)

Figure 1. Overview of the literature search process.

4. Analysis

In this section, we present an in-depth analysis of our review in an attempt to answer our research questions, focusing on (i) the trend and techniques of SBPS, (ii) the guiding principles of each SBPS algorithm (type of user intent and representation of the search space), and (iii) the type of problem target.

4.1. Main SBPS Techniques and Trends (RQ1)

We start by analysing the type of SBPS techniques. Our survey identified a range of techniques for SBPS, which we have categorised into the following categories:

- **Uninformed:** denotes an algorithmic approach that systematically explores a given search space in a predetermined order, devoid of any heuristic strategies or domain-specific knowledge. Such algorithms adhere to a straightforward exploration pattern, sequentially examining potential solutions without incorporating any informed guidance or optimisation criteria.
- **Heuristic:** is an AI-based technique that is built for efficient search through large search spaces instead of using traditional exhaustive search methods. This type of algorithm often uses certain heuristics or rules to guide the search process towards the solution.
- **Metaheuristic:** is a higher-level search strategy that is designed to find optimal solutions for a wider range of problems instead of focusing on one particular task. This type of algorithm is often inspired by natural phenomena or human behaviour to guide the search. Metaheuristics often guide the search process using an iterative process in an attempt to reach better solutions.
- **Other:** category encompasses search-based program synthesis algorithms that cannot be classified within the previously defined categories. For instance, our survey identified a subset of papers that employ a pre-built database search approach for iterative program synthesis. Notably, these algorithms do not adhere to standard search techniques, and we have grouped them under this distinct category.

Table 2 provides a comprehensive summary of the identified approaches, classifying them according to the above algorithmic categories. Examination of the data underscores a notable predilection for the application of metaheuristics, evident in the identification of 33 papers employing them as their primary search strategy.

A variety of metaheuristic approaches were identified in our survey–with evolutionary approaches being the most used among them (in 25 papers). Analysing the evolutionary approaches shows that sharing the evolution framework, such approaches differ significantly in their architectures (e.g., Linear GP and Grammatical Evolution) and strategies (e.g., Linear, Push, and Tree-based GP). Additionally, combining GP with other techniques emerged as a prevalent strategy among various evolutionary approaches, with 6 papers. Correia et al. [43,44] proposed tackling the program synthesis problem as a model finding using a synthesiser called Alloy$*$. They further expanded their synthesiser to tackle complex synthesis problems by integrating it with a genetic programming module. Arcuri et al. [16] proposed Co-evolutionary program synthesis. This system evolves the program using a genetic algorithm and co-evolves it with a population of unit tests. They calculated the program's fitness using the unit test and evolved the unit test with these programs. Virgolin et al. [45] combined a model-based evolutionary algorithm called Gene-pool Optimal Mixing Evolutionary Algorithm with a tree-based genetic programming algorithm. Poliansky et al. [23] proposed genetic programming in conjunction with context-oriented behavioural programming.

The tree-based GP approach in the evolutionary approach category was utilised in 6 papers. Igwe and Pillay [46] investigated using a tree-based GP approach for program synthesis. Fernandes et al. [47] proposed a tree-based GP approach called Higher-order Typed GP with a grammar that supports higher-order functions, parametric polymorphism and parametric types. Hosseini Amini et al. [3] proposed a tree-based GP called Rule-Centered GP, which uses evolutionary rules to help the evolution process. Xu et al. [48] used tree-based GP with Lexicase selection for tackling job shop scheduling problems. Islam et al. [49] proposed a new mutation operator for tree-based GP that uses Monte Carlo simulation to expand and evaluate programs repeatedly. Moreover, Tao et al. [50] proposed a MOG3P, a multi-objective tree-based GP system by expanding objectives to code similarity to better guide the search process.

Four papers have been identified using GE as their approach in our survey. Kim et al. [4] proposed a new approach for GE that uses probabilistic modelling. They used probabilistic grammar and a new mapping process to create new individuals from the distribution of grammars. Schweim et al. [51] proposed multiple GE variants to use human-generated programs to guide the search. Chennupati et al. [52] presented a Multi-core GE algorithm to generate parallel sorting programs automatically. Whereas, Lopes and Costa [6] combined GE with the Artificial Regulatory Network model.

Three papers reported using Code Building Genetic Programming (CBGP). Partridge and Spector [53] proposed CBGP, a program synthesis system that supports generic functions and polymorphic types. It generates graphs that can be translated into programs in human-readable source code. A deeper exploration of the capability of CBGP is presented in [54]. They further formalised the method using the type theory algorithm and analysed the approach with other GP methods in [55].

For other evolutionary search methods, Krawiec et al. [56] proposed Counterexample-Driven Genetic Programming (CDGP). They produce counter-examples for failed tests during evolution and use them to drive the search process. Helmuth et al. [14] proposed a human-driven genetic programming system to make the system easier for non-experts to use. To reduce the large training data, they also used counter-examples to reduce the training cost. Moreover, Ahmad and Helmuth [38,57,58] used PushGP, a GP system that uses a stack-based language (i.e., Push), to test program synthesis performance on two proposed program benchmark suite and two novel initialisation techniques (i.e., Lexicase Seeding and Pareto Seeding). Finally, Serruto and Alfaro [59] used many-objective linear GP to generate assembly language programs. They decomposed the program into segments and evolved them simultaneously, allowing these segments to collaborate during the process.

Now, we briefly describe other non-evolutionary meta-heuristic methods in selected publications. Four papers were found using Local Search to generate code. Nguyen et al. [60] used Iterated Local Search to tackle dynamic job shop scheduling problems. Their key idea is to perform multiple local searches, starting a modification of the best-existing program. Rosin [61] proposed a Delayed Acceptance Hill Climbing method, which updates the current-best candidates after a period of gathering additional candidate programs using Local Search. Bornholt et al. [62] presented a program synthesis framework combining global and Local Search. The global search coordinates the activities of the Local Search, while the Local Search explores different candidate solutions. Feser et al. [63] performed bottom-up enumeration synthesis and then used Local Search to fix these programs.

Swarm Intelligence (SI) is utilised in two papers: Hara et al. [1] proposed parallel Ant Programming (AP) using genetic operators of GP to tackle the premature convergence problem of existing AP systems. Nekoei et al. [2] proposed artificial bee colony expression programming (ABCEP), which combines artificial bee colony programming and expression programming to tackle weak convergence and high locality.

We found Mahanipour and Nezamabadi-pour [5] applied the gravitational search algorithm in program synthesis. Golafshani [64] used Biogeography-Based Optimization (BBO) for program synthesis problems. BBO is a new evolutionary algorithm that is inspired by biogeography science.

In our survey, heuristic search approaches were utilised in 14 studies, with diverse heuristic search methods being employed in selected publications. Three papers found for using A$*$ algorithm: Jin et al. [65,66] applied A$*$ algorithm to synthesis data transformation program. Lee et al. [8] used a Syntax-guided synthesis (SyGuS) framework by applying a probabilistic model on grammar. They used the A$*$ algorithm to find the resulting program efficiently. Two papers reported using the divide-and-conquer search method: Cropper [67] combined Divide-and-conquer search with constraint-driven search to learn optimal, recursive and large programs. Chen et al. [68] proposed a tool called Facon to generate programs in domain-specific languages based on input-output examples. They applied the Divide-and-conquer principle to tackle the stability issue.

For other heuristic approaches, one paper was found for each type. Liu et al. [69] proposed a probability-based approach to synthesise Java programs. They reduced the search space with the knowledge from open-source code repositories. Wong et al. [70] introduced Language for Abstraction and Program Search, a technique that uses natural language information to guide the neurally-guided search models. Yoon et al. [71] proposed a bi-directional inductive synthesis method that uses iterative forward-backwards abstract interpretation. Cui and Zhu [72] used a gradient-descent-based method to learn the probability distribution over all possible program space. Hua et al. [73] proposed execution-driven sketching, a backtracking search approach to synthesise Java programs. Miltner et al. [12] used lenses with priority queues to transform different data representations bidirectionally. Yuan et al. [74] proposed a trace-guided approach using version space algebra to tackle ambiguity and generalisation challenges for synthesising recursive programs. Herrmann et al. [75] used expert rules tree search to generate computer vision programs. Osera and Zdancewic [76] used the proof-theoretic search technique to synthesise recursive functions that process algebraic datatypes.

Surprisingly, we found 16 papers using an uninformed search algorithm to perform program synthesis. Seven papers were identified using the enumerative search method: Polozov and Gulwani [77] proposed a data-driven domain-specific deduction method for inductive synthesis. It efficiently combined deductive inference with enumeration search to learn solution programs. Zhang et al. [78] proposed interpretable program synthesis, which enables users to interact and guide the search process. The enumerative search is utilised in their method. Valizadeh and Berger [13] proposed a data-parallel algorithm using an enumeration search for regular expression inference. Guria et al. [10] proposed a lightweight and language-agnostic program synthesiser that uses enumeration search. Feng et al. [79] proposed a component-based synthesis algorithm that combines type-directed enumeration search and SMT-based deduction. Feser et al. [80] presented a program synthesis approach utilising inductive generalisation, deduction and enumeration search. Polikarpova et al. [81] proposed a SBPS approach for synthesising recursive functions based on a polymorphic refinement type specification. Three papers utilised Best-first search for their algorithm: Ameen and Lelis [9] proposed the Best-first Bottom-up Search (BEE) search to reduce the information loss problem of const-guided bottom-up search. Chen et al. [82] used best-first search to synthesise network specifications in a declarative logic programming language. Cropper and Dumancic [15] used the example-dependent loss function to guide the best-first search to learn large programs. Two papers were identified using a Top-down search: Ye et al. [83] used a top-down search in a neural model to find a solution program that satisfies the user's natural language intent and input-output examples. Bowers et al. [7] proposed a corpus-guided synthesis algorithm that performs a top-down search to generate library functions from a corpus of domain-specific language. For other uninformed search algorithms, only one paper was identified. Barke et al. [84] proposed using a partial solution from the synthesis process to build the model instead of training the model before the code generation process. They used a bottom-up search that use a probabilistic model to guide the search efficiently. Heule et al. [85] utilised random search to generate models for opaque code automatically. Opaque code is an executable code whose source is unavailable. Guerra et al. [17] presented Hoogle$*$, a type-directed component-based program synthesiser that can handle constants and λ-abstractions compared to the previous version. Ren et al. [86] used a Breadth-first search to generate nuclear power software source code automatically.

We found 5 papers using problem-specific search methods to generate the program. Fix et al. [87] used a combinatorial evolution approach to open-ended programming. They stored code blocks in a database and iteratively combined them to generate programs. Saha et al. [22] proposed AutoML to generate an ML pipeline based on a corpus of human-written pipelines. Similarly, it builds a database using a corpus and iteratively constructs the pipeline using the knowledge learned from the database. Shimonaka et al. [88] proposed a reuse-based code generation technique that utilises the signature of the Java method and test cases. First, it constructs the database using the Java source files. Then it uses a method extractor, searcher and processor to generate a Java program iteratively. Liu et al. [89]

proposed a framework called ITAS, which can synthesise programs iteratively using API knowledge from the internet. Liu et al. [90] proposed API recommendation via a general search engine.

Table 2. Algorithms Used for SBPS in Selected Publications

Category	Subcategories		Papers	Count	
Uninformed	Random search		[85]	1	16
	Enumerative search		[10,13,77–81]	7	
	Best-first search		[9,15,82]	3	
	Depth-first search		[17]	1	
	Top-down search		[7,83]	2	
	Bottom-up search		[84]	1	
	Breadth-first search		[86]	1	
Heuristic	A∗ search		[8,65,66]	3	14
	Probability-based search		[69]	1	
	Neurally-guided search		[70]	1	
	Bidirectional search		[71]	1	
	Divide and conquer search		[67,68]	2	
	Gradient-descent search		[72]	1	
	Back-tracking search		[73]	1	
	Lenses with priority queue		[12]	1	
	Trace-guided search		[74]	1	
	Expert rules tree search		[75]	1	
	Proof search		[76]	1	
Metaheuristic	Local search		[60–63]	4	34
	Gravitational search		[5]	1	
	Swarm Intelligence		[1,2]	2	
	Biogeography-Based Programming		[64]	1	
	Evolution	Push GP	[38,57,58]	3	
		Grammatical Evolution	[4,6,51, 52]	4	
		Code-building GP	[53–55]	3	
		Tree-based GP	[3,46–50]	6	
		Counter-example guided GP	[14,56]	2	
		Linear GP	[11,59]	2	
		GP combined with other technique	[16,23,43–45,91]	6	
Other	Database-based iterative search		[22,87,88]	3	5
	API search		[89,90]	2	

Figure 2 shows the number of publications identified for each SBPS category along with the year. In the following analysis, we divided the evolutionary approach from the metaheuristic algorithms for a more detailed examination. The total number of research publications in this domain exhibits a consistent upward trajectory, indicating a growing interest in SBPS within the research community.

Analysing the trends for each category reveals the following insights: (i) GP exhibits a solid presence each year, showcasing its dominant position in this field. The number of publications leveraging GP shows an increasing trend by year. (ii) Uninformed search method was popular in 2015, and the number of publications reported using it rapidly increased in 2023. (iii) The diversity of algorithms has grown since 2020.

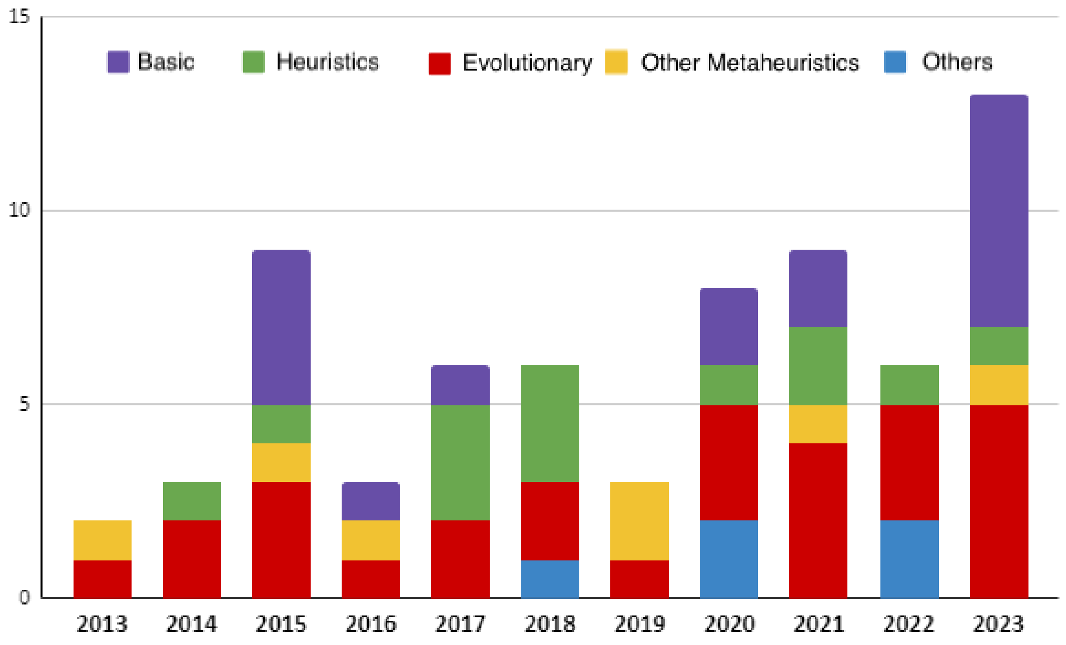

Figure 2. Number of studies by category and year

Key Findings Of RQ1

- ***Main SBPS techniques:*** *Metaheuristics are the most popular approaches for SBPS–among which, evolutionary approaches are most utilised.*
- ***Trend:*** *SBPS continues to attract an increasing and significant amount of novel work. SBPS approaches utilising evolutionary and uninformed search algorithms have become attractive in recent years.*

4.2. Guiding Principles of SBPS Algorithms (RQ2)

In this subsection, we analyse the search details of each SBPS algorithm in terms of required user input (intent) and representation of each solution.

4.2.1. Analysis of SBPS Algorithms Input Type (RQ2.1)

In this subsection, we analyse the input type of the selected SBPS algorithms. Programming by Example (PBE) is a programming paradigm that synthesises computer programs by using user-provided input-output examples. We introduce a new type of user interaction in computer synthesis, Programming by Instruction (PBI), which performs the programming synthesis based on the user's textual instruction. This textual instruction can be a natural language task description, programming rule, incomplete code snippet or other user specification form in a textual format.

Figure 3 illustrates the distribution of studies in our survey based on user input type. Notably, the PBE approach emerges as the predominant choice for synthesis algorithms, demonstrating consistent prevalence across all years. This preference can be attributed to its ability to guide the search towards correct solutions precisely. Specifically, a program is considered correct if it successfully passes all training and test input-output cases. The continued presence of PBI since 2020, independently or in conjunction with PBE, underscores the increasing popularity of textual intent, particularly in the context of the prevailing Large Language Models paradigm.

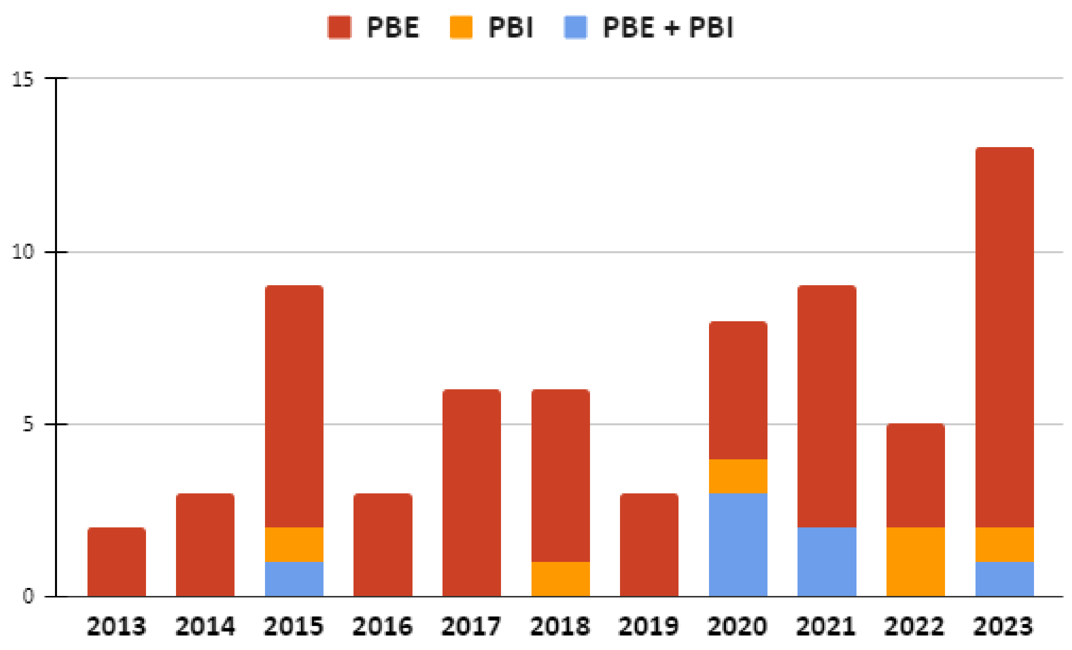

Figure 3. Number of studies by Input Type and Year

4.2.2. Representation of SBPS Search Space (RQ2.2)

We now analyse the representation type of the search space in the SBPS algorithm. Among the studies selected, diverse representation methods, such as tree, linear code sequences, and more, have been identified. We start with introducing these representations and aim to gain insights into the connection between the representation type and the search algorithm type, as well as the trend of each representation type, along with the published year.

The tree representation is one of the most popular approaches for forming the search space in SBPS, where solutions are structured hierarchically in a tree format. Nodes within the tree correspond to distinct code parts, and the hierarchy information and connections of tree nodes capture the program's logical flow. The advantage of containing the structural information in such representation makes the algorithm modify the solution programs easier.

Another way to represent the program in the search space is using linear code sequence, where programs are directly presented in the search space using a sequence of code blocks or code segments. In linear code sequence representation, each element in the sequence corresponds to a specific part of the code, and the sequential order dictates the program's execution flow. This representation type is straightforward, improving the solution program's interpretability during the search process.

Rule-based representation stands out as a declarative approach, encoding solutions as sets of rules or logical expressions. This method captures complex conditions and actions, offering a more abstract and high-level representation. This type of representation often uses predefined rule sets, and it ensures the syntactical correctness of the generated program.

Graph Representation is designed for problems where dependencies between program elements are crucial. Solutions are modelled as nodes and edges in a graph, effectively representing relationships and dependencies between different parts of the program.

In specific scenarios within the evolutionary algorithms such as Genetic Evolution (GE) [92], solutions are encoded as strings of symbols or characters. This type of representation is called string representation. In this approach, the sequence and arrangement of symbols within the string directly mirror the program's structure. In this representation, the term "genome" is often used to refer to the encoded string, encapsulating the genetic information that determines the solution. The corresponding program or solution derived from this genome is called "phenotype". It represents the expressed functionality of the genetic information encoded in the string. This encoding mechanism provides a concise and human-readable way to capture the essential elements of a program's structure within a string format.

Lastly, Stack Representation organises solutions for each data type into a stack in a last-in, first-out (LIFO) manner. This type of representation is introduced with Push language by [93,94], specifically designed for solving program synthesis problems.

Each representation type comes with its own set of strengths and is often chosen based on the specific requirements and characteristics of the program synthesis task. Table 3 provides an overview of the representation types identified in the surveyed papers, along with the corresponding papers.

Table 3. Representation used for SBPS in selected papers

Representation	Count	Papers
Tree	26	[1–3,7,9,11,16,17,23,46–50,56,60,63,64,71,74,76,78,79,83,84,86]
Rule-based	13	[8,10,12,15,23,61,68,72,73,77,80–82]
Linear Code Sequence	12	[22,43,44,62,67,70,75,85,88–91]
Graph	6	[53–55,65,66,69]
String	9	[4–6,13,14,45,51,52,59]
Stack	3	[38,57,58]

As shown in Table 3, there is a large domination of tree representation, with 26 publications, almost twice the number of other representation types. The reason behind this dominance is evolutionary search algorithms often use trees to represent the search space, and the evolutionary algorithms are reported as the most used algorithm among selected papers. It further underscores the effectiveness of the tree representation and the evolutionary algorithm for tackling program synthesis tasks. Followed by rule-based representation with 13 publications identified, linear code sequence with 12 and string representation with 9. While less prevalent, other representation types include graph (6 papers) and stack (3 papers). These alternative representations contribute to the diversity of approaches explored in the surveyed literature.

Table 4 summarises the algorithm types corresponding to each representation type. We observed a general preference for representation type for each algorithm type:

- The most common approach for representing search space for the metaheuristic approaches is a tree representation, which was reported in 14 out of 33 publications.
- It is also the same for non-evolutionary metaheuristic approaches that tree representation is identified as the most used approach in this category.
- The representation type for heuristic algorithms are evenly distributed on tree, rule-based, linear code sequence and graph. However, no selected paper reported a heuristic algorithm using string or stack representation.
- SBPS approaches using an uninformed search algorithm are more likely to use a tree or rule-based representation. No paper has been reported using an uninformed search with graph, string, or stack representation.
- For other SBPS algorithms, linear code sequence seems to be the promising way to synthesise the program, while all 5 studies in this category reported using linear code sequence as a representation method.
- Rule-based representation is mainly utilised in the uninformed search algorithms, while string and stack representations are utilised in the evolutionary metaheuristic approach.

Table 4. Representation used for SBPS with algorithm type

Representation	**Uninformed**	**Heuristic**	**Metaheuristic**		**Other**
			Non-Evolution	**Evolution**	
Tree	[7,9,17,78,79,83,84,86]	[71,76]	[1,2,60,63]	[3,11,16,23,46–50,56]	
Rule-based	[10,15,77,80–82]	[8,72,73]	[61]	[23]	
Linear Code Sequence	[85]	[67,75]	[62]	[43,44,91]	[22,87–90]
Graph		[65,66,69]		[53–55]	
String			[5]	[4,6,14,45,51,52,59]	
Stack				[38,57,58]	

Figure 4 visually represents the distribution of studies across different representation types by year. We noticed that the usage of tree representation shows an increasing trend over the years, with a notable spike in 2023. Linear Code Sequence representation experienced a rise from 2020. However, this representation was not utilised in any selected research in 2023. The rule-based representation showed a steady presence from 2015 to 2023, with a spike in 2018. The graph representation was first used in 2017 and has shown frequent presence in recent years. The string representation was distributed randomly along the selected period and did not show a clear pattern.

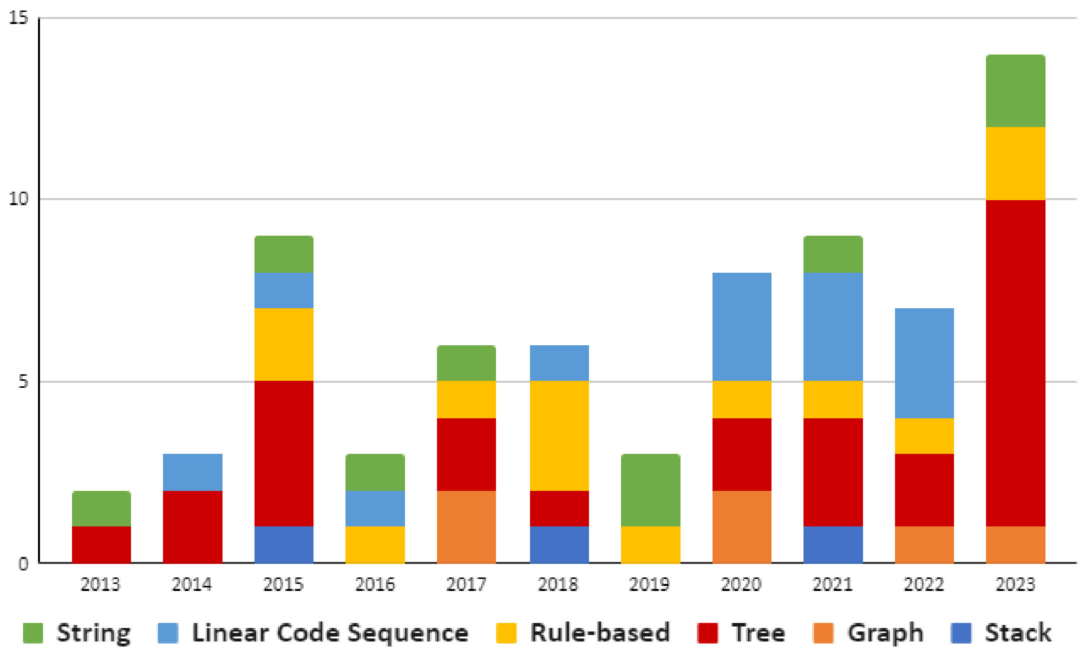

Figure 4. Number of studies by representation and year

Overall, the trends highlight the dynamic nature of representation choices in SBPS. Tree, linear code sequence, and rule-based representations emerge as popular choices, each with specific strengths and applications. Other representations, such as stack, show sporadic usage, indicating their suitability for particular scenarios or research contexts.

Key Findings Of RQ2

- ***Type of user intent:*** *SBPS approaches utilising the PBE technique is the dominant way to guide the search, while the PBI approach has become popular recently.*
- ***Seach space representation:*** *The Tree is the most popular approach for search space representation. The diversity of the type of representation has grown over the year.*

4.3. Type of Task Targeted by Each SBPS Algorithm (RQ3)

Going further, we analysed the targeted problem type for each publication and the dataset used for the experiment. By examining the addressed problem type and the dataset, we aim to understand the research context and methodology adopted in SBPS.

Table 5 shows the problem types and datasets employed in the selected publications. We observed a diverse spectrum of program synthesis approaches spanning various application domains. These domains range from simple problems such as algebraic calculations, symbolic regression, etc., to more challenging applications such as robot path-finding and high-level programming tasks in diverse languages like Python and Java. Notably, some studies extend their investigations to real-world complex problems, targeting ML pipeline generation and synthesising feature constructions for preprocessing the ML data.

Table 5. Problem Type with Used Dataset in selected publications

Problem Type	Dataset	Count		Papers
Symbolic Regression	Custom Benchmark	9		[1–7,45,49]
String Manipulation	SyGuS	10	5	[8–11,84]
	Custom Benchmark		5	[12–15,83]
Circuit Transformation	SyGuS	6	3	[8,71,84]
	Custom Benchmark		3	[1,4,59]
Array/Vector Transformation	OpenAI Gym toolkit	18	1	[91]
	Sorting		2	[16,52]
	SyGuS		9	[8,9,11,62,62,62,62,71, 84]
	Custom Benchmark		6	[17,43,45,76,80,81]
Programming	Apache dataset (JAVA)	29	1	[88]
	SyPet (JAVA)		3	[69,89,90]
	PSB1		7	[38,47,50,53,55,57,61]
	PSB2		1	[58]
	Algebra Calculation		6	[14,16,17,43,44,56]
	Computer Vision		1	[75]
	Array.prototype (Java Script)		1	[85]
	java.util(Java)		1	[73]
	Custom Benchmark		8	[7,14,46,51,54,67,74,90]
Other	ASCII Art	21	1	[15]
	Path Finding		2	[6,15]
	ML Pipeline		1	[22]
	Custom Real-World Problem		4	[3,4,68,72]
	Game of Tic-Tac-Toe		1	[23]
	Data Transformation		5	[10,65,66,79,80]
	Job Shop Scheduling		2	[48,60]
	User Study		1	[78]
	Feature Construction		1	[5]
	Network Analysis		1	[82]
	Inverse Constructive Solid Geometry		1	[63]
	Nuclear Power Software Development		1	[86]

We categorised the target problem of selected publications into six types:

- **Symbolic Regression:** This type of problem aims to discover mathematical expressions or symbolic representations that model the relationship within a given dataset.
- **String Manipulation:** It involves the task of generating or transforming strings based on specific rules or requirements.
- **Circuit Transformation:** This problem type targets automatically modifying or optimising electronic circuits based on a digital specification of certain circuits.
- **Array/Vector Transformation:** Similar to other transformation tasks, this problem type aims to manipulate or transform elements within arrays or vectors.
- **Programming:** This category of problem type generally aims to solve programming tasks using high-level programming languages like Python, Java, etc.
- **Others:** In this problem type, we have collected relatively challenging real-world problems which can not be included in previous categories.

Most selected studies aim for programming tasks (29), followed by other relatively challenging problems (21). Closely followed by array/vector transformation, 18 studies were reported. The distribution on other manipulation/ transformation-type problems is even, where 9 studies target symbolic regression, 10 for string manipulation, and 6 for circuit transformation.

Furthermore, we also observed patterns in the selection of datasets across different problem domains. Notably, datasets from the SyGuS competition are prominently featured in studies addressing challenges in string manipulation, circuit transformation and array/vector transformation problems. Interestingly, there is a notable absence of standard, commonly used benchmarks for symbolic regression in these studies, suggesting a preference for creating custom benchmarks tailored to the intricacies of this problem type. For high-level programming tasks, the General Program Synthesis Benchmark Suite (PSB1) [38] emerges as a promising benchmark, employed in 7 studies to evaluate the effectiveness of algorithms. Helmuth and Kelly published their updated benchmark, General Program Synthesis Benchmark Suite (PSB2) [58] in 2021. However, 3 studies [47,50,55] published after the release of the new version of the benchmark did not choose to use it to compare the algorithm with other approaches on PSB1. Additionally, the benchmark first used in SyPet [95] has gained popularity as a dataset for Java programming tasks. An interesting observation is using a user study to evaluate a code generation system in [78], showcasing a unique and user-centred approach in the selected research landscape.

We now describe some of the interesting problems collected in the "other" category, considering problems contained in this category are relatively challenging compared to other categories. Cropper and Dumancic [15] used the ILP system with best-first search to address the problem of learning to draw ASCII art. Saha et al. [22] proposed a technique that can generate an ML pipeline for a predictive task on a new dataset, while Mahanipour et al. [5] focused on feature construction that is an important task in pre-processing in ML tasks. Poliansky et al. [23] demonstrated their approach to the game of Tic-Tac-Toe. Chen et al. [82] aims to synthesise network specifications.

Going further, we study the relationship between the classification of problem type and the employed search algorithm type. Table 6 illustrates the distribution of selected publications based on problem types and the corresponding search algorithms. At least one publication was found for the evolutionary approach applied in each category of problems, with a primary focus on programming tasks (16 out of 34 studies). This highlights the great scalability of the evolutionary approach across various problem domains. Similarly, SBPS algorithms using "other" search algorithms show great presence on programming problems. We noticed these problem-specific techniques generally target relatively harder problems, considering no studies were found for tackling transformation or manipulation problems. The remaining metaheuristic, heuristic, and uninformed search algorithms exhibit even distribution across each problem domain.

Table 6. Target Problem Types per SBPS Algorithm Type

Problem Type	Uninformed	Heuristic	Metaheuristic		Other
			Non-Evolution	Evolution	
Symbolic Regression			[1,2,5]	[3,4,6,49]	
String Manipulation	[7,9,10,15,83,84]	[8]		[11,14]	
Circuit Transformation	[84]	[8,71]	[1]	[4]	
Array/Vector Transformation	[9,17,80,81,84]	[8,71,76]	[61,62]	[11,16,43,52,91]	
Programming	[7,17,85]	[67,69,73]	[61]	[14,16,38,43–47,50,51,53–58]	[87–90]
Other	[10,15,78–80,82,86]	[65,66,72,75]	[5,60,63]	[3,4,6,23,48,59]	[22]

Finally, we analyse the relationship between the representation and target problem types, aiming to gain insight into which representation is more suitable for various domains. Table 7 shows the distribution of selected publications based on representation and problem type. Algorithms using tree and string representation are applied to every identified problem type. However, tree representations focus more on solving relatively challenging target problems, i.e. programming tasks and other real-world programming tasks, while SBPS approaches with string representation are evenly distributed. Linear code sequence and graph representations are mainly employed in harder problem types, such as array/vector transformation, programming and other challenging real-world tasks. Rule-based representations are mainly used to tackle string manipulation and other real-world tasks. Stack representation is utilised in programming and other challenging programming tasks, showing potential for solving harder problems.

Table 7. Target problem type assessed for SBPS with representation type

Problem Type	Tree	Rule-based	Linear Code Sequence	Graph	String	Stack
Symbolic Regression	[1–3,5,49,71]				[4–6]	
String Manipulation	[7,9,83,84]	[8,10–12,15]			[13,14]	
Circuit Transformation	[1,71,84]	[8]			[4,59]	
Array/Vector Transformation	[9,11,16,17,76,84]	[80]	[8,43,62,81,91]	[62]	[45,52]	
Programming	[7,16,17,46,47,50,56,74]	[61,73]	[43,44,67,75,85,88–90]	[53–55,69]	[14,51]	[38,58]
Other	[3–5,15,23,48,60,63,78,79,86]	[15,23,61,68,72,80,82]	[22,67]	[65,66]	[5,6]	[57]

Key Findings Of RQ3

- ***Type of tasks:*** *SBPS approaches identified in this survey targeted various problems, from simple modification/manipulation problems to more challenging programming and real-world problems.*
- ***Relationship between problem type and search algorithm type:*** *Symbolic Regression tasks are mostly tackled with metaheuristics, whereas String Manipulation tasks are mostly tackled with uninformed algorithms. Programming tasks attracted a wide range of techniques, however evolutionary methods are the most used.*
- ***Relationship between problem type and representation type:*** *Tree and String representations are utilised to tackle all kinds of problems, while Linear Code Sequence and Graph representations are utilised more in challenging problems.*

5. Existing Challenge in SBPS

In this section, we summarise and discuss the observations and existing challenges from the results of our previous research questions analysis, thereby attempting to provide insight into the direction of future work.

5.1. Bridging Theory and Practice

We have noticed that many studies report their algorithms being evaluated on relatively straightforward problems, i.e. transformation and manipulation, restricting their availability to handle real-world programming scenarios. This reveals a significant hurdle in the early developmental stages of the field of SBPS, where most studies lack the ability to assess problems beyond the theoretical domain into real-world applications or tasks that transcend the predefined boundaries of training and testing scopes. Among the six categories of problem types that we collected on Table 5, more than half of them (Symbolic regression, string manipulation, circuit transformation and array/vector transformation, and algebra calculation in generic programming) are considered theoretical problems, which make up approximately 50% (43 out of 91) of the problem we analysed. Even other benchmarks in the programming category often contain many basic-level tasks. For example, PSB1 is a dataset with 29 problems selected from elementary-level programming courses, making it considerably easier than real-world problems.

The challenge of bridging the gap between theoretical frameworks and real-world applicability is evident in the field of SBPS. Looking ahead, finding ways to overcome these challenges and making SBPS techniques that are more applicable to real-world problems will be crucial for the future of this field. Thus, it's important to recognise and overcome the hurdles at this early stage to guide the field toward a more robust and impactful future.

5.2. Advancing Algorithms: Tools, Strategies, and Evolution

We observed that numerous studies demonstrate their implementation with a simple prototype, with the primary objective of showcasing the effectiveness of the proposed ideas through the employment of comparatively elementary principles and strategies. Nonetheless, they mentioned that better tools and strategies are needed to move these algorithms towards better performance. The academic community agrees that we urgently need stronger tools, like improved search strategies and fine-tuning of parameters. We realise that while the first versions of these algorithms may show promise in theory, their full potential remains untapped unless we use more advanced methods. To apply the SBPS field to complex problems, researchers and practitioners must recognise and embrace these acknowledged needs as opportunities for future progress. By directly tackling these challenges, the academic community can significantly contribute to improving and refining algorithmic approaches, making sure they can be used effectively in a wide range of situations.

5.2.1. Absence of a Common Benchmark

Our comprehensive survey reveals significant variability in programming tasks and datasets, which makes it challenging to compare the strength of each algorithm. A lack of standardised benchmarks persists across selected publications, even within the same programming task, such as symbolic regression. Although several studies have employed the SyGuS competition dataset as an evaluation benchmark, the specific SyGuS problems differ each year, and the analysed studies often select problems from distinct versions of the benchmark. One well-constructed benchmark, PSB1, has been utilised in six studies. However, considering the 28 studies identified in the programming task category, the utilisation rate remains notably low. We recommend that future research endeavours prioritise adopting common problem sets and datasets for program synthesis tasks. Such standardisation will facilitate more meaningful comparisons and drive advancements in the field.

5.3. Computational Challenges in Search-Based Program Synthesis

Our survey findings underscore a noteworthy observation regarding the computational overhead inherent in SBPS. It becomes evident that this computational cost rapidly increases during the search process, primarily attributable to the expanding dimensions of the search space and the consequential expenses incurred during the fitness evaluation. The expansion of the search space, a consequence of the intricate nature of the programming landscape, contributes significantly to the escalated computational demands. Future work can aim to reduce the search space while not influencing the performance of the search algorithms. Moreover, the expense associated with fitness evaluation emerges as a critical determinant in the observed higher computational costs. Most search-based algorithms refine their goal based on fitness evaluation, which has to be applied to every individual within the search space. Potential future work can be done to improve the efficiency of the fitness evaluation.

6. Conclusion

In pursuing a comprehensive understanding of the contemporary landscape of program synthesis through search-based algorithms, we surveyed the existing body of literature. This study revealed a considerable corpus of works dedicated to applying search-based algorithms in the realm of program synthesis. Notably, there is a discernible upward trajectory in the annual publication rate, indicative of this research domain's growing interest and significance.

The selected studies underwent a thorough analysis, enabling the categorisation of these works based on the employed search techniques, user intent type, and representations of the search space. This categorisation was complemented by trend analysis, offering insights into the evolving methodologies and approaches within the field. Furthermore, the review encompassed the collection of targeted problem types and datasets from each identified study, facilitating an in-depth examination of the empirical foundations underpinning these investigations. We also conducted an analysis of the relationships between the attributes we categorised, including target problem types and algorithm types to provide navigation guidelines for this large field particularly. These guidelines are of high and timely importance. As we witness a new research community (i.e., generative AI) taking on the program synthesis challenge, our study will provide an entry point to the acumen and expertise of the search-based community and help build synergy for collaboration between the two communities.

In addition to synthesising the current state of research in this domain, this review ventured into a comprehensive discussion of the challenges inherent in the field. By acknowledging and elucidating these challenges, the review contributes to the scholarly discourse, laying the groundwork for future research endeavours to address and overcome these complexities. Thus, this study is a valuable resource for researchers, educators, and practitioners alike, fostering a deeper understanding of the evolving landscape of program synthesis through search-based algorithms.

Acknowledgement

This work was supported by Science Foundation Ireland grant 13/RC/2094_P2 to Lero - the Science Foundation Ireland Research Centre for Software (www.lero.ie).

References

1. Hara, A.; Kushida, J.i.; Tanabe, S.; Takahama, T. Parallel Ant Programming using genetic operators. In Proceedings of the 2013 IEEE 6th International Workshop on Computational Intelligence and Applications (IWCIA), 2013, pp. 75–80.
2. Nekoei, M.; Moghaddas, S.A.; Mohammadi Golafshani, E.; Gandomi, A.H. Introduction of ABCEP as an automatic programming method. *Information Sciences* **2021**, *545*, 575–594.
3. Hosseini Amini, S.M.H.; Abdollahi, M.; Amir Haeri, M. Rule-centred genetic programming (RCGP): an imperialist competitive approach. *Applied Intelligence* **2020**, *50*, 2589–2609.
4. Kim, H.T.; Kang, H.K.; Ahn, C.W. A Conditional Dependency Based Probabilistic Model Building Grammatical Evolution. *IEICE Transactions on Information and Systems* **2016**, *E99.D*, 1937–1940.
5. Mahanipour, A.; Nezamabadi-Pour, H. GSP: an automatic programming technique with gravitational search algorithm. *Applied Intelligence* **2019**, *49*, 1502–1516.
6. Lopes, R.L.; Costa, E. GEARNet: Grammatical Evolution with Artificial Regulatory Networks. In Proceedings of the Proceedings of the 15th Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, 2013; GECCO '13, p. 973–980.
7. Bowers, M.; Olausson, T.X.; Wong, L.; Grand, G.; Tenenbaum, J.B.; Ellis, K.; Solar-Lezama, A. Top-Down Synthesis for Library Learning. *Proc. ACM Program. Lang.* **2023**, *7*.
8. Lee, W.; Heo, K.; Alur, R.; Naik, M. Accelerating Search-Based Program Synthesis Using Learned Probabilistic Models. In Proceedings of the Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, New York, NY, USA, 2018; PLDI 2018, p. 436–449.

9. Ameen, S.; Lelis, L.H. Program synthesis with best-first bottom-up search. *Journal of Artificial Intelligence Research* **2023**, *77*, 1275–1310.
10. Guria, S.N.; Foster, J.S.; Van Horn, D. Absynthe: Abstract Interpretation-Guided Synthesis. *Proc. ACM Program. Lang.* **2023**, *7*.
11. Yuan, Y.; Banzhaf, W. Iterative genetic improvement: Scaling stochastic program synthesis. *Artificial Intelligence* **2023**, *322*, 103962.
12. Miltner, A.; Fisher, K.; Pierce, B.C.; Walker, D.; Zdancewic, S. Synthesizing Bijective Lenses. *Proc. ACM Program. Lang.* **2017**, *2*.
13. Valizadeh, M.; Berger, M. Search-Based Regular Expression Inference on a GPU. *Proc. ACM Program. Lang.* **2023**, *7*.
14. Helmuth, T.; Frazier, J.G.; Shi, Y.; Abdelrehim, A.F. Human-Driven Genetic Programming for Program Synthesis: A Prototype. In Proceedings of the Proceedings of the Companion Conference on Genetic and Evolutionary Computation, New York, NY, USA, 2023; GECCO '23 Companion, p. 1981–1989.
15. Cropper, A.; Dumančić, S. Learning large logic programs by going beyond entailment. *arXiv preprint arXiv:2004.09855* **2020**.
16. Arcuri, A.; Yao, X. Co-evolutionary automatic programming for software development. *Information Sciences* **2014**, *259*, 412–432.
17. Botelho Guerra, H.; Ferreira, J.F.; Costa Seco, J. Hoogle: Constants and Lambda-abstractions in Petri-net-based Synthesis using Symbolic Execution. In Proceedings of the 37th European Conference on Object-Oriented Programming (ECOOP 2023); Ali, K.; Salvaneschi, G., Eds., Dagstuhl, Germany, 2023; Vol. 263, *Leibniz International Proceedings in Informatics (LIPIcs)*, pp. 4:1–4:28.
18. Tao, N.; Ventresque, A.; Saber, T. Program synthesis with generative pre-trained transformers and grammar-guided genetic programming grammar. In Proceedings of the LA-CCI. IEEE, 2023, pp. 1–6.
19. Tao, N.; Ventresque, A.; Saber, T. Assessing similarity-based grammar-guided genetic programming approaches for program synthesis. In Proceedings of the OLA. Springer, 2022.
20. Tao, N.; Ventresque, A.; Saber, T. Many-objective Grammar-guided Genetic Programming with Code Similarity Measurement for Program Synthesis. In Proceedings of the IEEE LACCI, 2023.
21. Tao, N.; Ventresque, A.; Saber, T. Multi-objective grammar-guided genetic programming with code similarity measurement for program synthesis. In Proceedings of the IEEE CEC, 2022.
22. Saha, R.K.; Ura, A.; Mahajan, S.; Zhu, C.; Li, L.; Hu, Y.; Yoshida, H.; Khurshid, S.; Prasad, M.R. SapientML: Synthesizing Machine Learning Pipelines by Learning from Human-Writen Solutions. In Proceedings of the Proceedings of the 44th International Conference on Software Engineering, New York, NY, USA, 2022; ICSE '22, p. 1932–1944.
23. Poliansky, R.; Sipper, M.; Elyasaf, A. From Requirements to Source Code: Evolution of Behavioral Programs. *Applied Sciences* **2022**, *12*.
24. Beltramelli, T. pix2code: Generating code from a graphical user interface screenshot. In Proceedings of the Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, 2018, pp. 1–6.
25. Li, Y.; Choi, D.; Chung, J.; Kushman, N.; Schrittwieser, J.; Leblond, R.; Eccles, T.; Keeling, J.; Gimeno, F.; Dal Lago, A.; et al. Competition-Level Code Generation with AlphaCode, 2022.
26. Jesse, K.; Ahmed, T.; Devanbu, P.T.; Morgan, E. Large language models and simple, stupid bugs. In Proceedings of the IEEE/ACM MSR. IEEE, 2023, pp. 563–575.
27. Asare, O.; Nagappan, M.; Asokan, N. Is github's copilot as bad as humans at introducing vulnerabilities in code? *Empirical Software Engineering* **2023**, *28*, 129.
28. Schuster, R.; Song, C.; Tromer, E.; Shmatikov, V. You autocomplete me: Poisoning vulnerabilities in neural code completion. In Proceedings of the USENIX Security 21, 2021, pp. 1559–1575.
29. Stechly, K.; Marquez, M.; Kambhampati, S. GPT-4 Doesn't Know It's Wrong: An Analysis of Iterative Prompting for Reasoning Problems. *arXiv* **2023**.
30. Krishna, S.; Agarwal, C.; Lakkaraju, H. Understanding the Effects of Iterative Prompting on Truthfulness. *arXiv* **2024**.
31. Pinna, G.; Ravalico, D.; Rovito, L.; Manzoni, L.; De Lorenzo, A. Enhancing Large Language Models-Based Code Generation by Leveraging Genetic Improvement. In Proceedings of the European Conference on Genetic Programming (Part of EvoStar). Springer, 2024, pp. 108–124.
32. Hemberg, E.; Moskal, S.; O'Reilly, U.M. Evolving Code with A Large Language Model. *arXiv preprint arXiv:2401.07102* **2024**.

33. Hemberg, E.; Jorgensen, S.; O'Reilly, U.M. Survey of Genetic Programming and Large Language Models. In *Genetic Programming Theory and Practice XXI*; Springer, 2025; pp. 67–86.
34. Tao, N.; Ventresque, A.; Nallur, V.; Saber, T. Grammar-obeying program synthesis: A novel approach using large language models and many-objective genetic programming. *Computer Standards & Interfaces* **2025**, *92*, 103938.
35. Tao, N.; Ventresque, A.; Nallur, V.; Saber, T. Enhancing Program Synthesis with Large Language Models Using Many-Objective Grammar-Guided Genetic Programming. *Algorithms* **2024**, *17*, 287.
36. Batouta, Z.I.; Dehbi, R.; Talea, M.; Hajoui, O. Automation in code generation: Tertiary and systematic mapping review. In Proceedings of the 2016 4th IEEE International Colloquium on Information Science and Technology (CiSt), 2016, pp. 200–205.
37. Sobania, D.; Schweim, D.; Rothlauf, F. A Comprehensive Survey on Program Synthesis With Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation* **2023**, *27*, 82–97.
38. Helmuth, T.; Spector, L. General program synthesis benchmark suite. In Proceedings of the GECCO, 2015.
39. Olmo, J.L.; Romero, J.R.; Ventura, S. Swarm-based metaheuristics in automatic programming: a survey. *WIREs Data Mining and Knowledge Discovery* **2014**, *4*, 445–469.
40. Bodík, R.; Jobstmann, B. Algorithmic program synthesis: introduction. *International journal on software tools for technology transfer* **2013**, *15*, 397–411.
41. Gulwani, S.; Polozov, O.; Singh, R.; et al. Program synthesis. *Foundations and Trends® in Programming Languages* **2017**, *4*, 1–119.
42. Alur, R.; Singh, R.; Fisman, D.; Solar-Lezama, A. Search-based program synthesis. *Communications of the ACM* **2018**, *61*, 84–93.
43. Correia, A.; Iyoda, J.; Mota, A. A family of multi-concept program synthesisers in Alloy*. *Science of Computer Programming* **2021**, *201*, 102536.
44. Correia, A.; Iyoda, J.; Mota, A. Combining model finder and genetic programming into a general purpose automatic program synthesizer. *Information Processing Letters* **2020**, *154*, 105866.
45. Virgolin, M.; Alderliesten, T.; Witteveen, C.; Bosman, P.A.N. Scalable Genetic Programming by Gene-Pool Optimal Mixing and Input-Space Entropy-Based Building-Block Learning. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference, New York, NY, USA, 2017; GECCO '17, p. 1041–1048.
46. Igwe, K.; Pillay, N. Automatic programming using genetic programming. In Proceedings of the 2013 Third World Congress on Information and Communication Technologies (WICT 2013), 2013, pp. 337–342.
47. Fernandes, M.C.; de França, F.O.; Francesquini, E. HOTGP–Higher-Order Typed Genetic Programming. *arXiv preprint arXiv:2304.03200* **2023**.
48. Xu, M.; Mei, Y.; Zhang, F.; Zhang, M. Genetic Programming with Lexicase Selection for Large-scale Dynamic Flexible Job Shop Scheduling. *IEEE Transactions on Evolutionary Computation* **2023**, pp. 1–1.
49. Islam, M.; Kharma, N.N.; Grogono, P. Expansion: A Novel Mutation Operator for Genetic Programming. In Proceedings of the IJCCI, 2018, pp. 55–66.
50. Tao, N.; Ventresque, A.; Saber, T. Multi-objective Grammar-guided Genetic Programming with Code Similarity Measurement for Program Synthesis. In Proceedings of the 2022 IEEE Congress on Evolutionary Computation (CEC), 2022, pp. 1–8.
51. Schweim, D.; Hemberg, E.; Sobania, D.; O'Reilly, U.M.; Rothlauf, F. Using Knowledge of Human-Generated Code to Bias the Search in Program Synthesis with Grammatical Evolution. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference Companion, New York, NY, USA, 2021; GECCO '21, p. 331–332.
52. Chennpati, G.; Azad, R.M.A.; Ryan, C. On the Automatic Generation of Efficient Parallel Iterative Sorting Algorithms. In Proceedings of the Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, New York, NY, USA, 2015; GECCO Companion '15, p. 1369–1370.
53. Pantridge, E.; Spector, L. Code Building Genetic Programming. In Proceedings of the Proceedings of the 2020 Genetic and Evolutionary Computation Conference, New York, NY, USA, 2020; GECCO '20, p. 994–1002.
54. Pantridge, E.; Helmuth, T. Solving Novel Program Synthesis Problems with Genetic Programming using Parametric Polymorphism. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference, New York, NY, USA, 2023; GECCO '23, p. 1175–1183.
55. Pantridge, E.; Helmuth, T.; Spector, L. Functional Code Building Genetic Programming. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference, New York, NY, USA, 2022; GECCO '22, p. 1000–1008.

56. Krawiec, K.; Blkadek, I.; Swan, J. Counterexample-Driven Genetic Programming. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference, New York, NY, USA, 2017; GECCO '17, p. 953–960.
57. Ahmad, H.; Helmuth, T. A Comparison of Semantic-Based Initialization Methods for Genetic Programming. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference Companion, New York, NY, USA, 2018; GECCO '18, p. 1878–1881.
58. Helmuth, T.; Kelly, P. PSB2: the second program synthesis benchmark suite. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference, 2021, pp. 785–794.
59. Serruto, W.F.; Alfaro, L. Many-Objective Cooperative Co-evolutionary Linear Genetic Programming Applied to the Automatic Microcontroller Program Generation. *International Journal of Advanced Computer Science and Applications* **2019**, *10*.
60. Nguyen, S.; Zhang, M.; Johnston, M.; Tan, K.C. Automatic Programming via Iterated Local Search for Dynamic Job Shop Scheduling. *IEEE Transactions on Cybernetics* **2015**, *45*, 1–14.
61. Rosin, C.D. Stepping stones to inductive synthesis of low-level looping programs. In Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence, 2019, pp. 2362–2370.
62. Bornholt, J.; Torlak, E.; Grossman, D.; Ceze, L. Optimizing Synthesis with Metasketches. In Proceedings of the Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New York, NY, USA, 2016; POPL '16, p. 775–788.
63. Feser, J.; Dillig, I.; Solar-Lezama, A. Inductive Program Synthesis Guided by Observational Program Similarity. *Proc. ACM Program. Lang.* **2023**, *7*.
64. Golafshani, E.M. Introduction of Biogeography-Based Programming as a new algorithm for solving problems. *Applied Mathematics and Computation* **2015**, *270*, 1–12.
65. Jin, Z.; Anderson, M.R.; Cafarella, M.; Jagadish, H.V. Foofah: Transforming Data By Example. In Proceedings of the Proceedings of the 2017 ACM International Conference on Management of Data, New York, NY, USA, 2017; SIGMOD '17, p. 683–698.
66. Jin, Z.; Anderson, M.R.; Cafarella, M.; Jagadish, H.V. Foofah: A Programming-By-Example System for Synthesizing Data Transformation Programs. In Proceedings of the Proceedings of the 2017 ACM International Conference on Management of Data, New York, NY, USA, 2017; SIGMOD '17, p. 1607–1610.
67. Cropper, A. Learning logic programs though divide, constrain, and conquer. In Proceedings of the Proceedings of the AAAI Conference on Artificial Intelligence, 2022, pp. 6446–6453.
68. Chen, H.; Wang, A.; Loo, B.T. Towards Example-Guided Network Synthesis. In Proceedings of the Proceedings of the 2nd Asia-Pacific Workshop on Networking, New York, NY, USA, 2018; APNet '18, p. 65–71.
69. Liu, B.B.; Dong, W.; Liu, J.X.; Zhang, Y.T.; Wang, D.Y. Prosy: Api-based synthesis with probabilistic model. *Journal of Computer Science and Technology* **2020**, *35*, 1234–1257.
70. Wong, C.; Ellis, K.M.; Tenenbaum, J.; Andreas, J. Leveraging language to learn program abstractions and search heuristics. In Proceedings of the International conference on machine learning. PMLR, 2021, pp. 11193–11204.
71. Yoon, Y.; Lee, W.; Yi, K. Inductive program synthesis via iterative forward-backward abstract interpretation. *Proceedings of the ACM on Programming Languages* **2023**, *7*, 1657–1681.
72. Cui, G.; Zhu, H. Differentiable synthesis of program architectures. *Advances in Neural Information Processing Systems* **2021**, *34*, 11123–11135.
73. Hua, J.; Khurshid, S. EdSketch: Execution-Driven Sketching for Java. In Proceedings of the Proceedings of the 24th ACM SIGSOFT International SPIN Symposium on Model Checking of Software, New York, NY, USA, 2017; SPIN 2017, p. 162–171.
74. Yuan, Y.; Radhakrishna, A.; Samanta, R. Trace-Guided Inductive Synthesis of Recursive Functional Programs. *Proc. ACM Program. Lang.* **2023**, *7*.
75. Herrmann, M.; Mayer, C.; Radig, B. Automatic generation of image analysis programs. *Pattern recognition and image analysis* **2014**, *24*, 400–408.
76. Osera, P.M.; Zdancewic, S. Type-and-example-directed program synthesis. *SIGPLAN Not.* **2015**, *50*, 619–630.
77. Polozov, O.; Gulwani, S. FlashMeta: A Framework for Inductive Program Synthesis. In Proceedings of the Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, New York, NY, USA, 2015; OOPSLA 2015, p. 107–126.

78. Zhang, T.; Chen, Z.; Zhu, Y.; Vaithilingam, P.; Wang, X.; Glassman, E.L. Interpretable program synthesis. In Proceedings of the Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems, 2021, pp. 1–16.
79. Feng, Y.; Martins, R.; Van Geffen, J.; Dillig, I.; Chaudhuri, S. Component-based synthesis of table consolidation and transformation tasks from examples. *SIGPLAN Not.* **2017**, *52*, 422–436.
80. Feser, J.K.; Chaudhuri, S.; Dillig, I. Synthesizing data structure transformations from input-output examples. *SIGPLAN Not.* **2015**, *50*, 229–239.
81. Polikarpova, N.; Kuraj, I.; Solar-Lezama, A. Program synthesis from polymorphic refinement types. *SIGPLAN Not.* **2016**, *51*, 522–538.
82. Chen, H.; Wu, C.; Zhao, A.; Raghothaman, M.; Naik, M.; Loo, B.T. Synthesizing Formal Network Specifications From Input-Output Examples. *IEEE/ACM Transactions on Networking* **2023**, *31*, 994–1009.
83. Ye, X.; Chen, Q.; Dillig, I.; Durrett, G. Optimal neural program synthesis from multimodal specifications. *arXiv preprint arXiv:2010.01678* **2020**.
84. Barke, S.; Peleg, H.; Polikarpova, N. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* **2020**, *4*.
85. Heule, S.; Sridharan, M.; Chandra, S. Mimic: Computing Models for Opaque Code. In Proceedings of the Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, New York, NY, USA, 2015; ESEC/FSE 2015, p. 710–720.
86. Ren, H.; Mo, W.; Zhao, G.; Ren, D.; Liu, S. Breadth First Search Based COSINE Software Code Framework Automation Algorithm. In Proceedings of the ASME Power Conference. American Society of Mechanical Engineers, 2015, Vol. 56604, p. V001T07A003.
87. Fix, S.; Probst, T.; Ruggli, O.; Hanne, T.; Christen, P. Automatic Programming As An Open-Ended Evolutionary System. *International Journal of Computer Information Systems & Industrial Management Applications* **2022**, *14*.
88. Shimonaka, K.; Higo, Y.; Matsumoto, J.; Naito, K.; Kusumoto, S. Towards automated generation of Java methods: A way of automated reuse-based programming. In Proceedings of the 2018 IEEE 12th International Workshop on Software Clones (IWSC), 2018, pp. 30–36.
89. Liu, J.; Dong, W.; Liu, B. Boosting Component-Based Synthesis with API Usage Knowledge. In Proceedings of the Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, New York, NY, USA, 2021; ASE '20, p. 91–97.
90. Liu, J.; Liu, B.; Dong, W.; Zhang, Y.; Wang, D. How Much Support Can API Recommendation Methods Provide for Component-Based Synthesis? In Proceedings of the 2020 IEEE 44th Annual Computers, Software, and Applications Conference (COMPSAC), 2020, pp. 872–881.
91. Liventsev, V.; Härmä, A.; Petković, M. Neurogenetic programming framework for explainable reinforcement learning. In Proceedings of the Proceedings of the Genetic and Evolutionary Computation Conference Companion, 2021, pp. 329–330.
92. Ryan, C.; Collins, J.J.; Neill, M.O. Grammatical evolution: Evolving programs for an arbitrary language. In Proceedings of the Genetic Programming: First European Workshop, EuroGP'98 Paris, France, April 14–15, 1998 Proceedings 1. Springer, 1998, pp. 83–96.
93. Spector, L.; Robinson, A. Genetic programming and autoconstructive evolution with the push programming language. *Genetic Programming and Evolvable Machines* **2002**, *3*, 7–40.
94. Pantridge, E.; Spector, L. PyshGP: PushGP in python. In Proceedings of the GECCO, 2017.
95. Feng, Y.; Martins, R.; Wang, Y.; Dillig, I.; Reps, T.W. Component-based synthesis for complex APIs. In Proceedings of the Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, 2017, pp. 599–612.