DOI: 10.15514/ISPRAS-2024-36(6)-4

A Decade of Advancements in Program Synthesis from Natural Language: A Systematic Literature Review

[1] *R. Ramirez-Rueda, ORCID: 0009-0004-2084-7505 <zS20000354@estudiantes.uv.mx>*
[1] *E. Benitez-Guerrero, ORCID: 0000-0001-5844-4198 <edbenitez@uv.mx>*
[1] *C. Mezura-Godoy, ORCID: 0000-0002-5386-107X <cmezura@uv.mx>*
[2] *E. Barcenas, ORCID: 0000-0002-1523-1579 <barcenas@fi-b.unam.mx>*

[1] *Facultad de Estadistica e Informatica, Universidad Veracruzana, Xalapa, Mexico.*
[2] *Universidad Nacional Autonoma de Mexico Ciudad de Mexico, Mexico.*

Abstract. Program Synthesis is the process of automatically generating software from a requirement specification. This paper presents a systematic literature review focused on program synthesis from specifications expressed in natural language. The research problem centers on the complexity of automatically generating accurate and robust code from high-level, ambiguous natural language descriptions – a barrier that limits the broader adoption of automatic code generation in software development. To address this issue, the study systematically examines research published between 2014 and 2024, focusing on works that explore various approaches to program synthesis from natural language inputs. The review follows a rigorous methodology, incorporating search strings tailored to capture relevant studies from five major data sources: IEEE, ACM, Springer, Elsevier, and MDPI. The selection process applied strict inclusion and exclusion criteria, resulting in a final set of 20 high-quality studies. The findings reveal significant advancements in the field, particularly in the integration of large language models (LLMs) with program synthesis techniques. The review also highlights the challenges and concludes by outlining key trends and proposing future research directions aimed at overcoming these challenges and expanding the applicability of program synthesis across various domains.

Keywords: program synthesis; program generation; natural language processing.

For citation: Ramirez-Rueda R., Benitez-Guerrero E., Mezura-Godoy C., Barcenas E. A. Decade of Advancements in Program Synthesis from Natural Language: A Systematic Literature Review. Trudy ISP RAN/Proc. ISP RAS, vol. 36, issue 6, 2024. pp. 59-82. DOI: 10.15514/ISPRAS-2024-36(6)-4.

Acknowledgments. This work was partially developed under the support of the National Council of Humanities, Science and Technology (CONAHCYT-Mexico) in the scope of the project "Infraestructura para Agilizar el Desarrollo de Sistemas Centrados en el Usuario" (Catedras, Ref. 3053). In addition, the authors thank CONAHCYT for the doctoral scholarship granted to the first author. We also thank the Universidad Veracruzana for the support in the development of this research.

Десятилетие достижений в синтезе программ по спецификациям на естественном языке: систематический обзор литературы

[1] *Р. Рамирес-Руэда, ORCID: 0009-0004-2084-7505 <zS20000354@estudiantes.uv.mx>*
[1] *Э. Бенитес-Гуэрреро, ORCID: 0000-0001-5844-4198 <edbenitez@uv.mx>*
[1] *К. Мезура-Годой, ORCID: 0000-0002-5386-107X <cmezura@uv.mx>*
[2] *Э. Барсенас, ORCID: 0000-0002-1523-1579 <barcenas@fi-b.unam.mx>*

[1] *Факультет статистики и информатики Университета Веракруса, Халапа, Мексика.*
[2] *Национальный автономный университет города Мехико, Мексика.*

Аннотация. Программный синтез – это процесс автоматического создания программного обеспечения на основе спецификации требований. В этой статье представлен систематический обзор литературы, посвященный синтезу программ из спецификаций, выраженных на естественном языке. Исследуемая проблематика заключается в сложности автоматического создания точного и надежного кода из высокоуровневых, неоднозначных описаний на естественном языке – барьер, который ограничивает более широкое использование средств автоматизации при разработке программного обеспечения. Для исследования этой проблемы авторы систематически изучали работы, опубликованные в период с 2014 по 2024 год, делая акцент на работы, в которых рассматриваются различные подходы к синтезу программ на основе данных на естественном языке. Обзор следует строгой методологии, включающей поисковые строки, адаптированные для сбора соответствующих исследований из пяти основных источников данных: IEEE, ACM, Springer, Elsevier и MDPI. В процессе отбора применялись строгие критерии включения и исключения, что привело к окончательному набору из 20 высококачественных исследований. Результаты показывают значительные достижения в этой области, особенно в интеграции больших языковых моделей (LLM) с методами синтеза программ. Обзор также освещает проблемы и завершается изложением ключевых тенденций и предложением будущих направлений исследований, нацеленных на преодоление этих проблем и расширение применимости синтеза программ в различных областях.

Ключевые слова: синтез программ; генерация программ; обработка естественного языка.

Для цитирования: Рамирес-Руэда Р., Бенитес-Гуэрреро Э., Мезура-Годой К., Барсенас Э. Десятилетие достижений в синтезе программ по спецификациям на естественном языке: систематический обзор литературы. Труды ИСП РАН, том 36, вып. 6, 2024 г., стр. 59–82 (на английском языке). DOI: 10.15514/ISPRAS–2024–36(6)–4.

Благодарности. Эта работа была частично выполнена при поддержке Национального совета по гуманитарным наукам, науке и технике (CONAHCYT-Мексика) в рамках проекта “Infrastructura para Agilizar el Desarrollo de Sistemas Centrados en el Usuario” (Catedras, Ref. 3053). Кроме того, авторы благодарят совет CONAHCYT за докторскую стипендию, предоставленную первому автору. Мы также благодарим Университет штата Веракрус за поддержку в проведении этого исследования.

1. Introduction

The development of a software system encompasses a detailed life cycle that includes stages such as requirement specification, design, prototyping, and testing [1]. Traditionally, this process is slow and susceptible to errors. To enhance efficiency and reduce errors, employing models at various abstraction levels, along with their mappings, can facilitate the automatic generation of code from high-level descriptions. These models, which capture the behavior and structure of the system, can be developed manually or derived from requirement specifications. Despite the benefits, the process of generating code from models necessitates establishing the models themselves, defining rules for mapping elements between models, and creating rules to generate code in the target programming language. These tasks require expert knowledge and sophisticated tools.

An alternative method is program synthesis [2], which involves generating software automatically from a requirement specification. Relying on artificial intelligence and formal methods, this approach aims to produce correct programs by formulating rules that map input specifications directly to programs, thus accelerating development. However, it is critical to acknowledge that software developed through this approach may be more prone to errors and could lack robustness.

In the realm of specifying systems, one commonly uses expressions in predicate logic, necessitating specialized expertise. To make this approach accessible not only to experts but also to end-users, specifications should ideally be articulated in a more intuitive form, such as natural language.

Our research takes as reference the work proposed by [3] and [4], which extensively review code generation with natural language, although they examine approaches that automatically generate source code from a description In natural language, we want to emphasize the areas of application, as well as to make known the types of inputs and outputs that are necessary to generate automatic code from natural language and finally analyze future trends.

To ensure the relevance of our study in this rapidly evolving domain, we consider recent advancements in natural language processing and artificial intelligence, particularly as they pertain to program synthesis. This includes the exploration of models like GPT-4 and other advanced transformer architectures to understand how they can be adapted for interpreting natural language program specifications. Moreover, we address the current challenges, such as achieving precision in interpreting complex requirements and the implications of automating code generation.

It is important to note that this work extends the paper "Program Synthesis and Natural Language Processing: A Systematic Literature Review," presented at the International Conference on Research and Innovation in Software Engineering (CONISOFT 2023). In this updated study, we expand the analysis by covering an additional five years and incorporating a new digital library (MDPI), thereby covering the last decade. Our objective is to analyze publications, identify emerging trends, and highlight opportunities for future research that were not addressed in the previous work. We selected twenty articles from major databases, including IEEE, ACM, Springer, Elsevier, and MDPI.

These studies investigate various methods of program synthesis, ranging from rule-based approaches, which employ explicit translation rules from natural language to code, to more advanced techniques that learn these rules from input-output pairs, integrating generative artificial intelligence models.

The paper is structured as follows: We begin with background information on the relevant research areas of program synthesis, natural language processing, and generative models. Next, we detail the methodology employed for the SLR, followed by a discussion of the findings. The paper concludes with a summary of the research outcomes.

2. Background

2.1 Program synthesis

In this section we discuss Program synthesis is an intriguing research domain focused on the automatic generation of programs from detailed specifications. This field is particularly valuable for creating small, complex programs that are verifiable and correct based on comprehensive specifications.

The domain is characterized by three critical dimensions [5]: the types of constraints that express user intentions, the operational search space, and the search techniques employed. User intentions can be depicted through various forms such as logical relations between inputs and outputs, demonstrations, natural language, input-output examples, or inefficient or related programs. The search space may be confined by potential program types, computational models like context-free grammars, or logical frameworks. Search techniques employed include exhaustive searches, version space exploration, machine learning, and logical reasoning.

Program synthesis can further be classified into methods such as deductive synthesis from full specifications [6], which generates programs based on probabilistic selection mechanisms. The viability of a program is determined by its alignment with specified criteria derived from its specifications. Despite its effectiveness, generating these detailed specifications is a considerable challenge and verifying them is computationally intensive.

Alternately, inductive synthesis starts with incomplete problem descriptions, which may include test cases, specified desired and undesired behaviors, input-output examples, or execution traces for particular inputs [7]. While this approach ensures correctness by construction, the creation of extensive programs remains a significant computational challenge, often requiring more effort to define a complete and correct specification than to write the program itself. Fig. 1 shows the possible program synthesis approaches.

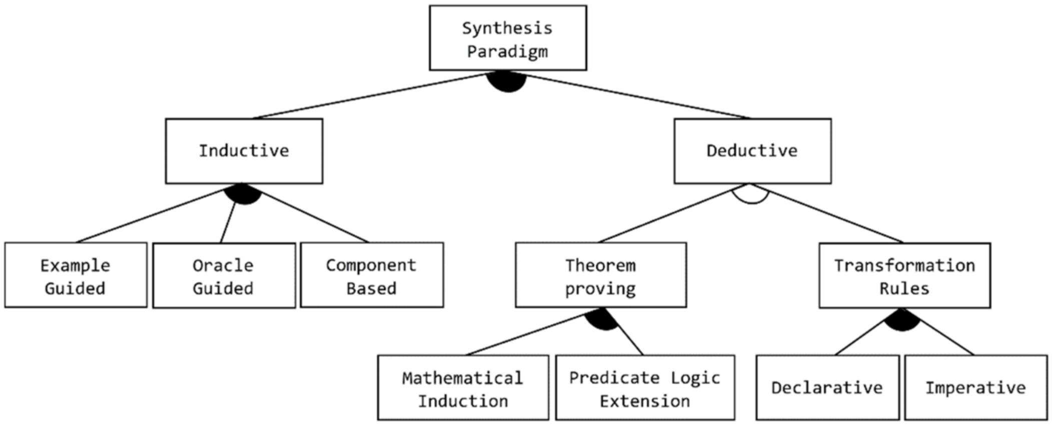


Fig. 1. Program Synthesis Paradigms [8].

Conversely, the integration of generative artificial intelligence is reshaping the software industry, not only by advancing techniques and tools but also by democratizing software development [9]. A significant obstacle in traditional program synthesis has been the requirement for complete specifications. However, modern advancements in software development are transforming this challenge by employing models capable of interpreting natural language descriptions to synthesize code across various programming languages. This transformation greatly simplifies the synthesis process and broadens access to those without specialized expertise.

A prime example is the use of large language models (LLMs), which empower non-programmers to create applications through intuitive natural language interfaces. This capability could herald a major shift in information technology education and training, with a greater emphasis on design and project management skills rather than on pure coding [10].

Unlike traditional expert systems that merely analyze or interact with existing data, program synthesis harnesses vast data sets and complex architectures to generate new and varied content. By leveraging continued advances in computing power, this approach employs deep neural networks, transformers, generative adversarial networks, and autoencoders to capture the complexity of data and effectively model high-dimensional probabilistic distributions across both specific and general domains [11].

Furthermore, by incorporating techniques that map the latent semantic space of language or images to multimedia representations in text, audio, or video, generative models can convert any type of input into a variety of output formats [12] [13]. This versatility makes generative models invaluable in numerous applications.

The extensive data access and complex architectures of these models offer unprecedented potential for content creation and transformation. Their ability to learn from diverse sources, generate various multimedia formats, and convert inputs from one format to another opens up a wide array of possibilities in multimedia generation and conversion, making these models indispensable tools in today’s technologically advanced world.

In summary, program synthesis is revolutionizing problem-solving by enabling non-experts to automate solutions without requiring deep knowledge of algorithm design and implementation [14]. Through the strategic use of various constraints, search spaces, and synthesis methods—both deductive and inductive—the field of program synthesis continues to evolve. While the need for complete specifications has historically been a barrier, recent technological advancements now allow for the use of natural language, thereby enhancing the field's accessibility and practical application.

2.2 Natural Language Processing

Artificial intelligence systems have significantly advanced the development of complex cognitive tasks. Natural Language Processing (NLP) serves as a pivotal bridge between human languages and computers, facilitating a myriad of applications [15]. Among the foundational techniques in NLP are regular expressions, which are essential for executing various practical NLP tasks.

Progress in NLP has led to sophisticated approaches for tasks such as text classification, knowledge discovery, and word recommendation. Prominent algorithms for word embedding—such as Word2Vec [16], GloVe [17], and Gensim [18] – play critical roles in these areas by capturing semantic relationships between words and enabling vector representations that are used in downstream tasks. Furthermore, deep learning-based sequence to sequence models (seq2seq) [19] have proven highly effective in machine translation tasks, facilitating the transformation of text from one language to another with high accuracy. Techniques that consider word order and linguistic elements like phonemes and sentences are instrumental in enabling inference and generating novel sentence elements, making models like Transformers especially powerful in generating human-like text [20-21].

Sequence modeling is another critical domain within NLP. Long Short-Term Memory networks (LSTM)[22] are particularly advantageous for these tasks due to their capacity to retain long-term information in a sequence. Unlike recurrent neural networks (RNN), which typically process information through tree structures in a seq2tree fashion, LSTMs incorporate bidirectional flows, thereby enhancing efficiency and performance in a variety of tasks, including speech recognition, time series prediction, and text generation[23]. More recent advancements, such as Bidirectional Encoder Representations from Transformers (BERT), further extend the capabilities of sequence models by pretraining on large corpora and fine-tuning for specific tasks, achieving state-of-the-art results in many NLP benchmarks. In conclusion, NLP technologies enable the seamless integration of natural language understanding within systems, thereby meeting diverse end-user needs and expanding the scope of possible applications. These advancements have profound implications not only in traditional applications like translation and sentiment analysis but also in emerging areas such as conversational AI, content generation, and human-computer interaction, where the ability to understand and generate natural language is crucial. In the next section we will analyze the method we used for this research.

3. Method

To explore diverse perspectives and support the research presented in this work, we adopted the systematic literature review methodology proposed by [24], while also incorporating recommendations from [25–28]. This methodology provides a rigorous framework for exploring the synergies between program synthesis and natural language processing, thereby enriching the research landscape and informing future studies in these areas. Additionally, this research considers fundamental aspects such as requirements, models, input-output formats, and evaluation metrics.

The systematic mapping we employ follows a structured approach that includes Research Questions, Search String, Data Sources, Selection Criteria, and Quality Assessment.

3.1 Research Questions

The objective of our systematic literature review is to obtain a comprehensive understanding of the key components involved in program synthesis and code generation, particularly through the lens of Natural Language Processing. We aim to reveal the mechanisms underlying automatic program generation and identify areas needing further research to thoroughly understand the context and advantages of program generation via synthesis.

The research questions formulated for this study are designed to systematically dissect these aspects:

Q1. What are the application areas?

Q2. What are the inputs used to synthesize a program?

Q3. What are the outputs generated from the program synthesizer and how are they used?

Q4. What type of synthesis is used?

3.2 Search String

We defined a search string aimed at capturing the intersection of key research domains: [((“Program” OR “Code”) AND (“Synthesis” OR “Generation”)) AND (“Natural Language Processing” OR “NLP”). This string was used to ensure that all pertinent literature was considered. Depending on the database, a general search string was defined and adapted to each search engine.

3.3 Data Sources

Five major data sources were selected to conduct a comprehensive search for literature related to program synthesis and natural language processing. The sources include ACM Digital Library, IEEE Xplore, Springer, Elsevier, and MDPI. These platforms were chosen for their extensive repositories of scientific papers and their relevance to the fields under study.

3.4 Selection Criteria

To ensure a focused and relevant data collection process, several inclusion and exclusion criteria were meticulously applied:

The exclusion criteria eliminated other types of documents, such as unpublished works, books, courses, newspapers, and master’s and doctoral theses.

The inclusion criteria considered only journal articles and conference proceedings published between 2014 and 2024, which allowed us to capture the most recent advances in the field. The search parameters were carefully established to filter data by titles, abstracts, and keywords of journal articles and conference proceedings that met the inclusion criteria. This methodological rigor ensured the collection of the most relevant and beneficial data for our systematic review.

3.5 Quality Assessment

Each study was evaluated using the criteria from the Center for Reviews and Dissemination (CRD) of the University of York, as well as the Database of Abstracts of Reviews of Effects (DARE) [29]. The criteria are based on three quality assessment (QA) questions:

QA1. Are sufficient details about the individual included studies presented?

QA2. Does it provide evidence to answer the research questions for this systematic review?

QA3. Is it a referenced study?

The questions were scored as follows:

- **QA1:** Y (yes), presents sufficient details in the study, P (Partially), presents information partially; N (no), does not have details and cannot be easily inferred.
- **QA2:** Y (yes), The authors based their research in such a way that they included appropriate strategies, identified and made reference to all the journals that addressed the topic of interest, and the study answered all the research questions; P (Partially), The study only partially answered the research questions, N (no), They did not answer the research

questions, and lacked adequate context.

- **QA3:** Y (yes), They are highly referenced studies, P (Partially), The study has a certain number of citations, N (no), The study does not have citations.

The scoring procedure was Y = 1, P = 0.5, N = 0 where information is not specified. Consequently, the possible score that could be obtained for assessing the quality of a primary study was in the range of 0 to 3 points. In this sense, the articles considered had to achieve a rating of 1.5 at least. In the next chapter we will discuss the results obtained.

4. Results

We thoroughly analyzed the full texts of 20 articles that met our selection criteria. Below, we present the initial results obtained from these articles, providing a summary of the studies included in our systematic review. This summary aims to establish a foundational understanding of the scope and impact of the research conducted in the field.

Following the initial overview, we present the quality evaluation of each article to ensure the reliability and validity of the reported findings. This evaluation was essential to maintaining the integrity of our systematic review.

Next, we will demonstrate how we answered the specific research questions posed at the beginning of our study. This analysis will help us identify key trends, gaps in current research, and potential areas for future research, aligning our findings with the overall objectives of our research.

4.1 Summary of the Studies

In this section we present a summary of the works examined, with the goal of answering the research questions formulated. The systematic search across the specified data sources initially yielded a total of 680,924 articles. By applying the inclusion criterion of publication years from 2014 to 2024 and the focus on journals and conferences, the results were refined to 401,104 articles. Further application of criteria related to the research topic narrowed this down to 20 articles directly relevant to the research objectives. The methodology applied in this search is summarized in Fig. 2.

Among these 20 relevant articles, 6 (30%) were published in journals, while 14 (70%) appeared in conference proceedings, as depicted in Fig. 3. We observed a clear trend of increasing publications up until 2021 and 2022, followed by a sharp decline in 2023. This trend could potentially reverse in 2024, influenced by emerging developments in the field.

The research questions and corresponding answers presented in this study have significant implications only 20 papers were directly relevant to the research questions under consideration, as detailed in Table 1.

Among the data sources, the ACM Digital Library demonstrated the highest precision, with an accuracy rate of 0.00215% in yielding relevant articles. A detailed distribution of relevant articles from each source is presented in Table 2. This allows for a discussion on how each identified source contributes to the understanding of program synthesis and natural language processing as outlined by the objective of this paper.

4.2 Quality evaluation

The results of the application of the quality evaluation show that, on average, the studies had a score of 2 points, with the exception of studies S1 and S2 that obtained a score of 1.5. As it can be seen in Table 3, the articles are of good quality and relevant to the investigation.

4.3 Application areas (Q1)

Program synthesis has become a pivotal tool in various domains, demonstrating significant utility, particularly in software engineering. Below we can see the application areas identified in the selected papers:

- **Program Synthesis for Education:** Enhancing learning and teaching in logic and programming through the generation of code from natural language problems.
- **Program Synthesis for Software Development:** Improving developer efficiency and productivity by automatically generating code, API calls, and optimizing program synthesis tools.
- **Program Synthesis for Robotics and AI:** Simplifying the programming of complex tasks in robotics and generating solutions for various NLP problems through advanced AI models.

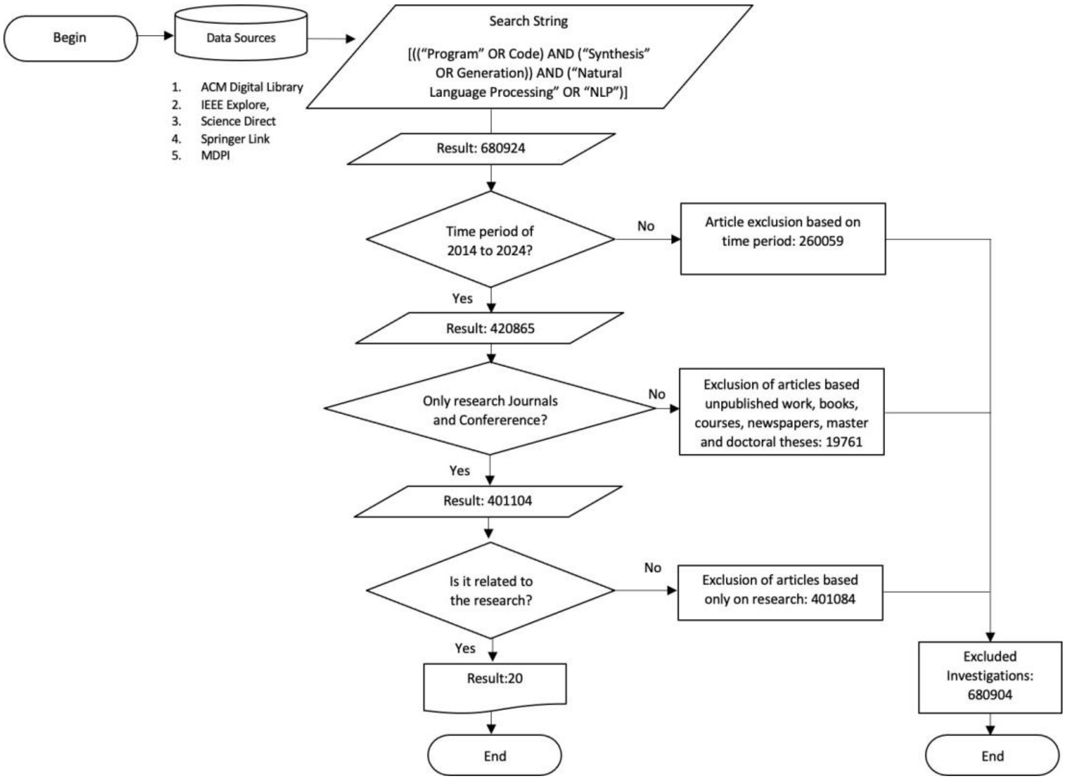

Begin
Data Sources
1. ACM Digital Library
2. IEEE Explore,
3. Science Direct
4. Springer Link
5. MDPI
Search String
[((“Program” OR Code) AND (“Synthesis” OR Generation)) AND (“Natural Language Processing” OR “NLP”)]
Result: 680924
Time period of 2014 to 2024?
No
Article exclusion based on time period: 260059
Yes
Result: 420865
Only research Journals and Conference?
No
Exclusion of articles based unpublished work, books, courses, newspapers, master and doctoral theses: 19761
Yes
Result: 401104
Is it related to the research?
No
Exclusion of articles based only on research: 401084
Yes
Result:20
End
Excluded Investigations: 680904
End

Fig. 2. Steps for the extraction of relevant documents from the selected sources.

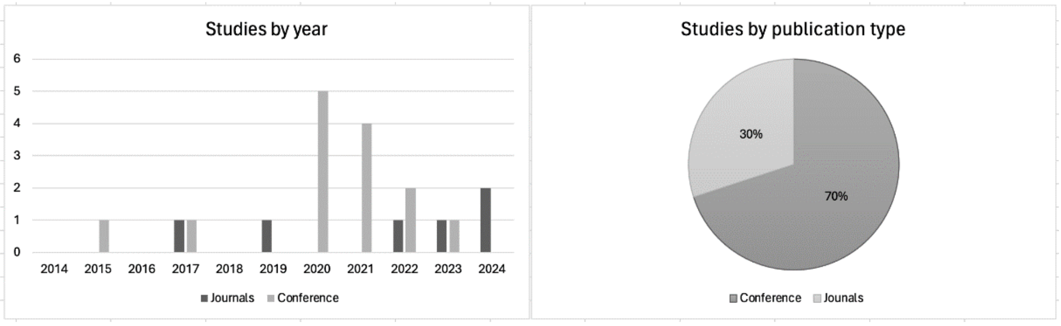


Fig. 3. Type of articles according to inclusion criteria and average of article according to year of publication.

Table 1. Selected Papers.

Title	Author(s)	Year	Ref
S1 Domain specific program synthesis	Archana, P., Harish, P. B., Rajan, N., P, S., and Kumar, N. S.	2021	[30]
S2 Collective intelligence for smarter neural program synthesis	Daiyan. W, Wei. D, and Yating. Z.	2020	[31]
S3 Generating context-aware API calls from natural language description using neural embeddings and machine translation	Phan, H., Sharma, A., and Jannesari, A.	2021	[32]
S4 HISyn: Human Learning-Inspired Natural Language Programming	Nan, Z., Guan, H., and Shen, X.	2020	[33]
S5 Interactive Program Synthesis by Augmented Examples	Zhang, T., Lowmanstone, L., Wang, X., and Glassman, E. L.	2020	[34]
S6 Deep nlp-based co-evolvement for synthesizing code analysis from natural language	Nan, Z., Guan, H., Shen, X., and Liao, C.	2021	[35]
S7 Interactive synthesis of temporal specifications from examples and natural language	Gavran, I., Darulova, E., and Majumdar, R.	2020	[36]
S8 Programming bots by synthesizing natural language expressions into API invocations	Zamanirad, S., Benatallah, B., Barukh, M. C., Casati, F., and Rodriguez, C.	2017	[37]
S9 Egeria – A Framework for Automatic Synthesis of HPC Advising Tools through Multi-Layered Natural Language Processing	Hui. G, Xipeng. S, and Hamid. K.	2017	[38]
S10 Interactive Synthesis using Free-Form Queries	Tihomir. G and Viktor. K.	2015	[39]
S11 Jigsaw – Large Language Models meet Program Synthesis	Naman. J, Skanda. V, Arun. I, Nagarajan. N,	2022	[40]
S12 Many-objective Grammar-guided Genetic Programming with Code Similarity Measurement for Program Synthesis	Ning. T, Anthony. V, and Takfarinas. S.	2023	[41]
S13 Program Synthesis Through Learning the Input-Output Behavior of Commands	Sihyung. L, Seung. Y. Nam, and Jiyeon. K.	2022	[42]
S14 Assessing Similarity-Based Grammar-Guided Genetic Programming Approaches for Program Synthesis	Ning. T, Anthony. V, Takfarinas. K.	2022	[43]
S15 Generative Model for NLP Applications based on Component Extraction	Bhardwaj, P. Khanna, S. Kumar, and Pragya.	2020	[44]
S16 Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis	Kia. R, Mohammad. R, Summit. G and Vu. L.	2021	[45]
S17 Prompt Problems: A New Programming Exercise for the Generative AI Era	Amarouche, B. A. Becker, and B. N. Reeves.	2024	[46]
S18 Automatic Acquisition of Annotated Training Corpora for Test-Code Generation	Magdalena. K and John. D. K.	2019	[47]
S19 Natural Language Generation and Understanding of Big Code for AI-Assisted Programming	Man-Fai, W. Shangxin. G, and Ching-Nam. H.	2023	[48]
S20 Effectiveness of ChatGPT in Coding: A Comparative Analysis of Popular Large Language Models	Carlos. E. C, Mohammed. N. A, and R. K.	2024	[49]

Table 2. Total items extracted.

Data Sources	Result	Useful Articles	Accuracy
IEEE	384457	9	0.00002%
ACM	2781	6	0.00215%
SPRINGER	494	1	0.00202%
ELSEVIER	2708	1	0.00036%
MDPI	10664	3	0.00028%

Table 3. Evaluation of the quality of the studies.

Study	QA1	QA2	QA3	Total Score
S1	P	Y	N	1.5
S2	P	Y	N	1.5
S3	Y	Y	P	2.5
S4	Y	Y	P	2.5
S5	Y	Y	Y	3
S6	Y	Y	P	2.5
S7	Y	Y	P	2.5
S8	P	Y	P	2
S9	Y	Y	N	2
S10	P	P	Y	2
S11	Y	Y	Y	3
S12	P	Y	P	2
S13	Y	Y	N	2
S14	Y	Y	P	2.5
S15	Y	Y	Y	3
S16	Y	Y	Y	3
S17	Y	Y	N	2
S18	Y	Y	P	2.5
S19	Y	Y	Y	3
S20	Y	Y	N	2

Education:

- [30] focuses on the use of program synthesis to solve propositional logic problems in an educational context, emphasizing the generation of code from problems described in natural language. This approach is ideal for teaching and learning in fields related to logic and programming.
- [46] introduces “Prompt Problems” to teach students how to write effective prompts for generating code using large language models (LLMs), helping them develop skills in formulating natural language prompts that produce functional code.

Software Development:

- [31] integrates collective intelligence and bio-inspired algorithms to optimize accuracy in code generation from user intents.
- [32] improves developer efficiency by automatically generating API calls based on natural language descriptions and the context of the surrounding code.
- [33] enhances code generation through natural language understanding, specifically aimed at software development.
- [34] develops interactive program synthesis tools, particularly for creating regular expressions, using augmented examples to clarify user intent and facilitate automatic code generation.
- [35] evaluates the effectiveness of ChatGPT and other large language models in code

generation tasks, highlighting their utility as programming assistance tools.

- [37] focuses on the development of interactive program synthesis tools for creating regular expressions using augmented examples.
- [38] creates a platform called BotBase that allows the transformation of natural language expressions into API invocations, facilitating bot programming.
- [39] creates advisory tools for optimizing high-performance computing programs using natural language processing.
- [40] develops a support tool for IDEs that generates Java code snippets based on free-text queries combining English and code.
- [41] uses pre-trained language models like GPT-3 and Codex to generate code from natural language descriptions, optimized for complex APIs like Python Pandas.
- [42] employs grammar-guided genetic programming for program synthesis, using multiple code similarity measures to improve accuracy in generating code from textual descriptions and input/output examples.
- [43] evaluates and improves the use of grammar-guided genetic programming for program synthesis, guiding the evolutionary process with code similarity measures.
- [45] combines pretrained language models with component-based synthesis techniques to generate programs from natural language descriptions and specific examples, particularly for generating regular expressions and CSS selectors.
- [47] focuses on the automatic creation of annotated data sets to generate automated test cases from quasi-natural language descriptions, using machine learning and machine translation techniques.
- [48] reviews the use of large language models trained with Big Code for various AI-assisted programming tasks, including code generation, completion, translation, refinement, summarization, defect detection, and clone detection.

Robotics:

- [36] facilitates task specification for robots using linear temporal logic (LTL) from natural language examples and interactions, simplifying the programming of complex and specific tasks in robotics applications.

Artificial Intelligence:

- [44] creates a generative model for natural language processing (NLP) applications, extracting meaningful components from case studies to address problems such as reading text, interpreting speech, measuring sentiment, and determining important parts, generating optimized solutions for different NLP problems.

4.4 Inputs used to synthetize a program (Q2)

In this section, the primary focus is to identify the different types of inputs that will be processed by the synthesizer programs. The exploration of the literature has allowed us to identify how these works take natural language expressions and synthesize examples based on the user's intended purpose, using different techniques to achieve the various objectives proposed by the authors. The results of relevant articles are detailed in Table 4.

The study by [30] introduces a tool for end-user programming designed to simplify the programming process and enable programmers to focus more on the core logic of the program. This tool removes the need to deal with language syntax and other domain-specific aspects. User input is provided in the form of a propositional verbal problem, which consists of facts, conditionals, and questions, thereby establishing the basis for a learning approach.

[31] centers on the automatic generation of source code from various user intents. The authors utilized natural language task descriptions as inputs, enabling the identification of web page tags that align with these characteristics. This study demonstrates the versatility of user intent expression and represents significant progress in solving programming tasks based on natural language descriptions, requiring minimal information about the target program.

Table 4: Types of inputs from different examples of program synthesis using NL.

Types of inputs	**Articles**
Verbal problems (Query)	[30]
Natural language task descriptions	[31], [48], [46]
Sentences and a part of the surrounding context.	[32], [39]
Natural language (query)	[33], [38], [45]
Description of a method in NL	[34], [47]
NL queries based on dependency structure	[35]
Specific descriptions	[36], [40]
Short description of a specification	[37], [42]
High-level specifications	[41]
Textual Problem Descriptions	[43], [44]
Programming Prompts	[49]

Similarly, the study in [32] employed a unique method involving the pairing of an instruction sentence with a section of corresponding code. The input consisted of a natural language user intent and a drafted method, using the Java language. A method name generator was then employed to extract tokens and variable names from natural language descriptions and adjacent code tokens, thus predicting potential method names.

The research presented in [33] adopts an approach driven by natural language understanding. The input consists of a natural language query containing a list of synonyms, named entities, and a dictionary of prepositions. This method reduces the need for extensive labeled examples, thereby freeing users from the task of gathering examples and facilitating natural language programming, especially in domains where labeled examples are difficult to obtain. The study in [34] investigates the use of natural language descriptions of methods as input to improve concrete word recognition. The researchers introduce a semantic analyzer that links variables to specific operational information, thus describing the method's particular behavior, parameter name, and return value information.

The field of code analysis presents numerous complexities, especially those associated with data types and operations. The research in [35] introduces a tool that significantly mitigates these complexities. This tool leverages natural language queries, drawing upon dependency structures in language, to interpret the code. The tool specifically automates the analysis of asymmetric binary relations between words in a sentence, such as subordinate words and their dependencies. In other words, it uses the syntactic structures of natural language to build a semantic understanding of code. This approach not only aids in extracting the core meaning of the code but also makes the process more comprehensible and accessible to programmers.

Simultaneously, natural language descriptions and programming by example have emerged as "user-friendly" alternatives for specifying complex tasks. [36] addresses these issues by using specific descriptions as inputs. This method generates grammatical rules for producing parseable commands, thus facilitating the straightforward specification of complex, repetitive tasks.

Lastly, although modern bot creation systems detect user intent, they require considerable development and configuration effort for each use case. [37] introduces a tool that uses a concise specification description as input, assisting in the generalization of critical tasks in the program generation process.

The inputs of Egeria [38] include optimization guides or other domain-specific documents relevant to HPC. Additionally, user queries or performance profiling reports can be fed into the synthesized advising tool to receive specific optimization advice.

The inputs used by the synthesis tool proposed in [39] include free-form queries composed of a mixture of English and Java code. These queries can describe desired functionalities or operations in natural language, possibly combined with partial code snippets. The system also incorporates context from the developer's current work in the IDE, such as the cursor position and existing code, to better understand and generate the appropriate code fragments.

Jigsaw [40] accepts multi-modal inputs for synthesizing programs. Users can input their intent or requirements in natural language and also include test cases, input/output examples. These are used to further specify the intended functionality of the code, helping to refine the synthesis process and ensure that the generated code meets the user's needs.

The inputs for synthesizing a program using MaOG3P [41] include high-level specifications or textual descriptions of the desired functionality of the program. Particularly, input/output examples specify what the program should produce given certain inputs, helping to guide the genetic programming process to evolve correct and efficient code.

The inputs for the program synthesis system proposed by [42] take the form of short descriptions of specifications. The system understands the available commands and their syntax, which guide the synthesis process. For instance, the system uses examples of desired inputs to learn and generate the corresponding program.

The inputs used in [43] include textual problem descriptions that describe a programming task provided in natural language, grammatical specifications, such as a defined grammar that dictates the syntax of the programming language in which the programs are developed, and similar code that is used to evaluate the suitability of evolved programs against a target source code, improving the relevance of the generated programs for the given problem descriptions.

The inputs for the NLP generative model discussed in [44] take the form of a Textual Problem Description, which is a description provided in natural language that outlines the problem to be solved by the model. These descriptions are extracted from case studies that identify significant components relevant to the problem being addressed.

The inputs for synthesizing programs in [45] take the form of natural language queries. For example, students craft prompts in natural language that describe the desired functionality or outcome of a program. This kind of input helps define the problem that needs to be solved by the generated code, guiding the LLM towards appropriate solutions.

The input for the multi-modal synthesis approach described in [46] is a Natural Language Description, which is a broad, often ambiguous description of a desired functionality. This kind of input provides a specification of how the desired code should function.

The inputs of the synthesis process described in [47] are descriptive method names, which are extracted from source code and are used as natural language descriptions of the functionality of the code, and also function bodies that are aligned with the method names to form a parallel text code corpus.

The inputs used in [48] are natural language descriptions, which describe the desired functionality in natural language, and also existing code fragments that serve as context or examples for the desired operations.

In [49] the inputs used to synthesize programs are programming prompts, which describe what the generated code should accomplish, as well as code examples, that can be used to guide the AI in generating appropriate code structures.

4.5 Outputs generated from the program synthesis (Q3)

Program synthesis offers flexibility by utilizing incomplete specifications, regardless of the specific approach employed, to generate code. The objective is to achieve a degree of final completeness in the produced output. However, it is important to note that the generated output may not always align with the end user's expectations.

[30] leverages postfix expressions (Boolean Logic) to establish a foundation for a domain-independent learning approach to problem-solving via program synthesis concepts. This process enables users, particularly programmers, to streamline their efforts by focusing on the core logic of the program, thereby mitigating concerns about language syntax and other domain-specific elements. Given the input "*Did Mary and Ram go to school?*", the output is "*Cannot be determined / True*".

The development of large and complex software projects requires a workforce trained in the fundamental structures of the programming languages they use. One potential approach to automate this process is the generation of a common keyword list. In this scenario, programmers need not memorize the keyword vocabulary or understand their exact implementation to write a program in the given language. For instance, a list of expected method names could be derived from a method description with surrounding code [32]. For example, for the input "*return random number with max value iterationWeight for Random*", the output would be new "*Random().nextInt(iterationWeight)*".

Alternatively, understanding how programmers code is a complex process that demands practical solutions. By deeply processing programmers' intentions and API documents written in natural language, it is possible to leverage a profound understanding through program synthesis tailored for this specific purpose. This approach circumvents the need for a large number of labeled examples, thus alleviating the user's task of collecting or generating examples. It also significantly impacts traditional methods. For instance, from the input "*Find statements whose init portion declares a single variable which is initialized to the integer literal 0*", the following code (in a DSL) is generated:

```
forStmt(
    hasLoopInit(
        declStmt(
            hasSingleDecl(
                varDecl(
                    hasInitializer(
                        integerLiteral(
                            equals(0))))))))
```

Code library functions have significantly increased developers' programming efficiency. They do so by simplifying constraint generation and accelerating constraint resolution through the creation of complete code based on constraint models of Java classes [34]. A pertinent example is a code fragment in a tree structure, as shown below:

```
(define-fun result () Int (- 1))
(define-fun this ()
    (Seq String) (seq.unit ""))
(define-fun or () String "")
```

This example includes encapsulated functions that streamline and speed up constraint generation through the use of generated constraint models.

Concurrently, attaining high software quality controls is a complex task. It requires support from various program optimizations, software debugging, security measures, and more. Therefore, code analysis in the early stages of development can provide developers with various preemptive options [35]. Such an approach employs "final comparison expressions" that originate from specific natural language descriptions and assist general programmers in conducting automated program analysis.

For instance, given the input "*Find all C++ call expression of the C++ method named string1*", the generated output expression in the form of an AST is:

```
cxxMemberCallExpr(
    callee(
        cxxMethodDecl(
            hasName(string1))))
```

The correct use of specifications often poses a challenge to non-expert users. Therefore, providing an output that illustrates a synthesized specification derived from an example and a natural language description can significantly enhance the accuracy of the synthesis method. Furthermore, it paves the way for the generalization of synthesized tasks to other unseen tasks [36]. For instance, for the expression "*step into water and then visit (6, 4)*", it is possible to obtain an LTL specification as "*step into water and then visit (Num, Num)*".

Undeniably, there are numerous endeavors aimed at refining the process of automatic code generation. Each study provides a perspective on how productivity in development can be enhanced. One increasingly popular approach is the use synthesize API calls from expressions in NL. To fully harness the potential of this approach, [37] propose a tool designed to foster the development of intuitive software solutions. This tool bridges the gap between user needs, expressed in natural language, and API invocations capable of satisfying these needs. An example is: synthesize API calls from expressions in NL

```
<url:https://api.yelp.com/v2,
    path:/search parameters:
    term=[italian,cafes],
    location=[sydney.opera_house]>
```

The outputs generated by Egeria [38] include an advising tool that provides a list of essential rules extracted from the input documents. This tool also serves as a question-answer agent that offers specific optimization suggestions based on user queries or performance profiling reports. Fig. 4 shows an example rule that is used to guide programmers in optimizing code more effectively without needing to manually sift through extensive documentation.

```
if(tx % 2 == 0 && ty % 2 == 0)
    out[tx * width + ty] = 2.0 * in[tx * width + ty]/sum;
else if(tx % 2 == 1 && ty % 2 == 0)
    out[tx * width + ty] = in[tx * width + ty]/sum;
else if(tx % 2 == 1 && ty % 2 == 1)
    out[tx * width + ty] = (-1.0) * in[tx * width + ty]/sum;
else
    out[tx * width + ty] = 0.0f;
```

Fig. 4. The Optimized Block [38].

The outputs generated by [39] are Java code fragments that respect Java syntax, type, and scoping rules, as well as conform to common usage patterns derived from a statistical analysis of existing code. These code fragments are presented to the developer within the IDE, offering several ranked suggestions that the developer can choose from. The primary use of these outputs is to insert appropriate code snippets into the developer's project, helping to bridge the gap between a high-level concept expressed in natural language and executable Java code.

The output of Jigsaw [40] is executable code that matches the user's specified intent and passes given test cases. Fig. 5 shows code that is generated after processing through a series of program analysis and synthesis techniques, which include correcting common errors detected in the initial outputs from pre-trained language models (PTLMs) like GPT-3 or Codex. The generated code helps programmers quickly implement solutions and focus on higher-level design and problem-solving tasks rather than the nuances of specific API calls or syntax correctness.

The outputs generated by MaOG3P [41] are executable code snippets that meet the requirements specified through the input descriptions and examples. These outputs are used to automate coding tasks, reduce development time, and improve the efficiency of the programming process. By synthesizing code that satisfies both the syntactic and semantic correctness, the generated programs help developers by providing ready-to-use code snippets that can be integrated into larger projects or used as standalone solutions.

```
Code Before                                     Code After
out=data[data.index.isin(test.index)]           out=data[~data.index.isin(test.index)]
df=df[df[‘foo’]>70)|df[‘foo’]<34]               df=df[(df[‘foo’]>70)|(df[‘foo’]<34)]
out=df.iloc[0,”HP”]                             out=df.loc[0,”HP”]
dfout=df1.append(df2,ignore_index=True)         dfout=df1.append(df2)
dfout=dfin.duplicated()                         dfout=dfin.duplicated().sum()
train=data.drop(test)                           train=data.drop(test.index)
dfin=dfin[“A”].rolling(window=3).mean()         dfin[“A”]=dfin[“A”].rolling(3).mean()
dfout=dfin[(x<40)|(y>53)&(z==4)]                dfout=dfin[((x<40)|(y>53))&(z==4)]
```

Fig. 5. Applications (Code After) of learned transformations on code snippets produced by PTLM (Code Before) [40].

The output from this system [42] is an executable program that conforms to the specifications derived from the input-output examples provided. These programs can then be used directly within software applications, helping to automate tasks or improve software functionality with minimal human coding effort.

The outputs from the program synthesis approach proposed in [43] are executable pieces of code that align with user-defined specifications and grammar rules. Fig. 6 shows programs that are evaluated for similarity against target codes to ensure that they meet the specified requirements. This can be used in Software Development to automate or speed up the development process by providing ready-to-use code snippets that fit the user’s intent. Finally, this is an example of teaching tools to demonstrate various programming techniques and solutions.

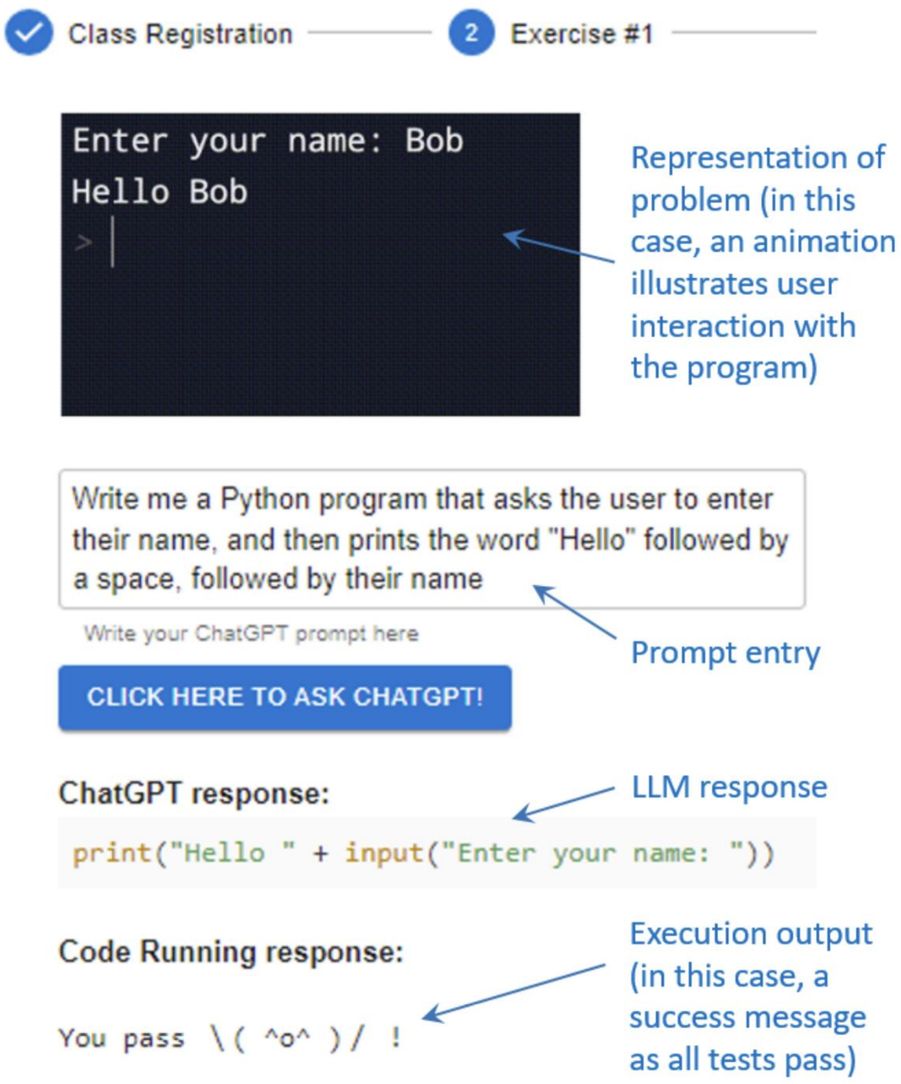
Problem	Textual Description	# Input/Output Pair	
		Training	Testing
Number IO	Given an integer and a float, print their sum.	25	1000
Median	Given 3 integers, print their median.	100	1000
Smallest	Given 4 integers, print the smallest of them.	100	1000

Fig. 6. Representation of target programs [43].

The outputs of the NLP generative model [44] are optimized solutions for NLP tasks. The model generates solutions that address specific NLP-related problems like speech interpretation, sentiment analysis, and text processing and adapted responses, because the system uses the outputs to adapt its responses based on the input it receives, making it suitable for interactive applications such as virtual assistants.

In the case of [45], the outputs are generated code based on prompts provided. A LLM is used to generate code that attempts to solve a specified problem, and then the generated code is evaluated against test cases to determine its correctness. This process aids in learning by providing immediate feedback on the effectiveness of the prompt and the functionality of the code.

The outputs generated in [46] are executable code snippets that precisely match the combined specifications provided by the natural language descriptions and the examples. Fig. 7 shows how that works. The synthesized programs are used in software development to automate coding tasks, ensuring that the generated code meets both broad functional requirements and specific operational details.


Class Registration
2
Exercise #1
Representation of problem (in this case, an animation illustrates user interaction with the program)
Write your ChatGPT prompt here
Prompt entry
CLICK HERE TO ASK CHATGPT!
ChatGPT response:
LLM response
Code Running response:
Execution output (in this case, a success message as all tests pass)

```
Enter your name: Bob
Hello Bob
> |
```

Write me a Python program that asks the user to enter their name, and then prints the word "Hello" followed by a space, followed by their name

```
print("Hello " + input("Enter your name: "))
```

```
You pass \( ^o^ )/ !
```

Fig. 7. Interface layout for a Prompt Problem within the web-based Promptly tool [46].

The outputs generated [47] are code fragments. They consist of test function names (as natural language descriptions) aligned with their respective function bodies (as code). These examples are compiled and semantically relevant test cases generated by machine learning models trained on the synthesized corpora. The goal is automating the creation of unit tests, reducing the time and effort required for manual test case development.

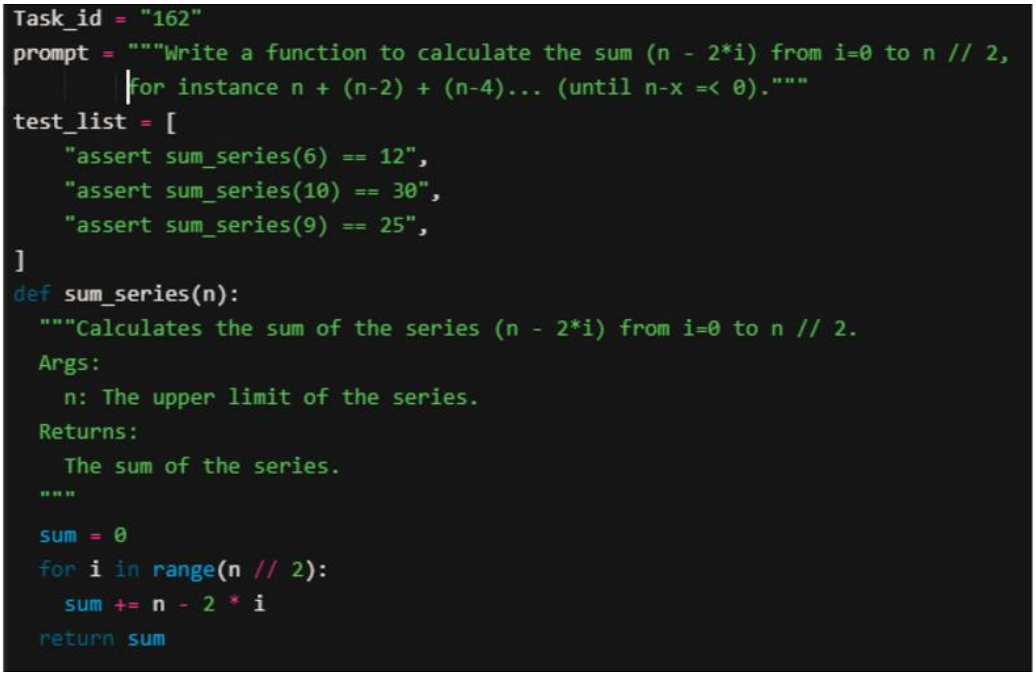
The outputs generated by the program synthesizer in [48] include executable code, which can be run directly or integrated into larger software projects. The study also allows generating descriptions in natural language or summaries of code blocks, useful for documentation. These results are used to improve productivity, improve code quality, help maintain and document code, and ensure the reliability and security of software systems.

The outputs from the program synthesizer presented in [49] are executable code within a software environment, as seen in Fig. 8. These results are used to directly implement functional requirements in software projects, automate routine coding tasks, and improve overall software quality and reliability through improved error detection and resolution capabilities.

4.6 Type of program synthesis used (Q4)

Program synthesis employs distinct methods, primarily classified into deductive and inductive synthesis, each tailored to specific aspects of programming from formal specifications.

Deductive Program Synthesis involves formally deriving a program from a given specification, treating the process as a theorem-proving task. It necessitates demonstrating that a program's output satisfies the conditions set out in the specification, a process confirmed through theorem proof. This approach requires detailed, often complex specifications and is particularly useful when correctness is paramount. In practice, deductive synthesis involves the use of formal methods to ensure that every step in the program construction adheres strictly to the logical constraints outlined in the specification. Notable works in this domain include [35], [34], and [37], which explore various applications of theorem proving in program synthesis, demonstrating the effectiveness of this approach in generating highly reliable software.



```
Task_id = "162"
prompt = """Write a function to calculate the sum (n - 2*i) from i=0 to n // 2,
         for instance n + (n-2) + (n-4)... (until n-x =< 0)."""
test_list = [
    "assert sum_series(6) == 12",
    "assert sum_series(10) == 30",
    "assert sum_series(9) == 25",
]
def sum_series(n):
  """Calculates the sum of the series (n - 2*i) from i=0 to n // 2.
  Args:
    n: The upper limit of the series.
  Returns:
    The sum of the series.
  """
  sum = 0
  for i in range(n // 2):
    sum += n - 2 * i
  return sum
```

Fig. 8. Example of outputs obtained on a data set of basic Python problems (MBPP) [49].

Inductive Program Synthesis, in contrast, starts from incomplete problem descriptions, which might include test cases, characteristics of desired and undesirable software behaviors, input-output examples, or computational traces. This approach encompasses several methodologies that aim to generalize from these examples to produce a program that satisfies the specification in a broader sense:

- *Genetic Programming and Incremental Evolution*, as exemplified by [30] where the synthesis process evolves programs iteratively, optimizing them to better fit the examples provided.
- *Counterexample-Guided Inductive Synthesis*, explored in studies such as [32] and [36], refines candidate programs by iteratively correcting them based on counterexamples, thus gradually improving their correctness.
- *Neural Program Synthesis*, with key contributions from [33] and [31], leverages deep learning models to synthesize programs from natural language or other high-level inputs, demonstrating significant advancements in automating complex programming tasks.

Egeria [38] utilizes an unsupervised, multi-layered design leveraging NLP techniques. Although not explicitly categorized, its synthesis approach suggests inductive reasoning through optimization based on general guidelines and specific user queries.

Jigsaw [40] integrates inductive synthesis with corrective transformations, initially using pre-trained language models for generating code snippets from natural language inputs, followed by corrective transformations to ensure accuracy, blending inductive learning with deductive refinements.

MaOG3P [41] and the approach outlined in [42] emphasize inductive synthesis through genetic programming and machine learning, respectively, focusing on evolving programs to meet specific input-output behaviors based on learned patterns.

The synthesis methodologies in [44] and [45] also follow inductive approaches, generalizing from specific examples to create applicable solutions across new scenarios.

Lastly, the approaches in [47], [48], and [49] exemplify the inductive synthesis prevalent in AI-assisted programming, where large datasets of code are used to predict and generate new code segments, demonstrating how modern AI tools, like ChatGPT, generalize from extensive training data to produce functional programming solutions.

This study concludes with an examination of [46], which combines inductive and deductive elements. The process starts with PTMs generating initial code candidates, followed by a Component-Based Synthesis (CBS) approach that deductively constructs the final program, ensuring it meets the provided examples through systematic component assembly and refinement. In the next chapter we present our main discussions of the study.

5. Results discussion

In this section the results of this systematic literature review reveal both the progress and ongoing challenges in the field of program synthesis, particularly when interfacing with natural language processing (NLP). The analysis of 20 selected studies highlights several key trends and areas of focus that have emerged over the past decade, also the systematic literature review on program synthesis and natural language processing (NLP) reveals significant advancements and emerging trends in this field. A key finding is the increasing integration of advanced artificial intelligence models, especially large language models (LLMs), which have demonstrated remarkable capabilities in interpreting natural language specifications and generating executable code.

This development is democratizing software development, allowing users with little or no programming experience to create functional applications using natural language instructions. The review also highlights the evolution of program synthesis methodologies, which have transitioned from rule-based approaches to more sophisticated techniques that leverage machine learning and genetic programming. These modern techniques can learn from input-output examples and user interactions, thus improving the accuracy and efficiency of code generation. However, significant challenges remain, such as achieving high precision in interpreting complex natural language requirements and ensuring responsible AI practices to guarantee the reliability of the generated code.

On the other hand, ambiguity in natural language specifications and the scalability of program synthesis systems represent crucial challenges in automatic code generation. Ambiguity, inherent in natural language, can lead to multiple interpretations of the same instruction, making it difficult to correctly understand and translate the user’s intentions into executable code. To mitigate this problem, it is necessary to develop techniques that effectively disambiguate specifications, using contextual models and interactive visualization tools. On the other hand, scalability is essential for these systems to be able to handle complex tasks and large volumes of data without losing performance. This requires the implementation of optimizations such as parallel processing and model compression, ensuring that systems can adapt to various domains and contexts without compromising the quality of the generated code.

Finally, the potential applications of program synthesis go beyond traditional software development. In the educational field, program synthesis tools are used to teach logic and programming concepts, automatically generating code from problem descriptions provided in natural language, making programming more accessible to a broader audience.

6. Conclusions

This investigation has examined the state of program synthesis from natural language, uncovering various trends and motivations within the field of automatic code generation. Through meticulous analysis of current literature, this study underscores the expanding role of natural language processing (NLP) tools and their potential to profoundly influence computing disciplines.

The advancements in NLP not only enhance communication capabilities but also facilitate the creation of sophisticated methods for generating syntactic representations of programming languages, as highlighted in the referenced paper [50]. Such methodologies leverage pre-trained, language-based components, promising to refine the process of transforming human language into executable code.

Furthermore, with AI-based systems becoming ever more integral to daily life and the disruptive capabilities of generative AI models, the incorporation of responsible AI practices becomes imperative. This approach will ensure the development and deployment of large language models and other generative systems are both reliable and trustworthy, fostering greater confidence in their applications.

Future research in the field of program synthesis should focus on improving the interpretability of systems, allowing coding decisions to be more understandable and reliable, especially for non-expert users. Furthermore, domain-specific synthesis models should be developed, using specialized datasets to improve the accuracy and relevance of synthesized programs. Optimizing the scalability and computational efficiency of these systems is equally vital, ensuring their large-scale adoption. Finally, it is critical to incorporate ethical considerations and responsible artificial intelligence principles, ensuring fairness, accountability, and transparency in synthesis systems, and minimizing biases. As for practical implications, integrating program synthesis tools into educational platforms can facilitate learning programming, while in software development, automating repetitive tasks and codebase generation will allow developers to focus on more creative aspects. Furthermore, improving the accessibility and usability of applications through natural language interfaces driven by program synthesis could revolutionize human-computer interaction.

Overall, this study illuminates the dynamic field of program synthesis from natural language, advocating for continued research and development. By harnessing advanced NLP and responsible AI, the gap between human language and computer programming can be bridged more effectively, setting a foundation for future innovations in automatic code generation.

References

[1]. J. Fu, F. B. Bastani, and I.-L. Yen, in Innovations for Requirement Analysis. From Stakeholders' Needs to Formal Designs, edited by B. Paech and C. Martell (Springer Berlin Heidelberg, Berlin, Heidelberg, 2007), pp. 43–61, ISBN 978-3-540-89778-1.

[2]. Z. Manna and R. J. Waldinger, Toward automatic program synthesis (1971), URL https://doi.org/10.1145/362566. 362568.

[3]. S. Jiho and N. Jaechang, A survey of automatic code generation from natural language (2021).

[4]. A. Bandi, P. V. S. R. Adapa, and Y. E. V. P. K. Kuchi, The power of generative ai: A review of requirements, models, input–output formats, evaluation metrics, and challenges (2023), URL https://www.mdpi.com/1999-5903/15/8/260.

[5]. S. Gulwani, in Formal Methods in Computer Aided Design (2010), pp. 1–1.

[6]. Z. Manna and R. Waldinger, A deductive approach to program synthesis (1980), URL https://doi.org/10.1145/357084. 357090.

[7]. E. Kitzelmann, in Approaches and Applications of Inductive Programming, edited by U. Schmid, E. Kitzelmann, and R. Plasmeijer (Springer Berlin Heidelberg, Berlin, Heidelberg, 2010), pp. 50–73, ISBN 978-3-642-11931-6.

[8]. A. F. Subahi, Cognification of program synthesis—a systematic feature-oriented analysis and future direction (2020), URL https://www.mdpi.com/2073-431X/9/2/27.
[9]. J. Sauvola, S. Tarkoma, M. Klemettinen, J. Riekki, and D. Doermann, Future of software development with generative ai (2024).
[10]. C. Ebert and P. Louridas, Generative ai for software practitioners (2023).
[11]. K. S. Kaswan, J. S. Dhatterwal, K. Malik, and A. Baliyan, in 2023 International Conference on Communication, Security and Artificial Intelligence (ICCSAI) (2023), pp. 699–704.
[12]. G. Brunner, A. Konrad, Y. Wang, and R. Wattenhofer, Midi-vae: Modeling dynamics and instrumentation of music with applications to style transfer (2018), URL https://api.semanticscholar.org/CorpusID:49317433.
[13]. T. Karras, S. Laine, and T. Aila, A style-based generator architecture for generative adversarial networks (2021), URL https://doi.org/10.1109/TPAMI.2020.2970919.
[14]. D. Jonassen, J. Howland, J. L. Moore, and R. Marra, in Learning to Solve Problems with Technology: A Constructivist Perspective (2002).
[15]. E. D. Liddy, Natural language processing. (2001).
[16]. D. Jatnika, M. A. Bijaksana, and A. A. Suryani, Word2vec model analysis for semantic similarities in english words (2019), the 4th International Conference on Computer Science and Computational Intelligence (ICCSCI 2019) : Enabling Collaboration to Escalate Impact of Research Results for Society, URL https://www.sciencedirect.com/science/article/ pii/S1877050919310713.
[17]. Venkatesh, S. U. Hegde, Z. A. S, and N. Y, in 2021 International Conference on Advances in Electrical, Computing, Communication and Sustainable Technologies (ICAECT) (2021), pp. 1–8.
[18]. L. Wang, Y. Ling, Z. Yuan, M. Shridhar, C. Bao, Y. Qin, B. Wang, H. Xu, and X. Wang, Gensim: Generating robotic simulation tasks via large language models (2024), 2310.01361.
[19]. A. Namboori, S. Mangale, A. Rosenbaum, and S. Soltan, Gemquad : Generating multilingual question answering datasets from large language models using few shot learning (2024), 2404.09163.
[20]. T. Mikolov, M. Karafiat, L. Burget, J. Cernocky´, and S. Khudanpur, in Recurrent neural network based language model (2010), vol. 2, pp. 1045–1048.
[21]. T. Mikolov, A. Deoras, D. Povey, L. Burget, and J. Cernocky, Strategies for training large scale neural network language models (2011).
[22]. S. Islam, H. Elmekki, A. Elsebai, J. Bentahar, N. Drawel, G. Rjoub, and W. Pedrycz, A comprehensive survey on applications of transformers for deep learning tasks (2023), 2306.07303.
[23]. P. Yin and G. Neubig, in A Syntactic Neural Model for General-Purpose Code Generation (2017), pp. 440–450.
[24]. B. Kitchenham, O. Pearl Brereton, D. Budgen, M. Turner, J. Bailey, and S. Linkman, Information and Software Technology 51, 7 (2009), ISSN 0950-5849, special Section – Most Cited Articles in 2002 and Regular Research Papers, URL http: //www.sciencedirect.com/science/article/pii/S0950584908001390.
[25]. K. Petersen, R. Feldt, S. Mujtaba, and M. Mattsson, in Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (BCS Learning and Development Ltd., 2008), EASE'08, p. 68–77.
[26]. N. S. M. Yusop, J. Grundy, and R. Vasa, IEEE Transactions on Software Engineering 43, 848 (2017).
[27]. R. Moguel-S´anchez, C. Mart´ınez-Palacios, J. Ochar´an-Hern´andez, X. Lim´on, and A. S´anchez-Garc´ıa, Programming and Computer Software 49, 712 (2024).
[28]. P. O. Silva-Vasquez, V. Y. Rosales-Morales, and E. Ben´ıtez-Guerrero, Program. Comput. Softw. 48, 685–701 (2022), ISSN 0361-7688, URL https://doi.org/10.1134/S0361768822080187.
[29]. U. Dissemination, The database of abstracts of reviews of effects (dare) (2002).
[30]. P. Archana, P. B. Harish, N. Rajan, S. P, and N. S. Kumar, in 2021 Asian Conference on Innovation in Technology (ASIANCON) (2021), pp. 1–8.
[31]. D. Wang, W. Dong, and Y. Zhang, in 2020 35th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) (2020), pp. 98–104.
[32]. H. Phan, A. Sharma, and A. Jannesari, in 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW) (2021), pp. 219–226.
[33]. Z. Nan, H. Guan, and X. Shen, HISyn: Human Learning-Inspired Natural Language Programming (Association for Computing Machinery, New York, NY, USA, 2020), p. 75–86, ISBN 9781450370431, URL https://doi.org/10.1145/3368089. 3409673.
[34]. Z. Zhang, S. Wu, R. Jiang, M. Pan, and T. Zhang, in Proceedings of the Tenth Asia-Pacific Symposium on Internetware (Association for Computing Machinery, New York, NY, USA, 2018), Internetware '18, ISBN 9781450365901, URL https: //doi.org/10.1145/3275219.3275229.

[35]. Z. Nan, H. Guan, X. Shen, and C. Liao, in Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction (Association for Computing Machinery, New York, NY, USA, 2021), CC 2021, p. 141–152, ISBN 9781450383257, URL https://doi.org/10.1145/3446804.3446852.
[36]. I. Gavran, E. Darulova, and R. Majumdar, Interactive synthesis of temporal specifications from examples and natural language (2020), URL https://doi.org/10.1145/3428269.
[37]. S. Zamanirad, B. Benatallah, M. C. Barukh, F. Casati, and C. Rodriguez, in 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE) (2017), pp. 832–837.
[38]. H. Guan, X. Shen, and H. Krim, in SC17: International Conference for High Performance Computing, Networking, Storage and Analysis (2017), pp. 1–14.
[39]. T. Gvero and V. Kuncak, in 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering (2015), vol. 2, pp. 689–692.
[40]. N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma, in 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE) (2022), pp. 1219–1231.
[41]. N. Tao, A. Ventresque, and T. Saber, in 2023 IEEE Latin American Conference on Computational Intelligence (LA-CCI) (2023), pp. 1–6.
[42]. S. Lee, S. Y. Nam, and J. Kim, Program synthesis through learning the input-output behavior of commands (2022).
[43]. N. Tao, A. Ventresque, and T. Saber, in Optimization and Learning – 5th International Conference, OLA 2022, Syracuse, Sicilia, Italy, July 18-20, 2022, Proceedings, edited by B. Dorronsoro, M. Pavone, A. Nakib, and E.-G. Talbi (Springer, 2022), vol. 1684 of Communications in Computer and Information Science, pp. 240–252, ISBN 978-3-031-22039-5, URL https://doi.org/10.1007/978-3-031-22039-5_19.
[44]. A. Bhardwaj, P. Khanna, S. Kumar, and Pragya, Generative model for nlp applications based on component extraction (2020), international Conference on Computational Intelligence and Data Science, URL https://www.sciencedirect. com/science/article/pii/S1877050920308577.
[45]. K. Rahmani, M. Raza, S. Gulwani, V. Le, D. Morris, A. Radhakrishna, G. Soares, and A. Tiwari, Multi-modal program inference: a marriage of pre-trained language models and component-based synthesis (2021), URL https://doi.org/10. 1145/3485535.
[46]. P. Denny, J. Leinonen, J. Prather, A. Luxton-Reilly, T. Amarouche, B. A. Becker, and B. N. Reeves, in Proceedings of the 55th ACM Technical Symposium on Computer Science Education V. 1 (Association for Computing Machinery, New York, NY, USA, 2024), SIGCSE 2024, p. 296–302, ISBN 9798400704239, URL https://doi.org/10.1145/3626252.3630909.
[47]. M. Kacmajor and J. D. Kelleher, Automatic acquisition of annotated training corpora for test-code generation (2019), URL https://www.mdpi.com/2078-2489/10/2/66.
[48]. M.-F. Wong, S. Guo, C.-N. Hang, S.-W. Ho, and C.-W. Tan, Natural language generation and understanding of big code for ai-assisted programming: A review (2023), URL https://www.mdpi.com/1099-4300/25/6/888.
[49]. C. E. A. Coello, M. N. Alimam, and R. Kouatly, Effectiveness of chatgpt in coding: A comparative analysis of popular large language models (2024), URL https://www.mdpi.com/2673-6470/4/1/5.
[50]. M. Rabinovich, M. Stern, and D. Klein, Abstract syntax networks for code generation and semantic parsing (2017), URL https://arxiv.org/abs/1704.07535.

Информация об авторах / Information about authors

Роландо РАМИРЕС-РУЭДА учится в аспирантуре по программированию в Университете Веракруса (Мексика). Лектор отделения системного программирования в одном из институтов этого университета. Сфера научных интересов: автоматический вывод, искусственный интеллект, мультиагентные системы.

Rolando RAMÍREZ-RUEDA – PhD Student in Computer Science from the University of Veracruz in Mexico. Professor in the systems department of Veracruz University Institute in Mexico. Research interests: Automated Reasoning, Artificial Intelligence, Multiagent Systems.

Эдгард БЕНИТЕС-ГУЭРРЕРО имеет степень PhD по программированию Гренобльского университета (Франция). Профессор факультета статистики и информатики Университета Веракруса (Мексика). Сфера научных интересов: человеко-машинное взаимодействие, искусственный интеллект, совместные вычисления, управление данными и визуализация данных.

Edgard BENÍTEZ-GUERRERO – PhD in Computer Science from the University of Grenoble in France. Professor at the Faculty of Statistics and Informatics of the University of Veracruz in Mexico. Research interests: Human Computer Interaction, Artificial Intelligence, Collaborative Computing, Data Management and Visualization.

Кармен МЕЗУРА-ГОДОЙ получил степень PhD по программированию в Университете Савойи (Франция). Профессор факультета статистики и информатики Университета Веракруса (Мексика). Основные научные интересы: человеко-машинное взаимодействие, пользовательский опыт взаимодействия, совместная работа с поддержкой компьютера, визуализация и мультиагентные системы.

Carmen MEZURA-GODOY – PhD in Computer Science from the University of Savoie in France. Professor at the Faculty of Statistics and Informatics of the University of Veracruz in Mexico. Main research interests: Human-Computer Interaction, User eXperience-UX, Computer-Supported Collaborative Work, Visualization and Multiagent Systems

Эверардо БАРСЕНАС имеет степень PhD по программированию Гренобльского университета, работает в должности доцента на факультете программной инженерии Национального университета Мексики. Его научные интересы включают модальную логику, теорию доказательств, автоматизированный логический вывод, описательную логику, проверки моделей и формальные верификации.

Everardo BARCENAS holds a PhD in Computing Science from the University of Grenoble, and is an Assistant Professor in the Computing Engineering Department of the National University of Mexico. His research interests include: Modal Logics, Proof Theory, Automated Reasoning, Description Logics, Model Checking and Formal Verification.