Program Generation Methods: Types and Instances

Daniil Borodin, Alexander Prutzkow

***Abstract*—The study systematizes existing approaches by chronological principle and categories. The strategy of searching for sources consists of using modern library platforms and keywords on the topic of program generation. We classified program generation methods and identified the following types and their instances. Template methods generate a program using natural language, UML diagrams, and formal specifications, as well as code generation for specific platforms, including emulation of processor architectures. The CASE methods convert high-level descriptions into executable code, including generation in Isabelle/HOL and the use of multi-level rule sets. We analyzed model-based code generation methods, including polyhedral models and the Ptolemy platform. We reviewed tools using of genetic algorithms for creating program code. Compositional programming is represented by the SPIRAL, KLEE projects and other modern developments. A separate type is made up of methods based on artificial intelligence and machine learning, including neural network architectures (AlphaCode, CODEnn) and large language models (CodeBERT, Code Llama). We identified their advantages and areas of application for the types. The types are presented as a scheme that corresponding to directions of development of program generation methods. The study revealed a tendency to move from traditional template methods to technologies based on large language models and machine learning. We will use results of this review in our study on generation of programs that transform arrays.**

***Keywords*—software code generation, development automation, template methods, CASE, UML, evolutionary algorithms, compositional programming.**

I. Introduction

Automatic program generation is the process of creating software to solve a specific problem with minimal human involvement. The program that implements this process is a code generator. The complexity of modern applications requires new approaches to development, which stimulates the development of automation based on various principles: from template programming to methods based on artificial intelligence.

With the right conditions and parameters, and correct development of the generator, most of the code can be created automatically. The programmer only has to manually complete the rest of the work and test it. The generator includes the following key components:

- program code templates – samples according to which the code will be created;
- domain metadata – the structure that needs to be modeled in the program;
- domain rules – parameters that define the structure and behavior of domain metadata, usually implemented in the generator program itself [1].

The use of generation in the development of the project's program code reduces human involvement, which provides the following advantages:

- using a generator ensures that the generated code matches the originally designed application structure and templates;
- when requirements change, it is possible to update templates, generator and/or metadata and output a new version of the code;
- after defining the metadata, the code generation process takes less time than human code creation;
- the generated code has no errors;
- the generator creates code in accordance to coding conventions.

There are various approaches to program generation, which can be classified according to several criteria.

Several reviews on program code generation have been published, but they cannot reflect the latest advances in research in this area. First, several reviews were published more than a decade ago, so they cannot cover the latest algorithms (reviews of 2000 [2], 2012 [3], 2013 [4], 2015 [5]). Second, modern reviews focus on the capabilities of code generation using large language models (LLM) and do not show the chronology of the development of these methods [6, 7, 8]. To fill the research gap, this paper provides an review of the types of program generation methods.

II. Purpose of the Study

The purpose of the study is to systematize and analyze program generation methods for further developing a method for generating array processing programs. The paper examines various approaches to source code generation, their classification and practical application. The paper is intended to form the state-of-art of program generation, identify development trends and determine perspective areas for future research.

The results of the conducted analysis of methods to program generation will be used in our research on the development of a method that generates of programs for the transformation of arrays.

Manuscript received April 18, 2025.
D. Borodin is with Lipetsk State Pedagogical University, 398020, Lenin str., 42, Lipetsk, Russia (e-mail: mail@prutzkow.com).

A. Prutzkow is with Ryazan State Radio Engineering University, 390005, Gagarin str., 59/1, Ryazan, Russia, and with Lipetsk State Pedagogical University, 398020, Lenin str., 42, Lipetsk, Russia (e-mail: mail@prutzkow.com).

III. Research Methodology

We investigate program generation methods by a systematic literature review. The review included a search strategy and publication selection by date.

The sources of literature were well-known online libraries of scientific publications on computer science and artificial intelligence, namely IEEE Xplore, ACM Digital Library, SpringerLink, INTUIT, ArXiv, CyberLeninka. Google Scholar was used to search for publications included in other libraries as well.

Keywords that we use in search queries are: code generation, program code generation, code generation, program code generation. The target interval is from 2000 to 2024. But there is a earlier paper [9] that is necessary for understanding the beginning of the development of the type of program generation specified in the paper.

IV. Types of program generation methods

We divide methods to program generation by chronology and by type. The emergence of new methods is associated with the development of computer science and artificial intelligence (AI) in recent years.

To systematize methods to program generation, the following types can be distinguished:

1. Template methods

One of the earliest methods to program generation since the 1970s is template-based code generation. Software source code is generated based on predefined templates, which contain both static elements and dynamic placeholders for inserting data or logic. These templates depend on specific task parameters and are created using specialized template description languages, which allows developers to quickly generate complex code fragments while maintaining architectural consistency. This type reduces the likelihood of errors that occur when manually writing repetitive or similar blocks, and also simplifies project scaling by changing input data without having to rewrite the templates themselves. The following subtypes of generation methods can be attributed to this method.

1.1. Natural Language Generation (NLP)

It consists of converting descriptions in natural language into program code.

1.1.1. One of the first studies in this area [9] proposed a natural language system that generates the corresponding source code using a program description. The system checks the completeness of the description. If the first specified description is sufficient, the program will create a corresponding problem-solving scenario with an answer. If the program description is insufficient, the system will ask questions to resolve the ambiguity until it is eliminated.

1.1.2. The use of the newly developed controlled processing language (CPL) language for code generation is proposed in [10]. CPL receives descriptions in natural language and generates an intermediate representation that limits the obtained characteristics to a subset of natural language and allows for better perception and creation of code. In addition, this language uses heuristics to resolve ambiguity in descriptions in natural language.

1.1.3. An end-user programming paradigm in Python is proposed in [11]. The Vajra system generates Python code fragments from a natural language description. The user enters a natural language command at a specific place in the source code. The system then generates a list of possible operators and associated parameters that are most similar in semantics. There are procedures that the user can select to resolve ambiguities in the process by choosing from several candidate fragments.

1.1.4. DeepPseudo [12] is a method for generating pseudocode in natural language using code feature extraction and transformers. It uses the Transformer encoder for analysis, extracts local semantic features through a special module, and then uses a pseudocode generator to represent the algorithm textually.

1.2. Generation based on UML diagrams

It is a process of automatic creation of program code from unified models presented in the form of diagrams of various kinds.

1.2.1. The application of the type was proposed in [13]. The GenERTiCA framework generates code for distributed real-time embedded systems based on modeling and aspect-oriented programming. During the design process of the system, UML diagrams are developed. The diagrams describe architecture and non-functional requirements, such as time constraints, fault tolerance, and security. GenERTiCA then generates the source code, including parts that are separated from the main logic, and integrates them into the code base at the compilation stage.

1.2.2. A similar study [14] specifies that it is necessary to use class diagrams for UML modeling and code generation based on them. An important point is that associations should be implemented as classes when generating code based on diagrams. This approach helps to solve problems with multiple relationships, aggregation, and associative classes.

1.2.3. The code generation process includes three main phases: input, transformation, and output [15]. In the input phase, the system accepts UML diagrams in XMI format. These diagrams are processed by XMIParser, which creates metamodel instances for each diagram. Then, in the transformation phase, CodeGenerator uses these metamodels to generate isolated code for each diagram: the structure is formed based on the class diagram, the control flow of methods is based on sequence diagrams, and object manipulations and user interactions are added through the actions of activity diagrams.

1.2.4. Another method for generation of Java code from UML diagrams using XMI representation is proposed in [16]. The diagrams are modeled in the BOUML tool, exported to XMI, from which metadata is extracted, and Java code is generated based on them.

1.2.5. In [17], a framework for generation of programs based on a visual model created by the user and on a data flow diagram, is described. The framework is a client-server application and uses the developed data storage format. The conducted studies have shown that the framework is suitable for automating complexly structured tasks when developing the original model and can be used to speed up data processing.

1.3. Generating code for specific platforms

It is a process of creating software code optimized for specific

hardware or software environments. This approach includes emulation of various processor architectures, automatic translation of serial code into parallel code.

1.3.1. In [18], the architecture and operating principles of QEMU, a machine emulator using portable dynamic translation to emulate multiple processors (x86, PowerPC, ARM, Sparc) on various platforms, are proposed. The main component of the system is the dynamic code generator dyngen, which translates the instructions of the target processor into micro-operations represented by fragments of C code compiled with GCC. These micro-operations are combined into the executable code of the host machine, which cached for reuse.

1.3.2. The framework [19] allows engineers to create sensor network components at both the application and protocol levels. The framework is based on Simulink, Stateflow, and embedded components of Coder. These components are building blocks for modeling, simulation, and subsequent code generation for various target platforms and operating systems.

1.3.3. The system automatically converts the sequential C code into parallel CUDA code [20]. The program is analyzed through an abstract syntax tree (AST), iteration space polytopes, and data dependencies are extracted. Based on them, affine transformations are created, which are used to generate operator domains. The final code is formed using a polyhedral generator, such as CLooG.

1.3.4. Generating EMF-compatible code in the Fujaba program development system [21] involves two approaches: (1) directly generating Java code from Fujaba models or (2) creating an Ecore file and then processing it with the EMF generator. Structural elements are converted to Ecore elements, and behavioral models are generated as separate Java code and integrated with the main structure. An example is a task management project, where the generator creates an Ecore file and Java code, which are combined by the EMF generator into a finished program.

1.4. Generating programs based on formal specifications

These are methods of generating software code based on rigorous mathematical descriptions of the requirements and behavior of a system.

1.4.1. The KVEST methodology [22] is focused on automated test generation and software verification based on formal rules. The key aspect is the use of explicit specifications that describes not only specific values, but also classes of acceptable values. This approach, combined with a model-oriented method and abstract data structures, makes it possible to create implementation-independent specifications that serve as the basis for subsequent generation of test sets as well as test coverage assessment through a modified criterion of a perfect disjunctive normal function.

1.4.2. There is a code generation mechanism: the control policy is specified in a high-level programming language and then automatically converted into the source code for a specific problem [23]. This approach eliminates the need for significant time costs and deep knowledge of code optimization. The study shows that the use of code generation solves a wide range of control problems and achieving increased performance compared to traditional general-purpose solvers.

1.5. Code generation for distributed systems

It is a process of generating program designed to run in a distributed computing environment.

1.5.1. In [24], a method for generating code for linear program sections is based on the exact joint solution of the problems of selecting and scheduling instructions, taking into account the restrictions on the number of registers. The advantages of the approach are: consideration of parallelism, code optimization when there is a shortage of registers, and automatic use of instructions with multiple results. The proposed algorithm improves the compilation process, providing a more optimal distribution of resources and increasing processor performance.

1.5.2. The GCD distributed computing system [25] integrates created modules and simplifies the process of prototyping complex engineering systems. The software toolkit includes a specialized template format, a library of functions for their interpretation, and a system of initialization files for source data.

1.6. Using the CASE tools that generate program code based on models or specifications

It is a form of implementation of a template method, which automates the process of converting high-level descriptions into executable code. Modern CASE tools are based on methodologies of structural or object-oriented analysis and design, which formalizes the stages of program development and minimizing the influence of third-party factors. In such methodologies, specifications are used to describe external requirements for the system, including text descriptions, use case diagrams, class, sequence and activity diagrams [26]. There are the studies related to this type of program generation:

1.6.1. Code generation in Isabelle/HOL [27] is based on a multi-stage process, where the key element is the introduction of an intermediate language, Mini-Haskell. The language is a bridge between the source higher-order logic with type classes and the target functional programming languages. The semantics of both the source language and Mini-Haskell are specified in terms of higher-order rewriting systems, which provides the basis for proving the correctness of the translation. This allows functions and data to be replaced by more efficient analogues within a single process.

1.6.2. Modeling systems by an analyst together with a subject area expert is proposed in [28]. During the ongoing dialogue “author-reader”, the model diagrams are constructed, verified, and corrected. The developed structure is supplemented with a quantitative assessment as well. For this purpose, the AllFusion Business Process Modeler package uses cost indicators of work, the so-called ABC analysis and user properties of user defined properties processes. The presented approaches allow for the program generation.

1.6.3. In the method [29], user enters the data of the entity structure and its attributes. The data are written to a unified storage (relational database) containing information about the template structure. After saving the data, to ensure reliability and scalability, another process reads the data, analyzes it, and then generates code.

In parallel with the template methods, another type of

program generation was developed that required consideration.

2. Code generation based on models

This type is a methodology for converting high-level software specifications into executable code by using formal models and transformation algorithms. This approach covers various methods, including code generation based on state diagrams, where automata transition graphs are the basis for creating programs with explicitly allocated states, and generation based on class diagrams, which forms the structure of object-oriented code. Particular attention is paid to creating code for real-time systems, where synchronous data flows and finite automata models are translated into code analyzed relative to the worst case of execution. Modern research is aimed at improving transformation algorithms and optimizing both the size of the generated code and the efficiency of the generation process itself. Let us consider the instances of the type.

2.1. A polyhedral model for code generation formalizes data and computation dependencies that simplifies program optimization, especially in the context of parallel computing [30]. Code generation includes automatic partitioning of computations, task redistribution, and optimization of execution order, which improves performance with multitasking and multi-core systems.

2.2. A method for generating code with a polyhedral model is a process of converting multidimensional representations of program cycles back into executable code, including the stages of design and separation of domains [31]. In this case, to improve scalability, methods for removing scalar dimensions and the use of domain iterators are used, which processes complex transformations and large programs with many operators.

2.3. The Ptolemy platform [32] transformes of models into executable code for embedded Java systems. These models are specified in advance and describe the behavior of the system and its interaction with external resources.

2.4. The method of generating a modeling program uses the translation of a control machine, specified in the form of a table or graph, into a production algorithm [33]. The production algorithm is presented in a text file, which is fed to the input of the generating program. The output is a program that models the specified machine.

2.5. Software can be developed based on a multi-level set of rules for generating software source code [34]. Metagraphs are chosen as the presentation structure. A model for generating software source code is presented, as well as a design methodology using a generation system based on a multi-level set of rules. The problem of developing automated tests within the framework of the proposed approach is considered.

2.6. The process of developing tests designed to check the coherence of memory accesses can be performed in the Elbrus assembly language [35]. The approach generates impacts for a test system that combines the RTL model of the processor with the software model of the memory subsystem. The test generator automatically creates sequences of operations that model various situations, including possible errors and coherence violations, which identifies problems at early stages of development.

2.7. The code2seq model [36] generates sequences of program code tokens based on its structural representation in the form of AST paths. The approach is based on the use of an architecture that uses neural networks with an attention mechanism, allowing the model to focus on the most informative AST fragments when predicting the output sequence. This approach outperforms previous models specifically designed for programming languages, as well as neural machine translation models.

2.8. ReCode [37] is a method based on extracting a subtree with links to existing code examples in a neural code generation model. Sentences that are similar to the input ones are extracted using a sentence similarity scoring method based on dynamic programming. Then, n-grams of action sequences that produce a related AST are extracted. As a result, the probability of actions that will cause the resulting n-gram action subtree to be in the predicted code is increased.

2.9. The problem of generation of commit messages is important for understanding code changes in frequent software updates. The authors of the study [38] propose a new model called ATOM, that uses AST to represent the structure of changed code, which helps to better account for its semantics. The model integrates both extracted and generated messages through a hybrid ranking module that selects the most appropriate message for a particular code change.

2.10. A tool for creating directed graphs based on a deterministic finite automaton simplifies the writing of program code [39]. In it, the program structure is presented as an automaton, where the states are different stages of the algorithm execution, and the transitions are determined by control structures. The transformation of such a representation into code is carried out taking into account predefined rules and templates.

With the use of the two code generation technologies discussed, it became clear that it was necessary to improve the process using previously developed algorithms.

3. Generating programs based on search and evolutionary algorithms

This type methods treat the creation of program code as an optimization problem or as a search for a solution in a space of possible options. These approaches are based on the principles of simulating natural evolution, where programs are formed by successively applying mutation and selection operations to achieve the desired behavior. One of the early examples is the use of genetic algorithms to create ROP chains, where the process begins with analyzing the executable file, identifying potentially useful code fragments, parameterizing them, and further optimizing them through mutations and fitness assessment. Another direction is the use of evolutionary algorithms to solve logical problems, where state generators form sets of solutions that are subject to iterative optimization. Particular attention is paid to the use of semantic information and modern machine learning (ML) methods to improve the quality of generation. Let us consider current studies of this type:

3.1. The Spi2Java tool [40] generates Java code implementing the cryptographic protocols described in the formal specification of the spi calculus language. Spi2Java is part of the Spi calculus toolkit, which also includes a preprocessor, a parser, and a security analyzer. The latter analyzes protocols and identifies their weaknesses. Once a

protocol has been analyzed and sufficient confidence in its correctness has been achieved, Spi2Java creates an implementation of it in Java, thereby reducing the risk of introducing vulnerabilities at the programming stage.

3.2. Code generation for solving convex optimization problems is possible using the CVXGEN tool [41]. Based on a high-level description of such problems, CVXGEN creates specialized C code that is compiled into a solver. The focus is on problems that can be transformed using convex programming techniques into small-size convex quadratic programs.

3.3. The ROPER tool [42] uses genetic algorithms to create ROP chains for the ARM architecture. First, parts of the program are found in the executable file, their parameters are calculated, and then the file is loaded into a virtual machine to test the chains. Genetic mutations change the addresses of program parts and data on the stack, and fitness is estimated by the difference between the current and target register values.

3.4. Programs in the Prolog language are generated through a declarative description of state generators that form the solution search space [43]. The approach uses of bit chains to represent the states of objects and bitwise operations to generate new states, while the process of constructing a solution itself is implemented through recursive computational procedures without the need to store the full state graph in memory. Generators can either save the history of previous states for step-by-step construction of a solution, or work without it, relying only on the current state.

3.5. In [44], hash functions are used for obfuscating program code are considered. Taking into account the features, a method of program generation is proposed, based on the genetic programming approach using the fitness function and an algorithm that is repeated a certain number of times, depending on the input data. Generation is carried out by combining bit and arithmetic operations, as well as pseudo-random permutations, which ensures high entropy and resistance to reverse analysis.

3.6. A genetic algorithm for code generation can be used to solve the problem of finding the minimum of a function of two variables without using derivatives [45]. The results obtained showed that the approach was able to write a program that solves a computational problem, and also, if necessary, improve a human-written algorithm.

With the development of program code and the creation of specialized online platforms for its storage and exchange, a method of generating programs based on the principles of compositional programming emerged.

4. Compositional programming

This type is a software development methodology based on the principle of forming predefined components or modules to create complex software systems. The approach is characterized by a high degree of modularity and code reuse, which is especially important in the context of modern development methods such as microservice architecture and distributed systems.

4.1. The SPIRAL project [46] generates code tailored to a specific architecture by formulating the tuning problem as an optimization problem using the mathematical structure of DSP algorithms and a feedback mechanism. The system can generate code for transforms such as the discrete Fourier transform and wavelet transform, as confirmed by experiments.

4.2. KLEE [47] is a symbolic execution tool that can generate tests with high coverage. KLEE operates on the basis of symbolic code execution, where input data is represented as symbolic values rather than specific numbers, which explores many possible program execution paths. The system uses a constraint solver to generate specific test cases that reproduce detected errors in the source code.

4.3. To improve the quality of C language compilers, the Csmith tool was created to generate random test scenarios to detect errors in compilers using differential testing [48]. The language generates programs with no undefined behavior due to control over the language constructs used and their combinations. The resulting programs comply with the C99 standard at both static and dynamic levels.

4.4 The Julia programming language [49] is designed for technical computing with a focus on high performance and dynamic typing. It provides tools for compile-time code generation using functions. This allows users to write optimized code and extend the output type system.

4.5. In [50] it is shown that currently there are no universal integrated development environments with semantic editing of program code. The creation of such environments can significantly increase the productivity of the programmer, due to semantic editing and the function of the version control system, tracking not textual changes, but changes in the abstract model of the program code.

4.6. The C code could be generated from a parametric description of a quadratic program (QP) as input [51]. The resulting code is compiled into an optimization solver for QP that can run on embedded platforms. In addition, this code is based on operator splitting quadratic program (OSQP). It is a new open source solver for quadratic programming. The generated C code is library-free and minimal in size.

4.7. In [52], a system for generation of parallel code from fragments of the C programs using the OpenMP polyhedral optimization model is presented. It improves parallelism and code accuracy through integer linear programming to find suitable affine transformations.

4.8. Generation of monitoring programs for technical objects is presented in [53]. The proposed structure consists of several subsystems: data integration, information acquisition and processing, generation of models of observed objects and data collection processes. The structure uses the inductive-deductive approach to constructing models of objects based on the data received from them. Interaction is carried out with n-dimensional vectors of numerical values characterizing the states of the elements of objects at certain points in time. In this case, the drivers perform preliminary aggregation and normalization of data before transmitting them to the system.

4.9. In [54], the following set of operations are used for code generation in loops: optimization of sequential transitions, calculation of the number of loop repetitions before the loop body, use of a delay slot, induced variables, and removal of unnecessary inductive variables.

4.10. REDCODER is an extraction framework that obtains code or summary data from a database and provides it as an

adjunct to code generation or summarization models [55]. REDCODER has a couple of unique features: (1) it extends the capabilities of modern data mining to find relevant code and (2) it can work with databases that include unimodal (code or natural language description only) or bimodal instances (code-description pairs).

The creation of ML and AI technologies has led to the development of new methods for generating programs based on these technologies. Many articles are currently devoted to this method.

5. Methods based on artificial intelligence and machine learning

They are a modern approach to program generation based on the use of natural language processing (NLP) algorithms, neural networks, and LLM for the automatic creation of program code. This approach has been actively developed since 2012 and demonstrates significant potential in the tasks of converting descriptions in natural language into executable code, as well as in the tasks of optimizing and improving existing programs. Let's consider the categories of research devoted to this topic.

5.1. Neural network architectures

Code generation uses neural networks and NLP technologies to transform text descriptions into executable code.

5.1.1. AlphaCode is a code generation system that solved 54.3% of the problems in the latest programming competition on the Codeforces platform [56]. AlphaCode solves problems by generating millions of diverse programs using specially trained transformer networks, and then filtering and clustering these programs into a maximum of 10 representations. This is the first time that an AI system has shown competitive results in a programming competition.

5.1.2. The method of generating input data for fuzzy testing of JavaScript interpreters improves the quality and speed of fuzzy testing [57]. The data is generated using neural networks and compilation with subsequent fragmentation of AST and aggregation of fragments. Using the method, it was possible to form a new set of input data. The method of generating semantically correct code for fuzzy testing provides coverage of 44.4% by lines of code and 51.2% by functions.

5.1.3. The CODEnn neural network is trained to find semantic similarities between a natural language description and a code fragment [58]. The study is based on the representation of the program in the form of vector spaces, which matches queries with code fragments, taking into account their semantic features. When the code fragment and the description are semantically similar, the embedded vectors will be close to each other.

5.1.4. The methodology for automating the creation of control programs for the CNC machines is considered in [59], where the key aspect is the use of intelligent systems, in particular neural networks, to solve the problem of generating and verifying control programs. At the same time, the need to use ML to analyze various processing parameters, such as tool types, cutting modes and motion paths, is emphasized, which increases the efficiency and safety of the mechanical processing action.

5.1.5. The Prophet system generates code fixes (patches) by training on a set of successful patches written by programmers and obtained from open source repositories [60]. Prophet uses a parameterized probabilistic model to assign a correctness probability to each candidate patch in the search space, based on universal characteristics of correct code that are identified and learned through statistical analysis.

5.1.6. A method analyzes natural language descriptions to generate Python code fragments [61]. The simulated neural architecture uses a probabilistic grammar model to explicitly capture the syntax of a programming language as a priori knowledge. This approach was also found to be effective in generating complex, multi-layered programs.

5.2. Large language models

They generate code using scalable transformer-based architectures trained on huge amounts of text data and program code.

5.2.1. Methods for generating malicious software code using several LLM are proposed in [62]. Compromise identifiers can be recognized from the generated code parts. It is shown that the code created using LLM has certain characteristics that can be detected by antiviruses.

5.2.2. CodeBERT is a bimodal pre-trained model for the PL programming language and the NL language [63]. The model uses a neural architecture based on Transformer and is updated using a hybrid objective function that includes a pre-training task to detect replaced tokens, which consists of identifying plausible alternatives sampled from generators. CodeBERT searches natural language code and generates code documentation.

5.2.3. Evaluation of LLM for Python program synthesis in MBPP and MathQA-Python benchmarks is considered in [64]. The results show that the performance grows logarithmically with increasing model size. The largest model without additional training solves 59.6% of MBPP problems, and additional training increases the accuracy by 10%. On MathQA-Python, the accuracy reaches 83.8%. Human interaction, including feedback, reduces errors by half. The analysis revealed difficulties with generating complex programs and limited ability of models to predict execution results.

5.2.4. Code Llama is a family of LLM based on Llama 2, designed for programming tasks [65]. All models are trained on sequences up to 16,000 tokens long and show improved results when processing inputs up to 100,000 tokens. The 7B, 13B, and 70B variants of the models support the feature of filling in the surrounding content. In the HumanEval and MBPP tests, Code Llama achieves accuracy of up to 67% and 65% respectively, outperforming other available models.

5.2.5. Using ChatGPT for code generation [66] has a high degree of non-determinism under the default setting: the proportion of coding tasks with zero equal test yield across different queries is 75.8%, 51.0%, and 47.6% for three different code generation datasets: CodeContests, APPS, and HumanEval, respectively.

5.2.6. In [67], the problem of software synthesis is considered as a task of creating a program based on specifications, through input-output examples or natural language, using LLM for code generation. A family of models CODEGEN trained on natural and programming language data is presented. A multi-stage approach to

software synthesis is also investigated through the creation of an open benchmark MTPB, which showed an improvement in synthesis when using multi-turn hints compared to a single-stage method.

V. Further discussions

The study of current methods of generating program code demonstrates several main vectors of its development. The issue of improving the quality of created programs is most actively considered. At the same time, special attention is paid to the last stages of the development of generation systems, in particular, to the tasks of calculating the compliance of the generated code with the original requirements. However, the initial stage of forming a candidate database remains insufficiently studied, despite its importance for the entire process.

There is a tendency to consider individual stages of the generation algorithm as independent components. This requires a revision of approaches to the design of code generation algorithms.

Modern research is moving away from traditional template-based methods to technologies using AI and ML. LLM show potential in tasks of converting natural language descriptions into executable code and optimizing existing programs. However, traditional methods such as compositional programming and the use of CASE tools continue to evolve and find application in various subject areas.

The integration of various code generation methods could significantly improve software development results. A promising direction is also the creation of reference data sets for the unification of evaluation methods and the development of research in the field of generation of program code.

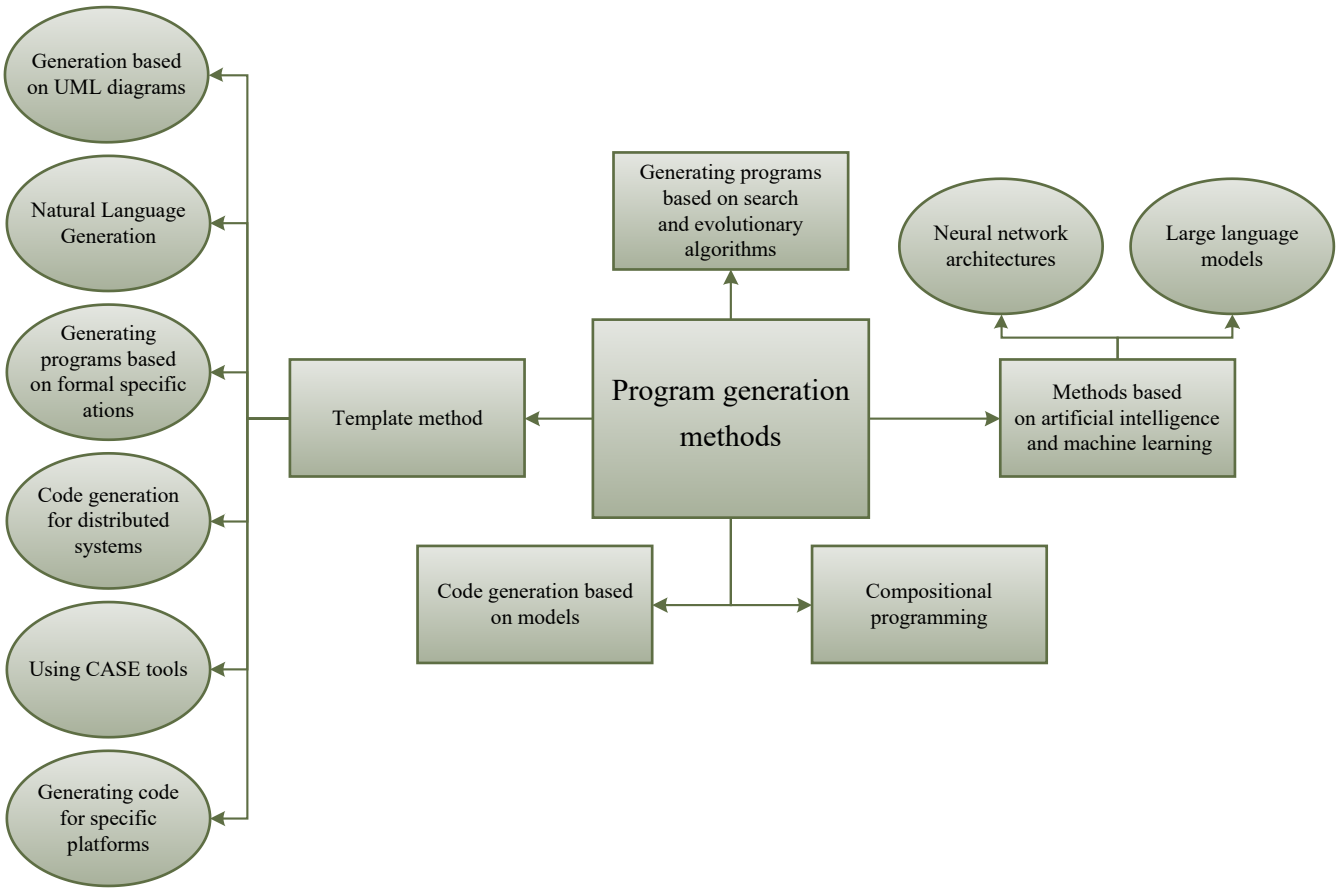
VI. Type summary

We present the summary of program generation methods as a scheme (see fig.).

The scheme reflects the considered directions of development of program generation methods. Each block of the scheme corresponds to the type of program generation.

VII. Application in our study

Key aspects useful for the developed method of generating programs for array transformation based on the depth-first search (DFS) algorithm include: template approaches for parameterization of array transformation operations, evolutionary algorithms as a source of ideas for search strategies, and work with state graphs, where DFS provides enumeration of transformation paths.



Types and subtypes of program generation methods

The review revealed gaps in the systematization of the initial stages of generation, which the method fills by modeling the state space of the array as a graph, where nodes are intermediate states, and edges are admissible operations. Thus, the review confirms the relevance of the combination of DFS with templates and the identification of areas for improving the method for generating array processing programs.

VIII. Conclusions

Program generation is a dynamically developing field that actively uses template approaches, artificial intelligence, evolutionary algorithms, and compositional methods. Further research focuses on the integration of these methods, which can significantly improve the efficiency of software development.

An analysis of existing approaches shows that each program generation method has its own specifics and areas of appropriate application. Traditional template methods and CASE tools remain in demand for solving well-formalized problems, while modern technologies based on LLM open up new possibilities in software development automation.

Further development of the field of program code generation will be largely determined by progress in the field of AI and NLP. At the same time, the need for a systematic approach to the design of generation algorithms, taking into account the interrelationship of all stages of the process of creating program code, remains important.

This paper presents a review of modern methods of software code generation, demonstrating the important role of AI and ML technologies in the development of this field. Compared with existing reviews, this paper classifies code generation methods more clearly, covering both traditional template approaches and the latest advances in neural network technologies over the past decade. Particular attention is paid to the relationship between the various stages of the code generation process and their impact on the final result. The review should become a valuable reference not only for researchers seeking to develop code generation technologies, but also for practicing programmers wishing to choose the most appropriate tools and approaches for their projects.

References

[1] Aldan A. Vvdenie v generaciu programmnogo koda [Introduction to Program Code Generation] [Electronic Resource]: Study Course / Author: Askar Aldan; INTUIT.ru. 2025. URL: https://intuit.ru/studies/courses/4432/975/lecture/14619?ysclid=m5land94x3873782162 (accessed: 02/09/2025). [in Rus].

[2] Bhartacharyya S.S. et al. Software Synthesis and Code Generation for Signal Processing Systems // IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing, 2002, 47(9):849-875.

[3] Domı E. et al. A Systematic Review of Code Generation Proposals from State Machine Specifications // Information and Software Technology, 2012, 54(10):1045-1066.

[4] Bajovs A. et al. Code Generation from UML Model: State of the Art and Practical Implications // Applied Computer Systems, 2013.

[5] Gurunule D., Nashipudimath M. A Review: Analysis of Aspect Orientation and Model Driven Engineering for Code Generation // Procedia Computer Science, 2015, 45:852-861.

[6] Shin J., Nam J. A Survey of Automatic Code Generation from Natural Language // Journal of Information Processing Systems, 2021, 17(3):537-555.

[7] Dehaerne E. et al. Code Generation Using Machine Learning: A Systematic Review // IEEE Access, 2022, 10:82434-82455.

[8] Cambaz D., Zhang X. Use of AI-Driven Code Generation Models in Teaching and Learning Programming: A Systematic Literature Review // 55th ACM Technical Symposium on Computer Science Education, 2024, 1:172-178.

[9] Heidorn G.E. An Interactive Simulation Programming System Which Converses in English // 6th Conference on Winter Simulation, 1973:781-794.

[10] Clark P. et al. Naturalness vs. Predictability: A Key Debate in Controlled Languages // Controlled Natural Language. Springer, 2009:65-81.

[11] Schlegel V. et al. Vajra: Step-by-Step Programming with Natural Language // 24th International Conference on Intelligent User Interfaces, 2019:30-39.

[12] Yang G. et al. Fine-Grained Pseudo-Code Generation Method via Code Feature Extraction and Transformer // 28th Asia-Pacific Software Engineering Conference (APSEC), 2021:213-222.

[13] Wehrmeister M.A. et al. GenERTiCA: A Tool for Code Generation and Aspects Weaving // 11th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC), 2008:234-238.

[14] Gessenharter D. Mapping the UML2 Semantics of Associations to a Java Code Generation Model // Model Driven Engineering Languages and Systems: 11th International Conference (MoDELS), 2008:813-827.

[15] Usman M., Nadeem A. Automatic Generation of Java Code from UML Diagrams Using UJECTOR // International Journal of Software Engineering and Its Applications, 2009, 3(2):21-37.

[16] Vadakkumcheril T. et al. A Simple Implementation of UML Sequence Diagram to Java Code Generation through XMI Representation // International Journal of Emerging Technology and Advanced Engineering, 2013, 3(12):1-5.

[17] Minakova O.V. et al. Postroenie generatora programmnogo koda dlya reshenia ingenernih zadach [Building a Program Code Generator for Solving Engineering Problems] // Bulletin of the Voronezh State Technical University, 2020, 6(3):14-19. [in Rus].

[18] Bellard F. QEMU, a Fast and Portable Dynamic Translator // USENIX Annual Technical Conference, FREENIX Track, 2005, 41(46):10-55.

[19] Mozumdar M.M.R. et al. A Framework for Modeling, Simulation and Automatic Code Generation of Sensor Network Application // 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Networks, 2008:515-522.

[20] Baskaran M.M. et al. Automatic C-to-CUDA Code Generation for Affine Programs // Compiler Construction: 19th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, 2010:244-263.

[21] Geiger L. et al. EMF Code Generation with Fujaba // Fujaba Days, 2007:25-29.

[22] Burdonov I.B. et al. Formalnie specifikacii v tehnolgiah obratnoy ingenerii [Formal Specifications in Reverse Engineering and Software Verification Technologies] // Proceedings of the Institute for System Programming of the Russian Academy of Sciences, 2000, 1:39-54. [in Rus].

[23] Mattingley J. et al. Code Generation for Receding Horizon Control // IEEE International Symposium on Computer-Aided Control System Design, 2010:985-992.

[24] V'yukova N.I. et al. Code Generation by the Method of Exact Joint Solution of Command Selection and Planning Problems // Software Engineering, 2014, 6:8-15. [in Rus].

[25] Sokolov A.P. et al. Development of Code Generation Software Based on Templates for Creating Engineering Analysis Systems // Software Engineering, 2019, 10(9–10):400–416. DOI: 10.17587/prin.10.400-416.

[26] Petrenko A.K., Marchuk A.G. Modern Approaches to Software Development Automation // Software Engineering, 2017, 4:22–30. [in Rus].

[27] Haftmann F., Nipkow T. Code Generation via Higher-Order Rewrite Systems // International Symposium on Functional and Logic Programming, 2010:103-117.

[28] Myshenkov K.S. Metodika obosnovania vibora CASE-sredstv dlya analiza i proektirovania system upravlenya predpriyatiyami [Methodology for Justifying the Selection of CASE-Tools for Analysis and Design of Enterprise Management Systems] // Innovations, 2013, 10(180):112-122. [in Rus].

[29] Tarasiev A.A. et al. Razrabotka prototipa CASE-sredstva dlya sozdania avtomatizirorannih system na osnove web-prilogeni s ispolzovaniem generacii koda [Development of a Prototype CASE-Tool for Creating Automated Systems Based on Web Applications Using Code Generation] // 28th International Crimean Conference «Microwave Engineering and Telecommunication Technologies» (CriMiCo), 2018:452-458. [in Rus].

[30] Bastoul C. Code Generation in the Polyhedral Model is Easier than You Think // 13th International Conference on Parallel Architecture and Compilation Techniques, 2004:7-16.

[31] Vasilache N. et al. Polyhedral Code Generation in the Real World // Compiler Construction: 15th International Conference, Held as Part of the Joint European Conferences on Theory and Practice of Software, 2006:185-201.

[32] Schoeberl M. et al. Code Generation for Embedded Java with Ptolemy // Software Technologies for Embedded and Ubiquitous

Systems: 8th IFIP WG 10.2 International Workshop (SEUS), 2010:155-166.
[33] Dovgal V.M. et al. On the Issue of Solving the Problem of Automatic Code Generation Based on a Given Control Production Algorithm // In the World of Scientific Discoveries, 2012, 1(25):220-235. [in Rus].
[34] Samokhvalov E.N. et al. Generaciya ishodnogo koda programmnogo obespechenia na osnove mnogourovnego nabora pravil [Source Code Generation of Software Based on a Multi-Level Set of Rules] // Herald of the Bauman Moscow State Technical University. Series «Instrument Engineering», 2014, 5(98):77-87. [in Rus].
[35] Burenkov V.S. Generator testov dlya verifikacii protokola kogerentnosti kashpamyati [Test Generator for Verifying the Cache Coherence Protocol] // Issues of Radio Electronics, Series ECT, 2014, 3:56-63. [in Rus].
[36] Alon U. et al. code2seq: Generating Sequences from Structured Representations of Code // arXiv preprint arXiv, 2018:1808.01400.
[37] Hayati S.A. et al. Retrieval-Based Neural Code Generation // arXiv preprint arXiv, 2018:1808.10025.
[38] Liu S. et al. Atom: Commit Message Generation Based on Abstract Syntax Tree and Hybrid Ranking // IEEE Transactions on Software Engineering, 2020, 48(5):1800-1817.
[39] Gitel'man V.S., Tutov I.A. Generacia koda na osnove determinirovannogo konechnogo avtomata [Code Generation Based on a Deterministic Finite Automaton] // Youth and Modern Information Technologies, 2022:299-301. [in Rus].
[40] Pozza D. et al. Spi2java: Automatic Cryptographic Protocol Java Code Generation From Spi Calculus // 18th International Conference on Advanced Information Networking and Applications (AINA), 2004, 1:400-405.
[41] Mattingley J., Boyd S. CVXGEN: A Code Generator for Embedded Convex Optimization // Optimization and Engineering, 2012, 13:1-27.
[42] Fraser O.L. et al. Return-Oriented Programme Evolution with ROPER: A Proof of Concept // Genetic and Evolutionary Computation Conference Companion, 2017:1447-1454.
[43] Polovikova O.N., Zenkov A.V. Solving a Certain Class of Logical Problems in Prolog by Declaring State Generators // Computer Tools in Education, 2019, 1:54-67. [in Rus].
[44] Lebedev R.K. Automatic Generation of Hash Functions for Program Code Obfuscation // Applied Discrete Mathematics, 2020, 50:102-117. [in Rus].
[45] Shintyakov D.V. Opit ispolzovaniya geneticheskih algoritmov dlya generacii koda algoritmov optimizcii [Experience of Using Genetic Algorithms for Generating Optimization Algorithm Code] // Processing, Transmission, and Protection of Information in Computer Systems, 2020:169-174. [in Rus].
[46] Puschel M. et al. SPIRAL: Code Generation for DSP Transforms // Proceedings of the IEEE, 2005, 93(2):232-275.
[47] Cadar C. et al. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs // OSDI, 2008, 8:209-224.
[48] Yang X. et al. Finding and Understanding Bugs in C Compilers // 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, 2011:283-294.
[49] Bezanson J. et al. Julia: A Fast Dynamic Language for Technical Computing // arXiv preprint arXiv, 2012:1209.5145.
[50] Vanyasin N.V. Semanticheskoe redaktirovanie programmnogo koda v intelektualnih integrirovannih sredah razrabotki prilozheniy [Semantic Editing of Program Code in Intelligent Integrated Application Development Environments] // Cybernetics and Programming, 2017, 1:61-68. DOI: 10.7256/2306-4196.2017.1.18881. [in Rus].
[51] Banjac G. et al. Embedded Code Generation Using the OSQP Solver // 56th IEEE Annual Conference on Decision and Control (CDC), 2017:1906-1911.
[52] Allamanis M. et al. A Survey of Machine Learning for Big Code and Naturalness // ACM Computing Surveys (CSUR), 2018, 51(4):1-37.
[53] Vodyakho A.I. et al. Systemi avtomaticheskoy generacii programm monitortinga [Systems of Automatic Program Generation for Monitoring] // Engineering Bulletin of Don, 2019, 8(59):19. [in Rus].
[54] Arhipov I.S. Generacia optimalnogo obyektnogo koda [Generation of Optimal Object Code] // Proceedings of the Institute for System Programming of the RAS (Proceedings of ISP RAS), 2020, 32(3):49-56. [in Rus].
[55] Parvez M.R. et al. Retrieval Augmented Code Generation and Summarization // arXiv preprint arXiv, 2021:2108.11601.
[56] Li Y. et al. Competition-Level Code Generation with Alphacode // Science, 2022, 378(6624):1092-1097.
[57] Kozachok A.V. et al. Method generacii semanticheski korrektnogo koda dlya fazzingtestirovaniya interprotatorov javascript [Method for Generating Semantically Correct Code for Fuzzing Testing of JavaScript Interpreters] // Cybersecurity Issues, 2023, 5:57. [in Rus].
[58] Gu X. et al. Deep Code Search // IEEE/ACM 40th International Conference on Software Engineering, 2018:933-944.
[59] Solkin A.Yu. Sposobi avtomatizacii sozdaniya upravlyaushih programm dlya malorezhushego oborudovaniya s CHPU [Methods for Automating the Creation of Control Programs for CNC Metal-Cutting Equipment] // Bulletin of the Tatishchev Volga University, 2012, 2(19):165-168. [in Rus].
[60] Long F., Rinard M. Automatic Patch Generation by Learning Correct Code // 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, 2016:298-312.
[61] Yin P., Neubig G. A Syntactic Neural Model for General-Purpose Code Generation // arXiv preprint arXiv, 2017:1704.01696.
[62] Filyukov D.A. Primenenie neironnih setey dlya formirovaniya koda vredonosnogo programmnogo obespecheniya [Application of Neural Networks for Generating Malicious Software Code] // Innovations and Investments, 2023, 7:199-204. [in Rus].
[63] Feng Z. et al. Codebert: A Pre-Trained Model for Programming and Natural Languages // arXiv preprint arXiv, 2020:2002.08155.
[64] Austin J. et al. Program Synthesis with Large Language Models // arXiv preprint arXiv, 2021:2108.07732.
[65] Roziere B. et al. Code Llama: Open Foundation Models for Code // arXiv preprint arXiv, 2023:2308.12950.
[66] Ouyang S. et al. An Empirical Study of the Non-Determinism of ChatGPT in Code Generation // ACM Transactions on Software Engineering and Methodology, 2025, 34(2):1-28.
[67] Nijkamp E. et al. Codegen: An Open Large Language Model for Code with Multi-Turn Program Synthesis // arXiv preprint arXiv, 2022:2203.13474.