LLM-based and Retrieval-Augmented Control Code Generation

Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, Nafise Eskandani
firstname.lastname@de.abb.com
ABB Research
Germany

ABSTRACT

Control code is designed and implemented for industrial automation applications that manage power plants, petrochemical processes, or steel production. Popular large language models (LLM) can synthesize low-level control code in the Structured Text programming notation according to the standard IEC 61131-3, but are not aware of proprietary control code function block libraries, which are often used in practice. To automate control logic implementation tasks, we proposed a retrieval-augmented control code generation method that can integrate such function blocks into the generated code. With this method control engineers can benefit from the code generation capabilities of LLMs, re-use proprietary and well-tested function blocks, and speed up typical programming tasks significantly. We have evaluated the method using a prototypical implementation based on GPT-4, LangChain, OpenPLC, and the open-source OSCAT function block library. In several spot sample tests, we successfully generated IEC 61131-3 ST code that integrated the desired function blocks, could be compiled, and validated through simulations.

CCS CONCEPTS

• **Software and its engineering** → **Automatic programming**; *Command and control languages*; • **Applied computing** → *Computer-aided design*; • **Computing methodologies** → *Natural language processing*.

KEYWORDS

Large language models, code generation, IEC 61131-3, industrial automation, PLC, DCS, ChatGPT, GPT-4

ACM Reference Format:
Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, Nafise Eskandani. 2024. LLM-based and Retrieval-Augmented Control Code Generation. In *Proceedings of 1st International Workshop on Large Language Models for Code (LLM4Code'24).* ACM, New York, NY, USA, 8 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

In industrial automation, engineers design and implement control logic code that processes a vast amount of sensor data and sends control signals to actuators, such as motors, pumps, robots, or heat exchangers [10]. This control logic code thus manages many complex production processes, e.g., chemical refineries, power plants, and mines. Programming control logic is still a largely manual process [12, 26]. However, recently LLMs have been found to be useful for generating programs according to the programming languages used in industrial automation [14].

While LLMs can synthesize basic control logic from natural language prompts [14], control logic in practice is often assembled by using pre-specified function blocks that encapsulate often used algorithms [26]. Such function blocks are bundled into programming libraries. Most function block libraries are proprietary to specific vendors and not available as public data. Therefore, LLMs trained on public Internet data are not aware of such function blocks and cannot instantiate them during code generation. This may make generated control logic code unnecessarily complex, inefficient, and hard to understand.

Researchers have proposed retrieval-augmented generation of Python and Java source code, e.g., to integrate formerly written code or summaries to inform code generation [19, 20, 22]. Furthermore, there are many benchmarks to evaluate the code generation quality of LLMs [6, 11, 18, 27], which again target general-purposed programming languages. Before the advent of LLMs, researchers have also attempted to generate control logic code e.g., using model-driven development [21, 24, 28] or rule-based processing of specification drawings [7, 9, 13]. However, there is no approach yet to utilize retrieval-augmented generation to improve the quality of LLM-based code generation for industrial control logic.

We propose an LLM-based and retrieval-augmented control code generation method that can automate many control programming tasks. It integrates pre-built and well-tested proprietary function blocks into the synthesized code. The method suggests loading text embeddings of function block specifications into a vector store and using this information to augment control logic generation prompts extracted from requirements documents. The LLM can answer the prompts with IEC 61131-3 Structured Text (ST) programs, which can directly be imported into a control programming environment. The method combines the automation domain knowledge and program generation capability of LLMs with proprietary domain knowledge encoded in pre-built function blocks.

To validate the method, we created a prototypical implementation using open-source tools based on GPT-4, LangChain, and OpenPLC. We report on an experimental evaluation of the method, where we generated control logic with function blocks from the OSCAT function block library, which we then compiled and tested. Our prototype implementation was able to generate functionally correct control logic in these tests, which demonstrates the basic feasibility of the method. Future work shall analyze the method's robustness with more and deeper tests.

LLM4Code'24, Apr 16th, 2024, Lisbon, Portugal

ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

2 BACKGROUND

Control engineers write control code for various industrial processes, including chemical refineries, power plants, and paper machines. In these installations, thousands of sensors measure temperature, flow, pressure, and level [10]. The signals are communicated to automation controllers, which cyclicly execute the control code (e.g., every 250 ms) on the latest sensor values to compute control outputs. The control output signals are communicated to actuators, such as pumps, motors, and valves. Control logic may involve simple boolean logic, PID control, or sophisticated optimization algorithms.

Control engineers often express control logic in IEC 61131-3 Structured Text, one of the popular PLC programming notations [26]. Its syntax is based on Pascal and C. The language supports typical control flow constructs, conditional statements, and object-oriented concepts, such as classes and inheritance. Program organization units (POU) in IEC 61131-3 structure the source code and can be specified as reusable function blocks in libraries.

Function block libraries often contain hundreds of pre-built function blocks for mathematical, logical, signal, and control functions [1]. There are libraries from different vendors and libraries for specific application domains (e.g., for building automation or mining). They can be loaded into control programming environments and be instantiated by control engineers so that they do not have to program all functionality from scratch. The proprietary source code of the function blocks is usually not available for the control engineers, who refer to reference specification documents to use the blocks. Fig. 1 shows a simple function block specification for computing a mathematical function.

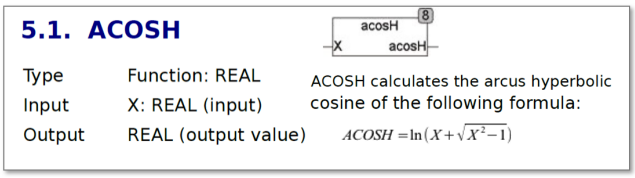

Figure 1: Example: function block specification from OSCAT open-source library

The requirements for control code design and implementation often come from process engineers with deep expertise in chemical production processes. Writing so-called *control narratives* is a lightweight method to specify the requirements. These prose texts (often hundreds of pages) describe the desired control strategies in natural language and abstract from the vendor-specifics of an automation system.

They also contain references to tag names (i.e., identifiers of sensors and actuators), which the control engineers often use for variable names in control logic. For example, a control narrative could state to fill a particular tank T101 to 70 percent, keep its temperature at 52 degrees Celsius using the temperature sensor TT107 and the control valve TIC104, and activate the mixer CV401 in tank T101 for 5 minutes. The control engineer then selects appropriate pre-built function blocks for the logic and implements the control logic in IEC 61131-3 ST accordingly. The goal of our method is to partially automate this process.

3 RETRIEVAL-AUGMENTED CONTROL CODE GENERATION METHOD

Fig. 2 illustrates our control logic code generation method, which applies the well-known retrieval augmented generation technique [17]. As inputs, the method uses function block specifications and control narratives, and as output, the method creates control logic code in a desired programming notation. The control logic code can be imported into typical integrated development environments for PLCs (e.g., CODESYS, TwinCAT, OpenPLC, etc.).

To prepare the control logic code generation steps a) to c) process the function block specifications and save the results into a vector store. These steps only need to be performed once per function block library, later all kinds of control logic generation prompts can be augmented using such a prepared vector store.

Step a) The **Document Loader** retrieves the function block specifications from the source artifacts. Popular frameworks, such as LangChain, provide more than 100 document loaders for different types of files (e.g. HTML, PDF, code) and locations (e.g., local storage, cloud storage, etc.). Function block specifications are often available as PDF, HTML, or XML documents. Most function block libraries are subdivided into multiple modules with a separate document per module. In this case, the Document Loader iterates over a set of files.

Step b) The **Document Chunker** splits each document into smaller chunks so that they can later fit into the LLM's context window. It is best practice to create chunks with semantically related information, which, in the context of our method, naturally fits the specification of a single function block with its inputs, outputs, and description of functionality. Single control logic generation prompts should refer only to a few function block types at once to remain within the context window limits when augmenting the prompts with the function block specifications.

Step c) After the document chunker has split a function block specification, the **Text Embedder** encodes each chunk into a vector of floating point numbers. This encoding enables measuring the distance between two vectors to assess their relatedness. The smaller the distance, the higher related are the chunks. Such text embeddings can be created locally or via a call to a public API. Many text embedding models are available (e.g., text-embedding-ada-002 from OpenAI), which vary in input token length, performance, and price for API calls.

The Embedder stores the result into a **Vector Store**, which is a special type of database to efficiently handle vector data. Vector stores are optimized for similarity searches, which identify related vectors given particular query vectors. This can use similarity measures such as cosine similarity or Euclidean distance. There are both managed vector stores (e.g., Pinecone, Weaviate, Amazon Elasticsearch) and local vector stores (e.g., Milvus, FAISS, Qdrant), which have different scalability and usability characteristics.

After processing the function block specifications and populating the Vector Store, the actual control logic generation process can start:

Step 1: The **Control Narrative Extractor** processes the control narratives available for an automation project and extracts concise control logic generation prompts out of them. The actual process

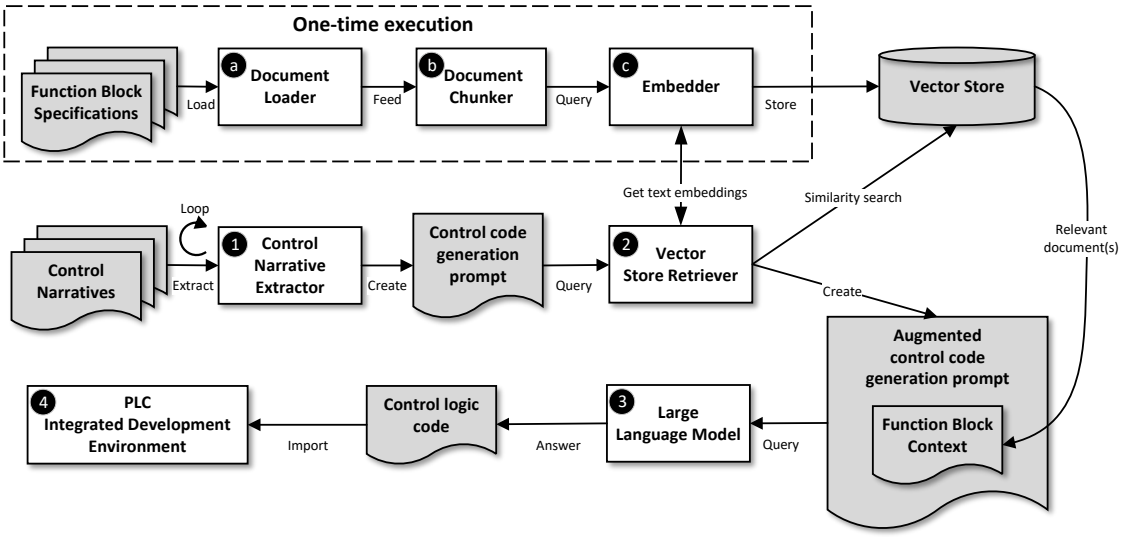

Figure 2: Retrieval-augmented Control Code Generation Method

may vary depending on the specific automation project and formulation of the control strategies. A heuristic is to generate individual prompts for single control loops. The extractor itself can use vector encoding and LLM for text summarization but this is not mandatory. Typical actions of the extraction process include searching for set point values, alarm limits, and control sequence steps in the control narratives.

To illustrate the concept, a simplistic control generation prompt can be formulated as follows:

> "Write an IEC 61131-3 Structured text program to use a PID controller with a set point value of 5.0. Use one or more of the following pre-specified function blocks."

Step 2: The **Vector Store Retriever** creates an encoding of the prompt using the Text Embedder, which for example looks as follows:

> 0.025929883, 0.0029817247, -0.016806405, -0.0059034266, -0.010274504, 0.017004129, -0.009963797, -0.008699787, -0.014871551, 0.0028104829, 0.019701626, 0.033358596, ...

Afterwards, using the floating point vector, it performs a similarity search using the Vector Store with a configurable number of search results. Different search techniques can be used, e.g., based on similarity score or distance. For example, the Maximum Marginal Relevance (MMR) search technique aims at improving the diversity of the search results, thus avoiding too many redundant results.

Step 3: The similarity search results may now contain several function block specifications (i.e., input/output parameters names and types, descriptions of functionality) retrieved from the Vector Store, which is augmented to the original control logic generation prompt. The method then queries an LLM with this augmented prompt. As an answer, the user retrieves control logic code that

```
1  VAR
2      PID_Controller: FT_PID;
3      Control_Set: CONTROL_SET2;
4      Control_PWM: CTRL_PWM;
5  END_VAR
6
7  Control_Set.KT := 5.0;
8  Control_Set.PID := TRUE;
9
10 PID_Controller.IN := Control_Set.KP;
11 PID_Controller.KP := Control_Set.PID_K;
12 PID_Controller.TN := Control_Set.PID_TN;
13 PID_Controller.TV := Control_Set.PID_TV;
14 PID_Controller.ILIM_L := -100.0;
15 PID_Controller.ILIM_H := 100.0;
16 PID_Controller.IEN := TRUE;
17 PID_Controller.RST := FALSE;
18
19 Control_PWM.CI := PID_Controller.Y;
20 Control_PWM.MAN_IN := 0.0;
21 Control_PWM.MANUAL := FALSE;
22 Control_PWM.F := 50.0;
```

Figure 3: Control logic code generated with LLM, which contains instantiation of function blocks based on the specification documents.

includes instantiations of the function block types augmented to the prompt. For example, Fig. 3 depicts the IEC 61131-3 ST code generated for the prompt stated above. The similarity search returned four results (not shown here for brevity), which included the function block specifications for FT_PID, CONTROL_SET2, and CTRL_PWM. These blocks are declared in lines 2-4 of the code and the remainder of the code uses their correct input and output parameters. In line 19, the output of the FT_PID block is mapped to the input of the CTRL_PWM block, demonstrating the correct connection of these blocks.

Step 4: Finally, the method suggests importing the generated control logic code into a typical **PLC Integrated Development Environment** (IDE). The import may be conducted for example by file transfer or API calls if supported by the IDE. PLC IDEs include editors to further refine and debug the code. Usually, such generated code is integrated into larger programs and projects. A control engineer can use the PLC IDE to interpret or compile the code, perform simulation test runs, and eventually download it to an industrial controller in the production environment.

4 PROTOTYPICAL IMPLEMENTATION

Creating a prototypical implementation using the **LangChain** framework [3] is a pragmatic way to evaluate our method. LangChain is a collection of tools and API wrappers to manage LLMs. These include tools for incorporating external knowledge into an LLM, interfacing with different LLM-APIs, and creating retrieval chains supported by conversational user interfaces. However, our method is not dependent on LangChain but can be realized with other means.

For representative **Function Block Specifications**, we assessed different commercial and open-source libraries. We reviewed a dozen ABB-internal function block libraries from different control systems and application domains. Ultimately, we decided to use the open-source function block library OSCAT BASIC to increase the transparency of our study and reproducibility of our results. It contains more than 400 function blocks for basic mathematical operations, signal generation, and automation control, specified in a 496-page PDF document (4.3 MB). Its complexity is comparable with several commercial libraries.

As **Document Loader** LangChain provides several options for PDF files, e.g., PyPDF, MathPix, PDFMiner, or PDF Plumber. For the OSCAT library PDF, we selected the PDFPlumberLoader, which loads the content of each PDF page as a string into a Python Document object. It also extracts metadata about the PDF and its pages. For other function block library specifications, using the PDFMiner to generate HTML text may be a viable option, since it retains structural information, such as font sizes, of the text, which can then be parsed with other tools to recover semantical coherences.

For a **Document Chunker** that creates chunks for individual function blocks, there are again different options. With the PDF loaded into an HTML file, it is possible to find headings based on HTML tags or font sizes. In our case, for the OSCAT library PDF document, we built a custom document chunker based on a regular expression. It matched the common numbering in the OSCAT document for the subsection heading of each function block to split the document into strings that matched the description of an individual function block. This procedure sacrifices generality to ensure that each document later found through similarity search is short and thus does not inflate the context augmented to the prompt. We recovered page numbers for each chunk by text comparison with the initial page-based splitting of the PDFPlumberLoader. This allows to preview PDF pages that were considered relevant to the prompt by the LLM.

For the **Text Embedder** we used the default embedding model (OpenAI's text-embedding-ada-002), which is available through LangChain and recommended by OpenAI for most use cases. According to OpenAI, this model has replaced five previous models and showed strong performance regarding text search, code search, and sentence similarity tasks. It has a context window of 8192 tokens and produces embeddings with 1536 dimensions, which is lower than previous models and thus makes the embeddings more cost-effective.

LangChain provides interfaces to dozens of **Vector Stores**. To store the embeddings, we preferred a simple, local vector database and selected FAISS-CPU 1.7.4. The database file containing the text embeddings had a size of 4.5 MBytes. FAISS is implemented in C++ and contains several methods for efficient similarity search. We used a score-based similarity search, which showed good results.

We collected over 50 **Control Narratives** from customers and manually formulated several control code generation prompts based on them. The wide range of documents used as input assures that our formulated prompts reflect realistic situations. Consequently, we did not implement the proposed **Control Narrative Extractor** for the prototype implementation. This component could be refined in future work.

Through an iterative process, we created a common template for our **Control Code Generation Prompts** (Fig. 4). It first instructs the LLM to create a *function block* instead of an IEC 61131-3 *program*, so that the resulting code can be easily embedded into other programs. Then the actual description of the desired logic follows. Afterward, we provide further instructions to utilize the augmented function block specifications, omit comments and explanations, and follow a common output structure. The prompt template is generic but may need modifications for PLC IDEs, which for example do not support all IEC 61131-3 constructs.

```
Write a self-contained IEC 61131-3 ST function block:

<Query>

Use the following pre-specified function blocks if useful.
Do not write code for the inner body of the function block.
Do not invoke the function blocks directly with the '()' operator,
only read out their outputs.
Do not include comments in the code.
Provide no explanations, only the source code.

The generated ST Code must follow this format:

FUNCTION_BLOCK <name of the function block>
VAR_INPUT (** here define input variables **) END_VAR
VAR_OUTPUT (** here define output variables **) END_VAR
VAR (** here define internal temp variables **) END_VAR
(** here write ST code of this function block**)
END_FUNCTION_BLOCK
```

Figure 4: Prompt template for control logic generation

As **Vector Store Retriever** we used a simple RetrievalQA chain, which we configured to return the documents found through a similarity search in the Vector store in the answer. We also added a conversational user interface using Streamlit, which we enhanced to display thumbnails of found source documents after each code generation answer to allow users to check the validity of the code generation by reading the original function block specification in the OSCAT library PDF file.

As **Large Language Model** we selected gpt-4-32k, version 0613, since it has shown good IEC 61131-3 ST code generation quality in similar experiments [14]. GPT-4's temperature parameter was set to 0 to produce as deterministic output as possible, other parameters were left on default values. Document retrieval including the call of embeddings API was performed in a sub-second time range while the delivery of the complete LLM answer took up to 20 seconds.

We selected the OpenPLC-Editor [2] as **PLC Integrated Development Environment**, which is available as open-source and can be used by other researchers to verify or replicate our results. It provides an ST-code editor, compiler, and debugger, which allows to run the code. A Python script (OpenPLC-Importer) can import custom function blocks into an OpenPLC project.

5 EXPERIMENTAL EVALUATION

In this section, we provide several tests of the RAG-based control code generation method using the prototype implementation (code available on Github [15]). The goal is to assess the principle feasibility of correct code generation. Comparing the generation quality of different LLMs or comparing the code generation duration with manual implementation are considered future work.

5.1 Test 1: Sample & Average

For the first test, we embedded the following query into our prompt template (cf. Fig. 4):

> "Sample an incoming signal at one-second intervals and calculate the mean value of the last eight sampled data points."

OSCAT provides at least two function blocks that can be used in ST-code to implement the query. The SH_1 function module is specified on page 300 of the document and has two inputs and three outputs (Fig. 5). Notice that the standard output parameter "OUT" is not listed in the output variables, but only occurs in the function block drawing and the text. The module samples an input value with an adjustable sampling time. The FT_AVG function module is specified on pages 98 and 99 of the document and has four inputs and one output. It calculates a moving average over each of the last input values "N".

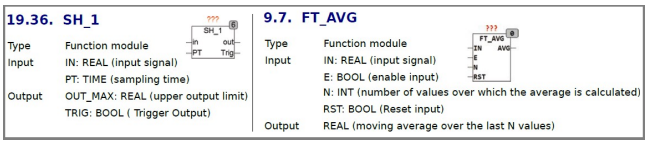

Figure 5: OSCAT function modules

Fig. 6 shows the ST-code generated by GPT-4 for our query augmented with results from a similarity search on the OSCAT embeddings in the vector store. The code follows the template structure prescribed in our prompt template. In line 12, GPT-4 correctly calls the SH_1 function block with a timing value of 1 second, to address the "one-second interval" in our query. In line 13, the FT_AVG function block is called. However, GPT-4 used the SH_1 incorrect variable OUT_MAX as input instead of the correct variable OUT, because the variable OUT was not explicitly listed in the specification document. Also, the output variable of FT_AVG is wrongly named in line 14, since again the OSCAT specification document did not provide the correct name. We thus manually corrected both errors and changed the variable OUT_MAX to OUT and the variable OUT to AVG.

We then imported the code and the required dependent function blocks into an OpenPLC project. The SH_1 block additionally required the OSCAT function T_PLC_MS and the FT_AVG block required the OSCAT function block DELAY and the function INC1. Then we instantiated our generated function block in a simple IEC 61131-3 ST program and set the input signal to 10.0. The code compiled successfully to C-code. We then simulated the code using the OpenPLC Debugger, which allows us to force input values and monitor output values. Fig. 7 shows the respective line charts of

```
1  FUNCTION_BLOCK SAMPLE_AND_AVERAGE
2  VAR_INPUT
3      signal_in: REAL;
4  END_VAR
5  VAR_OUTPUT
6      avg_out: REAL;
7  END_VAR
8  VAR
9      sh1: SH_1;
10     ft_avg: FT_AVG;
11 END_VAR
12 sh1(IN := signal_in, PT := T#1S);
13 ft_avg(IN := sh1.OUT_MAX, E := TRUE, N := 8, RST := FALSE);
14 avg_out := ft_avg.OUT;
15 END_FUNCTION_BLOCK
```

Figure 6: Generated ST-code with two OSCAT function blocks instantiated

the execution for more than 100 seconds (x-axis: runtime in seconds, y-axis: variable value). At 25 seconds, we set the input value (SIGNAL_IN) from 10.0 to 20.0 and the output value (AVG_OUT) correctly calculated new average values for eight intervals before also reaching the value 20.0. This test has thus demonstrated that the generated code with the manual fixes provides the desired functionality.

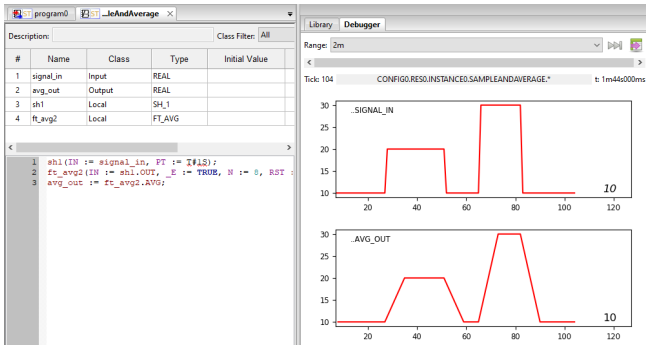

Figure 7: Running the generated ST-code in OpenPLC shows the expected functionality

5.2 Test 2: Sine Wave & Stair

In a second test, we formulated this query, aiming at the integration of more complex function blocks:

> "Generate a sine wave with a period time of 10 sec, a signal amplitude of 10, a signal offset of 5, and a signal delay of zero. Use the output of this as the input to a stair function with a step size of 2.3."

In the OSCAT basic library, there are two sine wave generators and two stair functions, which can be used to implement code according to the prompt. Fig. 8 shows the code resulting from the retrieval-augmented generation.

In line 8, the GEN_SIN function block from the OSCAT PDF page 261 is correctly selected and instantiated. This function block provides a sine wave generator with a programmable period. In line 9, the STAIR function from the OSCAT PDF page 304 is correctly

```
1  FUNCTION_BLOCK SINE_TO_STAIR
2  VAR_INPUT
3  END_VAR
4  VAR_OUTPUT
5      outputSignal: REAL;
6  END_VAR
7  VAR
8      genSin: GEN_SIN;
9      stair: STAIR;
10 END_VAR
11 genSin(PT := T#10S, AM := 10, OS := 5, DL := 0);
12 stair(X := genSin.OUT, D := 2.3);
13 outputSignal := stair.OUTPUT;
14 END_FUNCTION_BLOCK
```

Figure 8: Generated ST-code for prompt requesting sine wave and stair function

selected. However, in OSCAT this is a callable *function*, not a function block that can be instantiated and assigned to a local variable. In this case, we corrected the error manually, removed line 9, and changed line 13 to call the STAIR function instead of assigning the output of a STAIR function block.

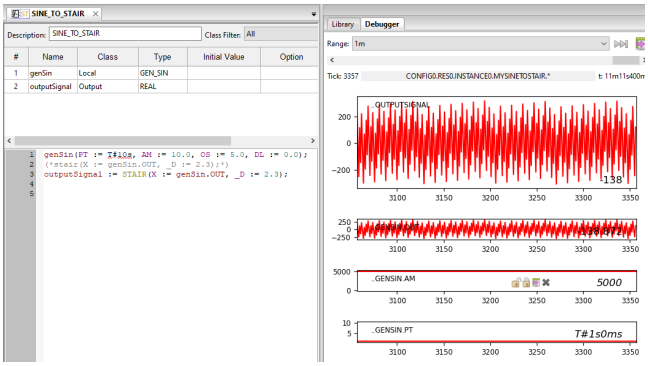

Figure 9: Output of sine wave generator function block in OpenPLC

We then imported the function block code into OpenPLC and created a simple program instantiating it. We also imported the GEN_SIN function block and the STAIR function from an OSCAT library implementation for OpenPLC into the same OpenPLC project file. The GEN_SIN function block required the additional OSCAT functions MODR, FLOOR2, and SIGN_R to be imported. Afterward, the code compiled successfully. Fig. 9 shows the output signals of the generated function block from a run with the OpenPLC Debugger. We forced different input values to generate different kinds of sine waves with stair functions.

5.3 Test 3: Control Module

The third test created a custom PID controller and required a high number of parameters to be set. We formulated the following query:

> "Create a PID controller with dynamic anti-wind up and manual control input for temperature control in an ammonium nitrates reactor. Set point = 180.0, Kp = 50.0, Ki = 1.2, Kd = 10.0, limits between -100 and +100. Add a timer to only set the PID controller to automatic mode after 10 seconds."

The generated code in Fig. 10 shows that GPT-4 correctly selected the OSCAT function block CTRL_PID (page 382-384) and the IEC 61131-3 standard block TON to answer the query. The CTRL_PID block is composed of several other function blocks internally and has 12 inputs and 3 outputs. Lines 11-16 show that our desired parameter values were integrated into the code. GPT-4 also automatically correctly converted the kd, kp, and ki parameters from the prompt as needed for the CTRL_PID block in lines 29-30. It also correctly connected the timer block TON with the PID controller.

```
1  FUNCTION_BLOCK AMMONIUM_NITRATE
2  VAR_INPUT
3      temp_actual: REAL;
4      manual_mode: BOOL;
5      manual_input: REAL;
6  END_VAR
7  VAR_OUTPUT
8      control_output: REAL;
9  END_VAR
10 VAR
11     set_point: REAL := 180.0;
12     kp: REAL := 50.0;
13     ki: REAL := 1.2;
14     kd: REAL := 10.0;
15     ll: REAL := -100.0;
16     lh: REAL := 100.0;
17     pid_controller: CTRL_PID;
18     timer: TON;
19     timer_done: BOOL;
20 END_VAR
21
22 timer(IN:=NOT manual_mode, PT:=T#10S);
23 timer_done := timer.Q;
24
25 IF timer_done THEN
26     pidcontroller(ACT:=temp_actual,
27                   SET:=set_point,
28                   KP:=kp,
29                   TN:=kp/ki,
30                   TV:=kd/kp,
31                   LL:=lower_limit,
32                   LH:=upper_limit,
33                   MAN:=manual_mode,
34                   M_I:=manual_input,
35                   RST:=FALSE);
36     control_output := pidcontroller.Y;
37 ELSE
38     control_output := 0.0;
39 END_IF;
40
41 END_FUNCTION_BLOCK
42
```

Figure 10: Generated ST-code for PID controller

Fig.11 shows a screenshot from running the generated code in OpenPLC. We integrated the function block into a program and manually added simple simulation logic that mimicked a temperature curve in an ammonium nitrates reactor. The code ran with a cycle time of 100 ms. Fig.11 shows that the PID controller correctly switched to auto-mode and wrote control outputs only after 100 ticks (10 secs) when the defined timer ran out. The actual temperature value (MYACT) increases towards the desired set-point of 180 degrees Celsius and then remains on that level. At tick 400, we changed the set-point to 100 and the PID controller correctly lowered the temperature to 100 after a few seconds.

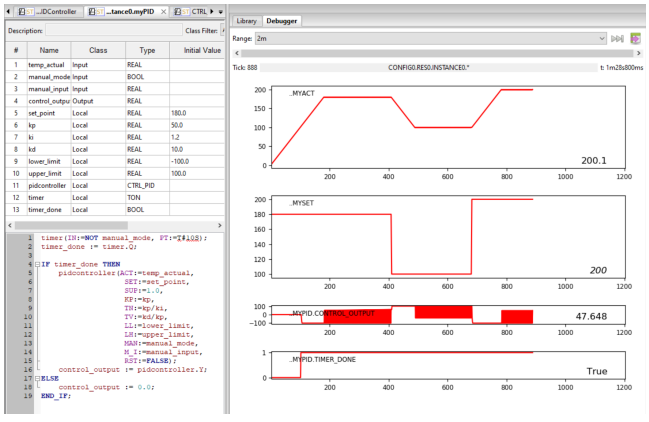

Figure 11: Output of the custom PID controller function block in OpenPLC

The three tests have shown that GPT-4 can correctly identify the required complex function block for control code generation queries supported by retrieval-augmented generation. While providing

initial encouraging evidence, these tests are merely spot samples and not exhaustive. More systematic testing is required in future work.

6 THREATS TO VALIDITY

In this section, we review our study's internal, construct, and internal validity.

The *internal validity* refers to establishing the causal dependency between the manipulation of independent variables, i.e., control narratives, function block specifications, and internal system parameters, to the change of dependent variables, i.e., generated control code quality. A threat to internal validity is the inherent non-deterministic behavior of LLMs resulting in non-repeatable probabilistic generation of control logic. To address this issue we set the LLM model temperature to 0 to reduce output randomness as much as possible. To avoid syntactical and functional errors in the generated code, we compiled and executed the code. This provided an initial quality assessment of the generation output. We established internal validity only through three initial tests, but have not yet exhaustively tested the method. There may be hidden interfering variables that have not surfaced in our tests so far. We also publish our prompts, raw data, and source code to enable independent replication [15].

The *construct validity* describes the extent to which the tests accurately assess what they are supposed to measure. The control generation prompts are still simple though not artificial, but there is more complex control logic in practice. As a construct for a control logic notation, we used IEC 61131-3 ST, which is used in many automation systems [26]. As LLM, we chose GPT-4 and we based the tooling on LangChain, both of which are currently popular choices. The used OpenPLC IDE is less complex than commercial PLC IDEs, but includes all relevant elements (e.g., code editor, compiler, debugger) needed to assess the code generation quality. The OSCAT library is also available as open-source and rarely used in commercial systems. Nevertheless, we argue that its complexity is comparable to several commercial function block libraries.

The *external validity* refers to the potential for transferring the obtained results to other contexts and settings. The retrieval-augmented generation used in our method is independent of the OSCAT function block library and should apply to many other function block libraries in academia and practice. Future work could tackle providing evidence for even more complex libraries and function blocks. Our method is also not dependent on a particular industrial automation domain but can generate code for a vast range of applications, spanning from chemical reactors, and power plants, to industrial paper machines. IEC 61131-3 features four additional programming notations besides ST (including graphical notations), which have the same expressiveness. With appropriate conversion tools, our approach could be also generalized for these notations.

7 RELATED WORK

Our method relates to code generation approaches for general-purpose programming languages and specifically to control code generation approaches. Dehaerne et al. [4] recently reviewed 37 publications, where machine learning was used for code generation. Among these, there are several approaches for Java and Python where code was generated from natural language descriptions. Xu et al. [29] conducted experiments with Python programmers on IDE-integrated code generation from natural language queries and found mixed results regarding the impact on the developer workflow, time efficiency, code correctness, and code quality. Peng at al. [23] showed in an experiment with 95 developers that 55% of development time savings are possible with GitHub Copilot. Vaithilinga et al. [27] analyzed the suitability of LLM-based code generation tools. There are also several benchmarks to evaluate LLM-generated code [6, 11]. Liu et al. [18] proposed the EvalPlus framework for evaluating the correctness of LLM-generated code.

Several frameworks for *retrieval-augmented generation of Python and Java code* have been proposed. Liu et al. [19] designed a retrieval-augmented code generation method using hybrid graph neural networks and tested it with programs written in C. REDCODER [22] integrated formerly written code or summaries into a retrieval-augmented code generation process. ReACC [20] used an external context for code completion by retrieving semantically and lexically similar code snippets from existing codebases. However, none of these approaches specifically handled control logic for industrial applications or IEC 61131-3 code.

Koziolek et al. [12] surveyed different *approaches for control code generation* in industry and academia. In practice, function blocks from control libraries are mostly manually instantiated by importing them into a programming environment and instantiating them in control programs [8]. In some cases where the production processes are highly standardized, sophisticated function blocks only require "glue logic" to connect the blocks. In a few domains, control logic is directly specified in schematic system control diagrams that describe the production processes formally, and can then be easily converted into control logic [5]. While all these practical approaches support control logic implementation, they have different constraints and do not allow for generating all kinds of control logic.

Researchers have proposed several methods for generating control logic code using *model-driven development* [12]. Witsch et al. [28] designed PLC-statecharts to transform UML models into IEC 61131-3 code. Lukman et al. [21] used a domain-specific modeling language called ProcGraph to specify control logic and provide an IEC 61131-3 code generator. Schumacher et al.[24] transformed GRAFCET models into IEC 61131-3 control logic. One commercial PLC IDE (Codesys) provides UML modeling capabilities and allows transformation into control logic, however many control engineers are not familiar with UML.

Other researchers generated control logic from *piping and instrumentation diagrams* (P&IDs), which are CAD drawings created by process engineers [8]. The CAEX transformer applied pre-specified rules on XML-based P&IDs to generate IEC 61131-3 control logic. Steinegger and Zoitl [25] imported P&IDs into an ontology and then generated IEC 61131-3 code. Grüner et al. [7] proposed the ACPLT Rule Engine to convert CAEX-based P&IDs into a graph database (Neo4J) and then used graph queries (Cypher) for the code generation. The AUKOTON method [9] mapped P&IDs to a domain-specific model in Eclipse Ecore and then generated IEC 61131-3 programs. Koziolek et al. [13] applied a rule-based control code generation approach in four large-scale case studies. These

approaches have not yet gained widespread adoption due to missing standard notations for P&IDs, which complicates developing robust tooling.

Finally, researchers have started to apply *LLMs in the context of generating control logic code*. Koziolek et. [14] created a collection of 100 representative prompts for generating IEC 61131-3 ST control logic, to test LLM code generation capabilities. In tests, they found good generation quality by GPT-4. Another approach [16] has tested generating control logic directly from P&ID drawings using LLM-based image recognition. None of the surveyed approaches has attempted retrieval-augmented control code generation with LLMs as proposed in this paper.

8 CONCLUSIONS

We introduced a novel retrieval-augmented code generation method that can integrate pre-built function blocks into synthesized IEC 61131-3 ST code. The method suggests creating text embeddings for function block specifications (e.g., from reference manuals), storing them in a vector store, and using a similarity search to augment control code generation prompts with information about the blocks. Tests using a prototypical implementation based on open-source tooling demonstrated the feasibility of the method.

The method can save control engineers significant time in implementing control logic, as the prompts are easy to formulate, can be extracted out of requirements documents (control narratives), and the code generation can be performed in a few seconds. Control engineers can use the function blocks provided by a specific automation vendor and use their familiar programming environments. It is conceivable to run the method in a batch mode for a large series of control logic generation prompts and let the control engineer only supervise the process and check the results.

Researchers can use our method and tooling as a template for enhancing retrieval-augmented control logic generation. The method could for example be combined with test case generation to check the generated code's quality automatically. Various parameters used in the method (e.g., for document splitting, configuring the similarity search, or refining the outputs) could be optimized for specific contexts. Converters could be added to the method to support other notations than IEC 61131-3 ST.

In follow-up work, we intend to apply the method to commercial function block libraries and PLC programming environments. The code generation capabilities of the method shall be tested in more comprehensive experiments, testing different kinds of control logic generation prompts and many different function blocks. If successful, the possibilities and limitations for non-interactive batch runs of the method shall be investigated.

REFERENCES

[1] 2023. OSCAT - Open Source Community for Automation Technology. http://www.oscat.de/.

[2] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio De Souza, and Thelma Virginia Rodrigues. 2014. OpenPLC: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference (GHTC 2014)*. IEEE, 585–589.

[3] Harrison Chase. 2023. LangChain: framework for developing applications powered by language models. https://github.com/langchain-ai/langchain.

[4] Enrique Dehaerne, Bappaditya Dey, Sandip Halder, Stefan De Gendt, and Wannes Meert. 2022. Code generation using machine learning: A systematic review. *Ieee Access* (2022).

[5] Rainer Drath and Idar Ingebrigtsen. 2018. Digitalization of the IEC PAS 63131 standard with AutomationML. In *2018 IEEE 23rd International Conference on Emerging Technologies and Factory Automation (ETFA)*, Vol. 1. IEEE, 901–909.

[6] Xueying Du, Mingwei Liu, Kaixin Wang, Hanlin Wang, Junwei Liu, Yixuan Chen, Jiayi Feng, Chaofeng Sha, Xin Peng, and Yiling Lou. 2023. Classeval: A manually-crafted benchmark for evaluating llms on class-level code generation. *arXiv preprint arXiv:2308.01861* (2023).

[7] Sten Grüner, Peter Weber, and Ulrich Epple. 2014. Rule-based engineering using declarative graph database queries. In *2014 12th IEEE International Conference on Industrial Informatics (INDIN)*. IEEE, 274–279.

[8] Georg Gutermuth. 2010. *Collaborative Process Automation Systems*. ISA, Chapter Engineering, 156–182.

[9] David Hästbacka, Timo Vepsäläinen, and Seppo Kuikka. 2011. Model-driven development of industrial process control applications. *Journal of Systems and Software* 84, 7 (2011), 1100–1113.

[10] Martin Hollender. 2010. *Collaborative process automation systems*. ISA.

[11] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. *arXiv preprint arXiv:2302.05020* (2023).

[12] Heiko Koziolek, Andreas Burger, Marie Platenius-Mohr, and Raoul Jetley. 2020. A classification framework for automated control code generation in industrial automation. *Journal of Systems and Software* 166 (2020), 110575.

[13] Heiko Koziolek, Andreas Burger, Marie Platenius-Mohr, Julius Rückert, Hadil Abukwaik, Raoul Jetley, and Abdulla P P. 2020. Rule-based code generation in industrial automation: four large-scale case studies applying the cayenne method. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*. 152–161.

[14] Heiko Koziolek, Sten Gruener, and Virendra Ashiwal. [n. d.]. ChatGPT for PLC/DCS Control Logic Generation. In *Proc. IEEE Int. Conf. on Emerging Technologies and Factory Automation (ETFA2023)* (2023-09-15).

[15] Heiko Koziolek, Sten Grüner, Rhaban Hark, Virendra Ashiwal, Sofia Linsbauer, and Nafise Eskandani. 2023. LLM-CodeGen-RAG Github Repository. https://github.com/hkoziolek/LLM-CodeGen-RAG.

[16] Heiko Koziolek and Anne Koziolek. 2023. LLM-based Control Code Generation using Image Recognition. *arXiv preprint arXiv:2311.10401* (2023).

[17] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474.

[18] Jiawei Liu, Chunqiu Steven Xia, Yuyao Wang, and Lingming Zhang. 2023. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210* (2023).

[19] Shangqing Liu, Yu Chen, Xiaofei Xie, Jingkai Siow, and Yang Liu. 2020. Retrieval-augmented generation for code summarization via hybrid gnn. *arXiv preprint arXiv:2006.05405* (2020).

[20] Shuai Lu, Nan Duan, Hojae Han, Daya Guo, Seung-won Hwang, and Alexey Svyatkovskiy. 2022. Reacc: A retrieval-augmented code completion framework. *arXiv preprint arXiv:2203.07722* (2022).

[21] Tomaž Lukman, Giovanni Godena, Jeff Gray, Marjan Heričko, and Stanko Strmčnik. 2013. Model-driven engineering of process control software–beyond device-centric abstractions. *Control Engineering Practice* 21, 8 (2013), 1078–1096.

[22] Md Rizwan Parvez, Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Retrieval augmented code generation and summarization. *arXiv preprint arXiv:2108.11601* (2021).

[23] Sida Peng, Eirini Kalliamvakou, Peter Cihon, and Mert Demirer. 2023. The impact of ai on developer productivity: Evidence from github copilot. *arXiv preprint arXiv:2302.06590* (2023).

[24] Frank Schumacher and Alexander Fay. 2014. Formal representation of GRAFCET to automatically generate control code. *Control Engineering Practice* 33 (2014), 84–93.

[25] Michael Steinegger and Alois Zoitl. 2012. Automated code generation for programmable logic controllers based on knowledge acquisition from engineering artifacts: Concept and case study. In *Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012)*. IEEE, 1–8.

[26] Michael Tiegelkamp and Karl-Heinz John. 2010. *IEC 61131-3: Programming industrial automation systems*. Vol. 166. Springer.

[27] Priyan Vaithilingam, Tianyi Zhang, and Elena L Glassman. 2022. Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models. In *Chi conference on human factors in computing systems extended abstracts*. 1–7.

[28] Daniel Witsch and Birgit Vogel-Heuser. 2011. PLC-statecharts: An approach to integrate UML-statecharts in open-loop control engineering–aspects on behavioral semantics and model-checking. *IFAC Proceedings Volumes* 44, 1 (2011), 7866–7872.

[29] Frank F Xu, Bogdan Vasilescu, and Graham Neubig. 2022. In-ide code generation from natural language: Promise and challenges. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2022), 1–47.