AI vs. Dinosaurs – Automated Re-implementation of Legacy Mainframe Applications in Java by Combining Program Synthesis and GPT

Simon Fischer-Heselhaus[1] and Philipp Brune[1,2](✉)

[1] SQ Solutions GmbH, Platz der Einheit 2, 60327 Frankfurt am Main, Germany
Simon.Fischer@sq-solutions.de
[2] Neu-Ulm University of Applied Sciences, Wileystraße 1, 89231 Neu-Ulm, Germany
Philipp.Brune@hnu.de
https://sq-solutions.de, https://hnu.de

Abstract. Large-scale mainframe applications written in outdated languages such as COBOL still form the core of the enterprise IT in many organizations, even though their flexibility and maintainability declines continuously. Their manual re-implementation in modern languages like Java is usually economically not feasible. Automated code conversion of legacy programs usually produces poor quality code in the target language, even with recent AI tools such as ChatGPT. In addition, code conversion recovers dead or unnecessary code artifacts in the new language. Therefore, in this paper we explore a novel approach, which does not convert the legacy code, but instead uses the existing input/output data to generate program tokens through program synthesis. These tokens are subsequently translated into input tokens and submitted to ChatGPT to produce the target code. The approach is illustrated and evaluated by means of a semi-realistic example program. The obtained results look promising, but need to be further investigated.

Keywords: Program Synthesis · Legacy Modernization · AI · COBOL · GPT · Large Language Models

1 Introduction

For many years, possible ways how to deal with the growing amount of program code considered "legacy" is one of the grand challenges of software engineering [21]. State-of-the-art architectures and technologies are often a prerequisite for moving applications to the cloud, create new digital services and business models, and integrate with new technologies like e.g. blockchain or AI [3,28].

However, modernization of legacy applications remains a big challenge for various reasons. One is the lack of developers who are familiar with its code and have the skills to maintain and update it. This is getting worse when developers

© The Author(s), under exclusive license to Springer Nature Switzerland AG 2024
S. Bouzefrane et al. (Eds.): MSPN 2023, LNCS 14482, pp. 205–221, 2024.
https://doi.org/10.1007/978-3-031-52426-4_14

retire or move to other fields, thus losing their specific knowledge [32]. Another challenge is the lack or inadequacy of documentation [?]. In some extreme cases, even the source code may be missing, meaning that the program cannot be changed at all.

Additionally, legacy code can lead to vendor lock-in. This means that companies are constrained by specific programming languages, platforms, or technologies that were selected in the past. This can make it difficult to switch to more modern technologies or vendors and create additional costs and risks.

Legacy code represents a significant burden for many organizations, and the pressure to transform or migrate such systems continuously grows. One reason is the sheer volume of existing code. It is estimated that there are between 180 and 200 billion lines of such code worldwide [8]. This figure becomes even worse considering that the global code base is growing by billions of lines annually [7].

Automating transformation and reimplementation projects in software development are a major challenge, and many of these projects encounter significant difficulties [28,33]. One important reason is that many programs depend on proprietary middleware considered outdated today., like e.g., old database systems or transaction monitors.

The use of exotic or less common programming languages can also be an obstacle, as expertise in such languages is scarce, and transformation tools may be lacking. In addition, the quality of legacy code is often already poor, and automated transformation could actually degrade code quality further. In addition, there are concepts related to programming languages that cannot be easily transferred to other programming languages. A typical example is imperative code that cannot be directly transferred to object-oriented programming languages, leading to ineffective code after an automated transformation [29].

However, given the growing amount of legacy code, automated code transformation nevertheless remains an increasingly important topic in research and practice [29]. Manual code transformation is usually both time-consuming and costly [13]. Moreover, it requires specialized knowledge and, for large code bases, can lead to errors that are difficult to detect and fix [3].

Therefore, it is essential to develop more effective tools for automated code transformation to efficiently and cost-effectively transition legacy code into modern systems [13].

Recent advances in large language models (LLM) due to the introduction of generative pretrained transformer (GPT) architectures using attention mechanisms [10,35], popularized by services like ChatGPT[1], have revolutionized the art of programming and transformed software development in many ways. Despite these advances, there are some restrictions limiting their usefulness for legacy code transformation:

- LLM can translate code, but often do so without a deep understanding of the underlying concept or original purpose [18].
- There is a risk that LLM translate code that may be redundant or no longer have a use in modern systems.

---

[1] https://openai.com.

- Even if LLM are able to translate legacy code, different programming paradigms (e.g., imperative vs. object-oriented) may cause the performance of the transformed code to suffer.
- For a successful code transformation, LLM often require the full source code or specific details from the documentation describing what functions the code should perform.

Instead of LLM, in order to reproduce the relationships between inputs and outputs, deep learning models could be used, but the resulting program surrogate, in the form of a trained neural network, could neither guarantee that inputs always generate the expected outputs, nor is it verifiable or testable [9,27,31].

Therefore, in this paper a hybrid approach is proposed, which combines program synthesis, which can establish relationships between inputs and outputs but does not suffer from the black box problem, and use a LLM to generate the surrounding code.

The rest of this paper is organized as follows: In Sect. 2 the related work is analyzed in more detail. Section 3 describes the proposed solutions, which is experimentally evaluated by applying it to a semi-realistic COBOL program as described in Sect. 4. The results of the evaluation are presented in Sect. 5. We conclude with a summary of our findings.

2 Related Work

Legacy application modernization has been a topic for many years in research and practice [3,15,21,28].

Various approaches to automatic code transformation have been considered in the literature over the years. However, mainly symbolic, rule-based [15] or simple statistical methods have been investigated, mostly also only for restricted problem classes such as clone detection [19,34].

Only few recent works discuss approaches for using ML or AI methods in the context of code analysis or transformation, e.g., again for code clone detection [36–38]. However, approaches that use e.g. new approaches like GPT are missing so far. On the other hand, initiatives such as AI for Code are currently in the process of establishing a basis for such approaches by creating corresponding databases of program code (Codenet) [22].

As only little previous work exists on the use of AI for legacy application modernization, in this paper a novel hybrid approach is proposed, combining the earlier works on program synthesis with the recent LLM breakthrough by ChatGPT.

The theory of program synthesis dates back to the 1950s [11]. It was extensively defined and elaborated in the 1970s. The main problem of program synthesis, to automatically construct programs to match a given specification, has long been a research topic in computer science [4,20,30].

Thanks to increased computing power, the implementation of these ideas is more feasible today. A wide variety of approaches exist in the area of program synthesis, but a unified definition is lacking so far [16,23,40].

The predominant approaches as described by Si et al. [27] can be classified into three types: The **neural approach**, in which pseudocode is generated from input-output examples, the **symbolic approach**, which synthesizes based on specifications, and the **inductive approach**, in which a neural network acts as a surrogate for a program.

In particular, the symbolic approach can be a promising way forward in certain circumstances, but only if the specifications are sufficiently concrete and precise.

In the case of legacy code, as already mentioned, this is often not the case. The inductive approach, on the other hand, confronts researchers and developers with the well-known black box problem and is therefore often not applicable in sectors with extremely strict regulations, such as banking and insurance.

In this work, we focus on the Programming by Example (PBE) approach [39]. Well-known implementations include *RobustFill* (for string analysis), *AlphaRegex* (for regular expression generation), and *DeepCoder* (for integer and integer list analysis).

A particular advantage of program synthesis is evident in the legacy context: Many large corporations, banks, and insurance companies that use legacy code have an extensive database of input-output examples. These IO-based approaches theoretically do not require code. Moreover, they are language-independent and can therefore be applied to different programming languages. Despite the black-box nature of the underlying AI model, the results are editable and validatable [17].

However, there are challenges. These include problems due to "noise" in data [27], difficulties in handling loops [12], limited model availability, and limited mapping capability through Domain Specific Languages (DSL) [23].

Criticism also includes the overuse of synthetic data and the resulting insufficient semantic mastery, as well as limited generalization and accuracy rates [24]. In addition, the models are currently limited to integers, which leads to problems in representing floating point numbers and other data types.

In conclusion, the flexibility of DSL is both an advantage and a challenge. While it can be extended to include arbitrary constructs, an overly elaborate DSL increases model complexity.

All these points illustrate, that program synthesis alone is not sufficient to build a solution for legacy code transformation, which would be useful in practice. However, combining it with the power of LLM, e.g. in the form of ChatGPT, could bring a new quality to it.

Therefore, in this paper the question is addressed, how a combination of an existing approach for program synthesis with ChatGPT for the actual code generation could be applied to transform legacy code to modern languages like e.g. Java, and what could be expected with respect to the quality of the resulting code.

3 Design of the Solution

The functional core of the program candidate to be replaced, and thus the starting point for the LLM query shall be the relationship between input and output established by the program synthesis.

This core shall then be enriched step by step by further information, as illustrated in Fig. 1.

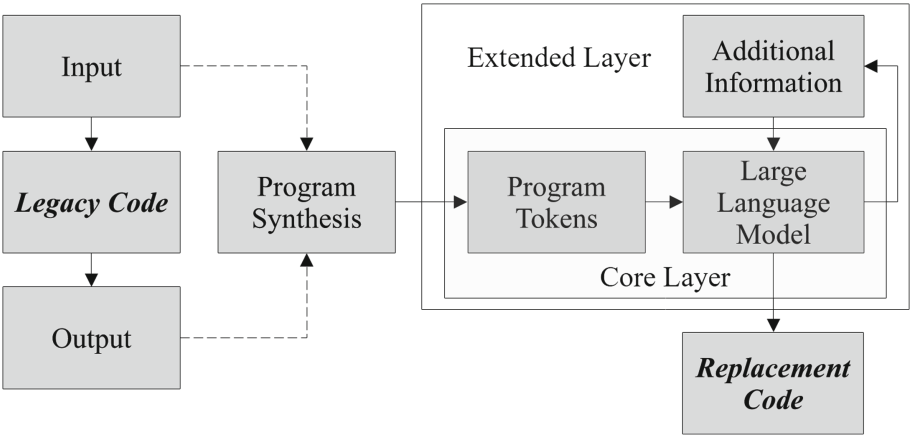


Fig. 1. Hybrid and multi-layered approach, with information generated by program synthesis at the core, enhanced with further information (e.g. embedding in desired framework).

The implementation of program synthesis in a real-world context follows a multistep process. This process can be divided into the following steps, as shown in Fig. 2.

1. **Extraction of inputs and outputs:** First, the necessary inputs and outputs are extracted from the available data.
2. **Generation of data combinations:** A Data Variator is used to generate all possible combinations of inputs and outputs. This allows to identify potential sub-combinations in the data.
3. **Application of program synthesis:** The next step is to perform the actual program synthesis.
4. **Translation to LLM query:** The result of the program synthesis is translated into an LLM query and submitted to the LLM system
5. **Code generation:** Based on the query, the LLM system generates the associated code.
6. **Optimization of query:**
   (a) **Refining:** Optimize the query by adding additional information from source code, comments, documentation and other sources.

(b) **Extension:** For more comprehensive code generation, information about frameworks, interfaces and other relevant aspects are integrated into the query.

This approach ensures a structured and comprehensive approach to program synthesis, generating high-quality and relevant code.

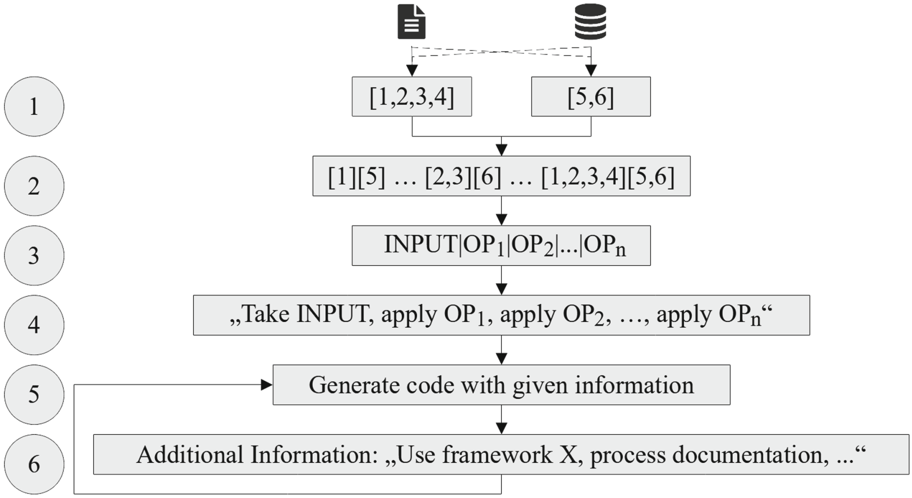

1
2
3
4
5
6
[1,2,3,4]
[5,6]
[1][5] … [2,3][6] … [1,2,3,4][5,6]
INPUT|OP1|OP2|...|OPn
„Take INPUT, apply OP1, apply OP2, …, apply OPn“
Generate code with given information
Additional Information: „Use framework X, process documentation, ...“

Fig. 2. Procedure to generate code using LLM and program synthesis, restricted to (lists of) integer values from different data sources

Program Synthesis: Per-Example Program Synthesis

PEPS (Per-Example program synthesis) aims to create a pseudo-program p_g that satisfies a set X consisting of N IO examples, each represented as an input-output pair. This program, consisting of T lines, is identified within a specified time limit. The behavior and structure of this program are influenced by a domain-specific language (DSL) originally introduced for DeepCoder [2].

The DSL presented by Balog et al. includes first-order functions such as `SORT` and `REVERSE` and higher-order functions such as `MAP` and `FILTER`. These functions can handle lambda functions as inputs. The DSL supports both integers and lists of integers as inputs and outputs.

Unlike PCCoder, which is based on a *Global Program Search (GPS)* approach and processes all IO examples simultaneously, N-PEPS implements an approach that handles each example individually. In this approach, PCCoder builds programs incrementally based on a concept of a program state. The state of this program is a $N \times (\nu + 1)$ dimensional memory created by executing t program steps over N inputs. The model consists of two neural components: H_θ and W_ϕ.

While H_θ generates the embedding of the current state, W_ϕ predicts three key variables for the next program line.

The model is trained with multiple instances of the specification X and the actual program p_g. For inference, complete anytime bar search (CAB) is used, stopping either when a solution is reached or when the maximum time is reached.

The main idea behind N-PEPS is to decompose the given N IO examples into N individual subproblems and then reassemble them later. Within N-PEPS, the module responsible for detecting PE solutions is called the PE search module.

The Cross Aggregator (CA) approach aims to detect the next line in the global solution p_g based on the execution data of individual programs. This is done using a cross-attention mechanism where the key and value matrices are generated from the PE program state embeddings and their subsequent lines.

To achieve the synthesis of a global solution p_g, the Cross Aggregator (CA) employs a neural network-based architecture with a multi-head attention mechanism. Instead of relying on elementary aggregation methods, the CA harnesses program execution state information, comparing the global solution's current state to those of preliminary solutions (PE solutions). By using a cross-attention mechanism, it determines the relevance of PE solution states, positing that a closely aligned PE state can provide significant cues for the next line of the global solution. This strategy ensures optimal combinations of PE solutions, proving especially crucial in scenarios where the comprehensive solution requires non-trivial combinations or introduces statements not found in the PE solutions.

The N-PEPS model mirrors one layer of the transformer encoder block. However, due to computation time constraints, the framework consists of only one layer. The key-to-value mapping in the N-PEPS model plays a critical role in the process.

Overall, N-PEPS enables efficient synthesis of programs based on IO examples. By combining domain-specific languages, neural models, and specialized search techniques, N-PEPS offers some improvements in program synthesis compared to previous models.

In the context of the development of program synthesis models, the evolution of the models can be seen as a progressive process that started with DeepCoder, evolved through PCCoder, and finally culminated in the development of N-PEPS. In this process, N-PEPS builds directly on the DSL originally developed for DeepCoder.

This ensures continuity in the application and development of the models, while allowing to build on the knowledge and functionality of the previous models. In terms of performance, significant improvements were observed; the results of N-PEPS outperformed those of PCCoder, which ultimately led to N-PEPS being selected as the preferred model. This indicates that the ongoing development and refinement of models in the field of program synthesis is leading to increasingly efficient and powerful solutions.

Large Language Models: General Pretrained Transformer

For the natural language processing application domain, the LLM in the form of ChatGPT (version 4) was chosen. The reasons for this choice are:

- Availability: ChatGPT is currently widely available.
- API access: ChatGPT's interface provides user-friendly and easy access, which facilitates integration and application in various contexts.
- ChatGPT is able to generate working code, often of good quality [1,5].
- ChatGPT has limited learning capabilities (few-shot/zero-shot learning) [5,6,14]

Prompt

Is there a relationship between the following inputs and outputs of a program?
Input=7,2,0, Output=18

ChatGPT Response

Given the provided input and output:
Input: 7, 2, 0 Output: 18
Let's explore possible relationships:
Looking at the numbers, a straightforward operation like addition results in:
7+2+0=9
However, that's not 18. So, addition doesn't seem to be the operation in question.
Another approach could be multiplication:
7×2×0=0
That's not the desired output either.
One might also consider other operations such as concatenation, but 7,2,0 would concatenate to 720720, which isn't 18.
With the information provided, there isn't a clear and direct arithmetic relationship that gives an output of 18 from the given input. It's possible that the program uses a more complex or non-trivial operation, or there might be an error in the provided data.
If you have more input-output pairs or additional context, it would be easier to deduce a potential relationship.

Fig. 3. An example of ChatGPTs limited reasoning abilities.

ChatGPT has significant capabilities in terms of generating coherent code. Nevertheless, it has inherent limitations in terms of deep understanding of context, especially when it comes to complex links between inputs and outputs. Although it is capable of responding to the information provided to it and generating responses based on its extensive training, it lacks the human ability to intuit and consider context beyond the immediate data, as seen in Fig. 3.

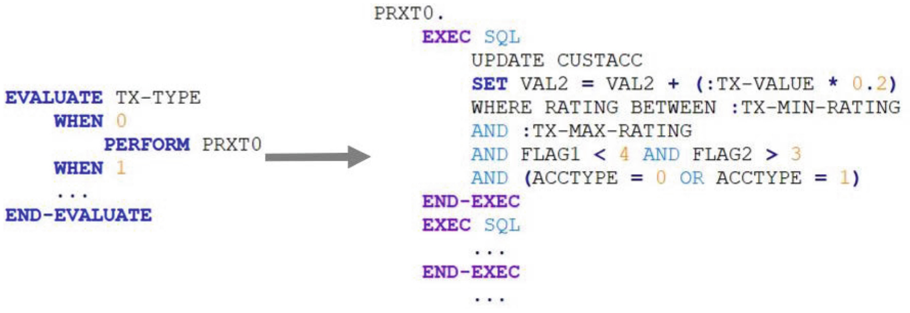
A simple solution, which is not directly obvious to ChatGPT, is to add all the numbers and then multiply by two. ChatGPT itself points out that there is *(a)* a lack of additional information and *(b)* a lack of input-output pairs. Even with an additional input-output pair, ChatGPT does not manage to establish a simple connection between input and output. The multiplier that might be "hidden" in a program is invisible to ChatGPT's analysis capabilities.

4 Experimental Evaluation

COBOL modules, which perform Db2 and dataset accesses, form the basis of this study. Publicly available repositories often only provide technology demonstrations or are too complex for demonstration purposes.

Therefore, self-created modules are used as a basis. A JCL batch job is composed of two COBOL modules: *TXPROC* and *SCORES*. *TXPROC* reads one transaction record each from a dataset. This record consists of a "type", a "minimum rating", a "maximum rating" and a "value". Depending on the "type", different processes are triggered, as illustrated in Fig. 4. The input lists of the IO pairs for the program synthesis are generated from the respective transaction record and the unchanged rows from the table CUSTACOT.

The table CUSTACOT represents a backup of CUSTACC before changes were made by TXPROC. The output lists of the IO pairs consist of the rows from CUSTACC changed by TXPROC, i.e. the delta between CUSTACC and CUSTACOT.

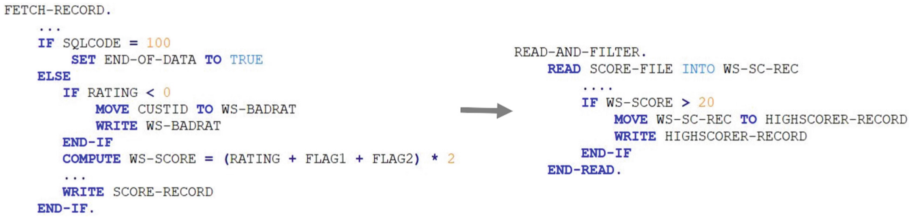


```
EVALUATE TX-TYPE
    WHEN 0
        PERFORM PRXT0
    WHEN 1
    ...
END-EVALUATE
```

```
PRXT0.
    EXEC SQL
        UPDATE CUSTACC
        SET VAL2 = VAL2 + (:TX-VALUE * 0.2)
        WHERE RATING BETWEEN :TX-MIN-RATING
        AND :TX-MAX-RATING
        AND FLAG1 < 4 AND FLAG2 > 3
        AND (ACCTYPE = 0 OR ACCTYPE = 1)
    END-EXEC
    EXEC SQL
        ...
    END-EXEC
        ...
```

Fig. 4. Code excerpt of COBOL module *TXPROC*

SCORES calculates a SCORE for each customer and records it in the corresponding dataset. This is calculated from (RATING + FLAG1 + FLAG2) * 2. Customers with a negative rating receive an additional entry in a BADRATING dataset. Following the SCORE processing, another dataset with HIGHSCORES is created. For this purpose, all entries with a score > 20 are transferred from the SCORE dataset to the HIGHSCORE dataset, as shown in Fig. 5.



```
FETCH-RECORD.
    ...
    IF SQLCODE = 100
        SET END-OF-DATA TO TRUE
    ELSE
       IF RATING < 0
           MOVE CUSTID TO WS-BADRAT
           WRITE WS-BADRAT
       END-IF
       COMPUTE WS-SCORE = (RATING + FLAG1 + FLAG2) * 2
       ...
       WRITE SCORE-RECORD
    END-IF.
```

```
READ-AND-FILTER.
    READ SCORE-FILE INTO WS-SC-REC
        ....
        IF WS-SCORE > 20
            MOVE WS-SC-REC TO HIGHSCORER-RECORD
            WRITE HIGHSCORER-RECORD
        END-IF
    END-READ.
```

Fig. 5. Code excerpt of COBOL module *SCORES*

The pre-trained model “E2” from the N-PEPS authors’ Git repository [25] is used for synthesis. Default parameters, see [26], are used and synthesis is applied directly to the inputs and outputs of the COBOL modules. The Data Variator is used to prepare the input-output pairs. The main goal is to identify partial relations between the different IO pairs. One challenge here is the combinatorial explosion, which allows up to $n!$ for $n = |InputVariables| + |OutputVariables|$ combination possibilities. This makes it difficult to check all combinations, especially for large programs.

The output of the program synthesis is transformed into tokens using a Python script, which can then be processed by ChatGPT.

Applying program synthesis methods to the given COBOL modules revealed some interesting observations.

Module 1: TXPROC

While examining the *TXPROC* module, we encountered the several difficulties. Program synthesis could not identify a direct path from inputs to outputs. This suggests that the relationship between the given inputs and outputs in *TXPROC* is non-trivial or not directly representable by the synthesis methods used.

The module partially accesses data of type ‘Decimal’ in Db2, respectively ‘PIC S9(5)V9(4) USAGE COMP-3’ in COBOL, and respectively ‘BigDecimal’ in Java. The mapping of these data types to the ‘Integer’ data type was required, resulting in compression losses. This could have an impact on the ability of synthesis to find correct and accurate paths.

Furthermore, the application of the Data Variator was only possible to a limited extent in this case. The main problem was the sheer number of possible combinations that could be generated by the Data Variator. This makes it impractical to analyze all possible combinations in an efficient amount of time.

Module 2: *SCORES*

As described the module is able to write entries in three datasets. This is achieved by the processing steps already described. In the following, the steps are divided into three logical units: SCORE, to calculate the score, HIGHSCORE, if the score is greater than 20 and BADRATING, if the rating is negative.

To calculate the SCORE, input data from the database were first connected to the corresponding output data from the dataset. For simplicity, it was assumed that the customer ID in the database has an equivalent in the dataset.

The resulting IO pairs were processed with the Data Variator and inputs for program synthesis were generated from them. An input for the program synthesis could look as follows:

```
{"examples":[
        {"output":[38,99],"inputs":[[99,1,5,7,7,-245,-230]]},
        {"output":[24,17],"inputs":[[17,3,0,5,7,-251,-253]]},
        {"output":[22,7],"inputs":[[7,0,3,7,1,-177,-200]]}
    ]
}
```

The program synthesis applied to the IO pairs shown here could not produce a result. Since we assume that one value of the output is the customer ID in each case, the focus is on the other value of the output list.

So the crucial "examples" are the following:

```
{"examples":[
        {"output":38,"inputs":[[5,7,7]]},
        {"output":24,"inputs":[[0,5,7]]},
        {"output":22,"inputs":[[3,7,1]]}
    ]
}
```

In the first test run, the following token, using the said DSL, was generated:

```
LIST|MAXIMUM,0|TAKE,1,0|ZIPWITH,+,2,2|SUM,3
```

For the first IO pair, this would mean the following: Take the maximum of the list $a = [5, 7, 7]$. Create a new *list* b containing all elements of the input list after the maximum (here: 7). Take the *list* b and the *list* b, add all elements in order, create *list* c from it. Sum all elements of *list* c. For the specified IO pairs, this procedure makes sense, but the first two steps do not seem necessary, since the lists are shorter than the "maximum". Apart from this, this could cause errors under corresponding conditions.

After adding another IO pair, namely `{"output":6, "inputs":[[1, 1,1]]}`, the program synthesis was able to generate a shortened, permissible token:

```
LIST|ZIPWITH,+,0,0|SUM,1
```

Using Python script, a prompt for ChatGPT is automatically generated from the pseudocode. When ChatGPT is asked to apply the steps to a given input, it is able to generate a corresponding output, as illustrated in Fig. 6 and Fig. 7.

Prompt

Apply the following logic if Input$_0$=$[7, 2, 0]$ and give a solution which is equal to the last input$_X$:

1. Input$_1$ = Take Input$_0$ and Input$_0$, add both elements of the lists according to their order.
2. Input$_2$ = Sum up all elements of Input$_1$.

Fig. 6. Example for ChatGPT query with token generated by program synthesis translated into natural language

ChatGPT Response

```
public int compute(int[] input_0) {
    int[] input_1 = new int[input_0.length];
    for (int i = 0; i < input_0.length; i++) {
        input_1[i] = input_0[i] + input_0[i];
    }
    int input_2 = 0;
    for (int value : input_1) {
        input_2 += value;
    }
    return input_2;
}
```

Fig. 7. Response to ChatGPT query with token generated by program synthesis translated into natural language

When asked to translate the functions into Java code as compact as possible, ChatGPT responds as follows:

In the last step we feed ChatGPT with additional information: We assume that we know that the data is taken from a database with a table named "CUSTACC" and the columns "RATING", "FLAG1" and "FLAG2" are evaluated. With this information ChatGPT is able to deliver a SQL query that is close to the original query, as shown in Fig. 8:

ChatGPT Response

Given the requirements, it seems you want to double the values of each column in the tuple and then sum them up. Here's how this can be represented using a Db2 SQL query:

```
SELECT
    SUM(2 * RATING + 2 * FLAG1 + 2 * FLAG2) AS INPUT_2
FROM
    CUSTACC;
```

Explanation:

- We multiply each column in the tuple (`RATING`, `FLAG1`, `FLAG2`) by 2 to simulate adding the column to itself.
- We then sum up all these doubled values to produce `Input_2`.

This will give you the sum of the doubled values of all rows in the `CUSTACC` table. **Note:** This query assumes that `RATING`, `FLAG1`, and `FLAG2` are numeric fields in your database. If they are not, this query might not run as intended.

Fig. 8. Response from ChatGPT when asked for a SQL query after enriching the original ChatGPT query with details such as the table layout.

It was not possible to generate results for the HIGHSCORE routine by means of program synthesis. Although HIGHSCORE works similarly to SCORE, not all customer IDs appear in the HIGHSCORE dataset. Accordingly, the IO pairs sometimes look different:

```
{"examples":[
            {"output":38,"inputs":[[5,7,7]]},
            {"output":[],"inputs":[[5,1,0]]},
            {...}
      ]
}
```

If only the entries present in the HIGHSCORE dataset are matched with the database, a subset of the functionally equivalent IO pairs of the SCORE routine is created; accordingly, the control structure of HIGHSCORE would be lost.

There is no negative rating (BADRATING) in the table used, so no entry is created in the corresponding dataset. In a realistic scenario, there would be two explanations for this: Either it is dead code that is no longer applied or the occurrence of a negative rating it is a rare, if not critical, event.

5 Results

Applying program synthesis to legacy code provides an impressive way to automatically generate tokens, which can be used as a starting point for code generation via LLM.

The tokens generated by program synthesis offer the advantage of transparency. This differs from "black box" approaches and allows direct comparison with the original legacy code. A notable advantage of program synthesis is the automatic elimination of dead code. However, there is a risk that certain edge cases could be overlooked if insufficient I/O examples are provided.

In theory, program synthesis can be applied at any level - be it IO within a JCL job, within a COBOL module, a process or a subroutine. The only requirement is the ability to create a relation between inputs and outputs.

The case study highlighted the advantages and some pitfalls of the multilayered AI approach. In particular, program synthesis can react very strongly to small changes in input-output pairs. It should be noted that the case study chosen is deliberately simple. Shrivastava et al. give other, much more complex examples [26].

Among the challenges is the need to map various data types to the 'integer' type, which in some cases can prevent the use of program synthesis.

Although there are theorems for mapping data types into congruent (number) spaces by approximation, there is no way to map floating point numbers effectively, i.e. without loss of information, which means that correlations between inputs and outputs are easily lost [?].

Furthermore, data preparation requires a lot of time and effort. Nevertheless, a detailed knowledge of inputs and outputs enables the creation of input-output pairs suitable for program synthesis. A completely open application to inputs and outputs is possible, but the processing time increases considerably if every conceivable combination is evaluated by means of program synthesis.

In addition, it has been observed that program synthesis generates overly complex tokens in some situations, which could make the process inefficient. It could be shown that linking program synthesis and LLM provides an approach to migrate legacy code to other programming languages even with little information.

In addition, the use of an LLM ensures that there is flexibility with regard to the target environment, represented here in the form of Java code and SQL query. However, it also became apparent that the code quality increases if the LLM is provided with appropriate additional information.

6 Conclusion

In conclusion, in this paper a new hybrid approach for AI-based transformation of legacy code to modern programming languages was presented. By combining program synthesis with GPT, it avoids translating the original legacy code, but recovers its actual functionality and generates a new program for it. The approach was evaluated in an explorative way in first experimental setup using a semi-realistic COBOL program as example.

The results are promising, but further research and development are needed to improve and expand it regarding the following aspects: First, stochastic methods, in particular cluster analysis, should be integrated into the overall workflow. By pre-analyzing the relationships between inputs and outputs, it would enable to perform inferences about control structures in legacy code. It could also help reduce the sheer volume of possible combinations for the Data Variator. If certain inputs have particularly strong relationships to specific outputs, these correlations could be brought to the forefront, leading to more efficient and targeted analyses.

Second, the constraints of the input and output values to an integer number range needs to be removed. The key questions are: What limits are reasonable? What range of values actually occurs in real inputs and outputs? And can there be alternative models or more efficient mappings that exceed or replace the integer number range?

Finally, the approach needs to be integrated in a software tool to provide a higher degree of automation and to make the program synthesis process more efficient and user-friendly. This will help to achieve faster and more accurate results, especially in complex legacy code environments.

References

1. Aljanabi, M., Ghazi, M., Ali, A.H., Abed, S.A., ChatGpt: ChatGpt: open possibilities. Iraqi J. Comput. Sci. Math. **4**(1), 62–64 (2023). https://doi.org/10.52866/20ijcsm.2023.01.01.0018. https://journal.esj.edu.iq/index.php/IJCM/article/view/539
2. Balog, M., Gaunt, A.L., Brockschmidt, M., Nowozin, S., Tarlow, D.: DeepCoder: Learning to Write Programs (2017). eprint: 1611.01989
3. Bandara, C., Perera, I.: Transforming monolithic systems to microservices - an analysis toolkit for legacy code evaluation. In: 2020 20th International Conference on Advances in ICT for Emerging Regions (ICTer), pp. 95–100 (2020). https://doi.org/10.1109/ICTer51097.2020.9325443
4. Biermann, A.W.: The inference of regular LISP programs from examples. IEEE Trans. Syst. Man Cybern. **8**(8), 585–600 (1978). https://doi.org/10.1109/TSMC.1978.4310035
5. Borji, A.: A Categorical Archive of ChatGPT Failures (2023). eprint: 2302.03494
6. Brown, T.B., et al.: Language Models are Few-Shot Learners (2020). eprint: 2005.14165
7. Ciborowska, A., Chakarov, A., Pandita, R.: Contemporary COBOL: developers' perspectives on defects and defect location. In: 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), pp. 227–238 (2021). https://doi.org/10.1109/ICSME52107.2021.00027
8. Di Nucci, D., et al.: A Language-Parametric Modular Framework for Mining Idiomatic Code Patterns (2019)
9. Dwivedi, R., et al.: Explainable AI (XAI): core ideas, techniques, and solutions. ACM Comput. Surv. **55**(9) (2023). https://doi.org/10.1145/3561048

10. Floridi, L., Chiriatti, M.: GPT-3: its nature, scope, limits, and consequences. Mind. Mach. **30**, 681–694 (2020)
11. Friedman, J.: Alonzo Church. Application of recursive arithmetic to the problem of circuit synthesisSummaries of talks presented at the Summer Institute for Symbolic Logic Cornell University, 1957, 2nd edn., Communications Research Division, Institute for Defense Analyses, Princeton, NJ, 1960, pp. 3–50. 3a–45a. J. Symb. Logic **28**(4), 289–290 (1963)
12. Gupta, K., Christensen, P.E., Chen, X., Song, D.: Synthesize, Execute and Debug: Learning to Repair for Neural Program Synthesis (2020). eprint: 2007.08095
13. Kiran Mallidi, R., Sharma, M., Singh, J.: Legacy digital transformation: TCO and ROI analysis. Int. J. Electr. Comput. Eng. Syst. **12**(3), 163–170 (2021)
14. Kojima, T., Gu, S.S., Reid, M., Matsuo, Y., Iwasawa, Y.: Large Language Models are Zero-Shot Reasoners (2023). eprint: 2205.11916
15. Kontogiannis, K.A., DeMori, R., Merlo, E., Galler, M., Bernstein, M.: Pattern matching for clone and concept detection. Autom. Softw. Eng. **3**(1), 77–108 (1996)
16. Laich, L., Bielik, P., Vechev, M.: Guiding program synthesis by learning to generate examples. In: International Conference on Learning Representations (2020). https://openreview.net/forum?id=BJl07ySKvS
17. Liventsev, V., Grishina, A., Härmä, A., Moonen, L.: Fully autonomous programming with large language models. In: Proceedings of the Genetic and Evolutionary Computation Conference. ACM (2023). https://doi.org/10.1145/3583131.3590481
18. Mahowald, K., Ivanova, A.A., Blank, I.A., Kanwisher, N., Tenenbaum, J.B., Fedorenko, E.: Dissociating language and thought in large language models: a cognitive perspective (2023). eprint: 2301.06627
19. Manilov, S.Z.: Analysis and transformation of legacy code (2018)
20. Manna, Z., Waldinger, R.J.: Toward automatic program synthesis. Commun. ACM **14**(3), 151–165 (1971)
21. Montakab, C.: Legacy transformation white paper. Technical report, softwaremining (2009). https://softwaremining.com/download/pdf/SM_Legacy_Transformation_whitepaper.pdf
22. Puri, R., et al.: Project codenet: a large-scale AI for code dataset for learning a diversity of coding tasks. arXiv preprint arXiv:2105.12655 1035 (2021)
23. Shi, K., Hong, J., Zaheer, M., Yin, P., Sutton, C.: Compositional Generalization and Decomposition in Neural Program Synthesis (2022). eprint: 2204.03758
24. Shin, R., et al.: Synthetic Datasets for Neural Program Synthesis. CoRR abs/1912.12345 (2019). http://arxiv.org/abs/1912.12345
25. Shrivastava, D., Larochelle, H., Tarlow, D.: N-PEPS. https://github.com/shrivastavadisha/N-PEPS
26. Shrivastava, D., Larochelle, H., Tarlow, D.: Learning to combine per-example solutions for neural program synthesis. In: Beygelzimer, A., Dauphin, Y., Liang, P., Vaughan, J.W. (eds.) Advances in Neural Information Processing Systems (2021). https://openreview.net/forum?id=4PK-St2iVZn
27. Si, X., Yang, Y., Dai, H., Naik, M., Song, L.: Learning a meta-solver for syntax-guided program synthesis. In: International Conference on Learning Representations (2019). https://openreview.net/forum?id=Syl8Sn0cK7
28. Sneed, H., Verhoef, C.: Re-implementing a legacy system. J. Syst. Softw. **155**, 162–184 (2019). https://doi.org/10.1016/j.jss.2019.05.012. https://www.sciencedirect.com/science/article/pii/S0164121219301050
29. Strobl, S., Zoffi, C., Haselmann, C., Bernhart, M., Grechenig, T.: Automated code transformations: dealing with the aftermath. In: 2020 IEEE 27th International

Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 627–631 (2020). https://doi.org/10.1109/SANER48275.2020.9054813. ISSN 1534-5351

30. Summers, P.D.: A methodology for LISP program construction from examples. J. ACM **24**(1), 161–175 (1977). https://doi.org/10.1145/321992.322002
31. Sun, J., et al.: Investigating explainability of generative AI for code through scenario-based design. In: 27th International Conference on Intelligent User Interfaces, Helsinki, Finland, IUI 2022, pp. 212–228. Association for Computing Machinery, New York (2022). https://doi.org/10.1145/3490099.3511119
32. Teplitzky, P.: Closing the COBOL Programming Skills Gap (2019). https://techchannel.com/Enterprise/10/2019/closing-cobol-programming-skills-gap
33. Terekhov, A., Verhoef, C.: The realities of language conversions. IEEE Softw. **17**(6), 111–124 (2000). https://doi.org/10.1109/52.895180
34. Trudel, M.: Automatic translation and object-oriented reengineering of legacy code (2013)
35. Vaswani, A., et al.: Attention is all you need. In: Advances in Neural Information Processing Systems, vol. 30 (2017)
36. Wavresky, F., Lee, S.W.: A methodology towards the adaptization of legacy systems using agent-oriented software engineering. In: Proceedings of the 31st Annual ACM Symposium on Applied Computing, pp. 1407–1414 (2016)
37. Weisz, J.D., et al.: Perfection not required? Human-AI partnerships in code translation. In: 26th International Conference on Intelligent User Interfaces, pp. 402–412 (2021)
38. White, M., Tufano, M., Vendome, C., Poshyvanyk, D.: Deep learning code fragments for code clone detection. In: 2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 87–98. IEEE (2016)
39. Ye, X., Chen, Q., Dillig, I., Durrett, G.: Optimal Neural Program Synthesis from Multimodal Specifications (2021). eprint: 2010.01678
40. Zhang, Y., Wang, D., Dong, W.: MerIt: improving neural program synthesis by merging collective intelligence. Autom. Softw. Eng. **29**(2), 45 (2022). https://doi.org/10.1007/s10515-022-00343-z