# Comparing LLM-based and Model-driven program translation

1st Kevin Lano
*Informatics Dept.*
*King's College London*
London, UK
kevin.lano@kcl.ac.uk

2nd Hanan A. Siala
*Informatics Dept.*
*King's College London*
London, UK
hanan.siala@kcl.ac.uk

3rd Kunxiang Jin
*Informatics Dept.*
*King's College London*
London, UK
kunxiang.jin@kcl.ac.uk

***Abstract*—Deep learning (DL) approaches for program translation have been extensively researched over recent years. These have been shown to be an effective alternative to manually-coded translators. At the same time, questions remain regarding the reliability and verifiability of DL approaches. Precise program translation approaches based on model-driven reverse engineering (MDRE) have also been researched. In this paper we provide the first detailed comparison of the results of the DL and MDRE approaches on tasks of COBOL translation to Python, Java and C#.**

***Index Terms*—Program translation, Deep learning, LLMs, Model-driven reverse engineering (MDRE).**

## I. Introduction

Program translation has been a long-standing concern within the software industry, motivated by the need to migrate software systems from one platform to another, or by the need to modernise legacy applications in archaic languages such as COBOL or VB into mainstream current languages such as Java.

Translation from COBOL is of particular interest to many businesses because of the large number of critical business systems which still exist in versions of COBOL [1]. The language originally dates back to the 1950's, and for at least 30 years it was the principal programming language used for business systems such as stock management, transaction processing, etc. Until the advent of powerful deep learning (DL) models such as pre-trained language models (PLMs) or large language models (LLMs), the main approach to COBOL translation was via explicit manually-coded rules [2]–[4]. Such translation projects usually involve substantial human and financial resources, moreover the effort is repeated across different projects, due to the high variation in system installations, custom versions or extensions of COBOL, specialised file or database systems, etc.

COBOL remains a challenge to both rule-based and DL approaches because of the archaic constructs and features of the language, in particular:

- Arbitrary transfer of control within a program by means of GO TO statements.
- Poor modularity – a program consists of global data definitions and a monolithic list of blocks of code grouped into 'sections' and 'paragraphs'.
- Large numbers of special-purpose statements, some with many different options and variants.
- Complex facilities such as the Report Writer, which are themselves specialised software languages.
- A machine-oriented view of data.

There is also a lack of monolingual or multilingual training data for COBOL translation, compared to more modern languages [5]. The large size of the language makes it difficult to assemble comprehensive training datasets for machine learning.

In this paper we survey DL and other program translation approaches, and carry out a comparative evaluation of a DL and a rule-based program translation approach, for COBOL translation.

## II. Background

Based on the successful use of DL for neural machine translation (NMT) of natural languages, similar NMT approaches were applied to the translation of software languages [6], [7]. However, these approaches used supervised learning and required the existence or creation of large scale parallel code datasets. A key advance was the application of unsupervised language modelling training to program translation in Transcoder [8]. This enabled the use of large monolingual code datasets to train a DL model with knowledge of the individual programming languages together with knowledge of language correspondences. Specifically, masked language modelling and denoising auto-encoding learning objectives were used together with coupled target-to-source and source-to-target training. The result was a PLM with significantly higher translation performance than manually-coded translators. Nevertheless, such an approach has limitations: it utilises common syntactic anchor points between programming languages, such as similar keywords $if$, $while$, etc., in different languages, and it operates in a stochastic manner. It is noticeable that the Transcoder translation accuracy is lower for more dissimilar languages (such as Python and Java) compared to more similar pairs (Java and C++). The scale of the back-translation training grows quadratically with the number of languages being considered. Deficiencies with the Transcoder results are discussed by [9].

Subsequent PLM approaches to program translation include GraphCodeBERT [10], the Transcoder-IR approach of [11], the CoTran approach of [12], and approaches utilising fine-tuning of PLMs [13], [14]. GraphCodeBERT uses data flow information to enable a PLM to learn more semantically-meaningful programming knowledge during pre-training. In [11], pre-training is enriched by the use of compiler intermediate representations, to increase the semantic awareness of the trained model. CoTran [12] incorporates semantic equivalence into the training objective by utilising automated test case generation. In order to improve the accuracy of Java-Python translation, a large parallel dataset for fine-tuning (supervised re-training) of a PLM was created by [13]. A similar fine-tuning approach was applied with a dataset of 7 programming languages in [14]. These approaches produced improved results compared to Transcoder, but still focussed on relatively similar pairs or groups of languages such as Java, JavaScript, C++, Go, Rust, Python, C#, etc, all with a common C-based heritage.

The advent of large language models (PLMs with at least 10 billion parameters [15]) brought a further change to the program translation landscape. Although LLMs are typically pre-trained on large general purpose datasets, and with simple language modelling objectives such as next token prediction, they have proven to be capable of carrying out diverse tasks as well as, or better, than specialised smaller-scale PLMs [15]. Apart from their baseline capabilities for program translation, general purpose LLMs such as ChatGPT can be enhanced for this task by fine-tuning, prompt tuning [16] or refinement [5].

Independently of ML approaches to program translation, manually-based re-engineering approaches using intermediate models and software specifications [2], [17], [18] led to the concepts of *model-driven reverse/re-engineering* (MDRE) [19] and *model-driven modernisation* (MDM) formalised by the Object Management Group (OMG) in their Architecture-driven modernisation (ADM) framework [20], [21]. These produce software models in notations such as the Unified Modeling Language (UML) and Object Constraint Language (OCL) from program code. Implemented in tools such as REMICS and AgileUML, MDRE enables multi-lingual translation between multiple programming languages, with some degree of assurance of semantic preservation [22]–[24]. However, the necessary program abstractors mapping code to specifications in UML/OCL, and code generations mapping specifications to code, need to be manually constructed.

Table I summarises the DL and MDRE approaches discussed above. It is noticeable that only the refinement LLM approach of [5] and the MDRE approaches of [23], [25] include coverage of COBOL, and the [5], [23] approaches will be compared in detail in Section IV (the tools of [25] were not available for evaluation).

TABLE I
PROGRAM TRANSLATION APPROACHES

| Approach | Method | Source languages |
|---|---|---|
| Transcoder [8] | PLM | C++, Java, Python |
| GraphCodeBERT [10] | PLM | Java, C# |
| Transcoder-IR [11] | PLM | C++, Rust, Go, Java |
| CoTran [12] | PLM | Java, Python |
| AVATAR [13] | Fine-tuning | Java, Python |
| CoST [14] | Fine-tuning | C++, Java, Python, C#, JS, PHP, C |
| FSCTrans [16] | LLM | C#, Java, Python |
| Refinement [5] | LLM | COBOL |
| AgileUML [23] | MDRE | Java, C, VB6, COBOL, JS, Pascal, Python |
| REMICS [22] | MDRE | Java, C#, Delphi |
| SOAMIG [25] | MDRE | Java, COBOL |
| XIRUP [26] | MDRE | J2EE, Java |

## III. COMPARED APPROACHES

Due to the poor intrinsic knowledge of COBOL present in existing LLMs, the approach of [5] adopts a series of filtering and refinement steps on the base LLM COBOL to Java translations to improve their correctness and readability. Steps of logical refinement (correction of generated Java based on compiler/execution feedback) and readability refinement are performed by the LLM, using its Java knowledge, which is considerably higher than its COBOL knowledge.

The approach of [23] also involves a series of steps, using the AgileUML toolset:

1) Abstraction of COBOL source code to a UML/OCL specification, using manually-coded abstraction rules written in the concrete grammar transformation language (CGTL) [27].
   Two simple abstraction rules, for COBOL compute statements, are:
   ```
   COMPUTE _1 EQUAL _2 END-COMPUTE  |-->
             _1 := _2 ;\n
   COMPUTE _1 EQUAL _2  |-->
             _1 := _2 ;\n
   ```
2) Possible restructuring of the UML/OCL specification within AgileUML, for example, to replace recursively-defined operations (originating from the abstraction of COBOL GO TO statements) by iterative definitions.
3) Code generation in the required target language, such as Java, C# or Python.
4) Via a bridge to the zAppDev low-code tools[1], further target implementations can be generated, for example for mobile or cloud platforms.

Table II shows an example translation from COBOL to Python using the AgileUML approach. The source program is example 1 in [5].

This example demonstrates that the AgileUML approach preserves program control flow, and that the program size does not change substantially (19 LOC of COBOL becomes 20 LOC Python). In contrast, the Java translation of the example by [5] has a radically different code structure (the program is divided into 4 methods) and control flow. The size is increased to 28 LOC.

[1]https://zappdev.io

TABLE II
COBOL AND PYTHON VERSIONS OF EXAMPLE 1 FROM [5] USING AGILEUML TRANSLATION

```
ACCEPT INP.
PERFORM ABLEN TIMES
  PERFORM VARYING J FROM CUR BY 1
  UNTIL INP ( J : 1 ) = SPACE
  END-PERFORM
  COMPUTE LEN = J - CUR
  MOVE INP ( CUR : LEN ) TO AB11 ( I )
  COMPUTE CUR = J + 1
  ADD 1 TO I
END-PERFORM.

COMPUTE DIV = AB11 ( 2 ) - AB11 ( 1 ) .
MOVE 0 TO S1 .
MOVE 1 TO I .

PERFORM DIV TIMES
  ADD I TO S1
  ADD 1 TO I
END-PERFORM.
```

```
self.INP = getOclFileByPK("System.in").readLine()
for _performTimes in range(1, self.ABLEN +1) :
  self.J = self.CUR
  while not
      (self.INP[(self.J-1):self.J + 1 - 1] == " ") :
    self.J = self.J + 1
  self.LEN = (self.J - self.CUR)
  self.AB11[self.I -1] =
     ocl.toInteger(
         self.INP[(self.CUR-1):self.CUR + self.LEN - 1])
  self.CUR = self.J + 1
  self.I = self.I + 1

self.DIV = (self.AB11[2 -1] - self.AB11[1 -1])
self.S1 = 0
self.I = 1

for _performTimes in range(1, self.DIV +1) :
  self.S1 = self.S1 + self.I
  self.I = self.I + 1
```

## IV. EVALUATION

We evaluate the refinement LLM approach and the AgileUML approach firstly in terms of their general performance for semantic accuracy and achieved readability and alignment measures, and then specifically compare their results on the five COBOL examples given with their Java translations in [5], Appendix C.

To evaluate achieved semantic accuracy we consider two measures: the *computational accuracy* (CA) measure of [8], computed as the percentage of all test cases which give the same result for the source and translated versions of the programs; and *execution accuracy* (EA), also used by [5], computed as the percentage of examples whose tests all give the same result when executed by the source and translated versions of the example.

As general measures for code readability factors, we consider (i) code size in lines of code (LOC); (ii) cyclomatic complexity (CC); (iii) the number of data, program/class and operation identifiers. These are established measures of code complexity and size [28].

To measure alignment between the source and target we also compute (i) the *completeness* of the translation wrt the source – the percentage of source identifiers which also appear in the target version translation; (ii) the *consistency* of the target wrt the source – the percentage of target identifiers which also occur in the source. We compute an overall *alignment* score as the harmonic mean of completeness and consistency:

$$alignment = \frac{2 * completeness * consistency}{completeness + consistency}$$

As examples of these measurement, in the case of Table II, all test cases are passed by the target code, so that CA and EA are both 1. The CC measure is 4 for both source and target versions, whilst there are 12 identifiers in the source and 16 in the target: the identifier SPACE has been removed and 5 additional identifiers $\_performTimes$, $getOclFileByPK$, $readLine$, $ocl$ and $toInteger$ have been introduced during the program abstraction stage to express the source program semantics. Completeness of the target relative to the source is therefore 0.92, and consistency is 0.69. Alignment is 0.79.

All data of the evaluation cases is available at https://zenodo.org/records/13359036.

### A. General comparison

The refinement LLM approach of [5] was applied to 322 COBOL code samples from the CodeNet platform. The samples each had test cases provided, and ranged in size from 11 LOC to 358 LOC. To evaluate the translation approach on these cases, execution accuracy (EA) using the CodeNet tests was used to assess preservation of the source code semantics. For evaluation of target code readability, different factors were considered: ease of understanding and maintenance; logical segmentation; low complexity; appropriate comments; meaningful naming conventions; testability; correct use of indentation. WizardCoder and GPT-3.5 LLMs were used for the translation and refinement processes. The optimal workflow combining these processes resulted in EA of 82% and a readability score of 0.62.

The 7 COBOL cases presented in [5] were selected for evaluation of the AgileUML approach, together with 3 real-world batch-processing examples. The largest case has size 159 LOC. Although the average size of the 10 cases (54.1 LOC) is small, the examples from [5] include cases of complex functionality taken from CodeNet programming problems. The batch-processing examples include use of the Report Writer and file processing.

Table III summarises the AgileUML translation results for these 10 cases.

The results of Table III show high semantic correctness values for the AgileUML translations. Full semantic correctness

TABLE III
AGILEUML TRANSLATION PROPERTIES

| Measure | COBOL | OCL | Java | C# | Python |
|---|---|---|---|---|---|
| CA [8] | – | 1.0 | 1.0 | 1.0 | 0.94 |
| EA [5] | – | 1.0 | 1.0 | 1.0 | 0.9 |
| Av. LOC | 54.1 | 68 | 87 | 108.2 | 107.8 |
| Size change from source | – | +25% | +60% | +100% | +100% |
| CC | 5.2 | 5.5 | 8.1 | 8.1 | 8.1 |
| #identifiers | 13.7 | 21 | 47.2 | 47.2 | 23.5 |
| Completeness wrt source | – | 0.9 | 0.9 | 0.9 | 0.88 |
| Consistency wrt source | – | 0.68 | 0.3 | 0.3 | 0.57 |
| Alignment | – | 0.77 | 0.45 | 0.45 | 0.69 |

is achieved except for one case of translation to Python. This is in contrast to the 0.82 EA correctness score for COBOL to Java translation reported in [5]. COBOL to C# EA of 0.86 and COBOL to Java EA of 0.75 are reported for an earlier version of AgileUML in [23]. Correctness problems encountered by [5] such as the difference between COBOL and Java array indexing (1-based versus 0-based indexing) are automatically handled by the AgileUML code generators.

In terms of the readability measures, code size is somewhat inflated by the AgileUML translation process, but the overall control flow structure is preserved, because conditional statements in COBOL map to conditionals in OCL and hence also in the target languages, and similarly for bounded and unbounded loops [24]. This is shown by the relatively small increases in cyclomatic complexity from the source to the model (6% increase on average) and to the target codes (56% increase on average). The increases are due to implicit conditions in the COBOL code, such as overflow behaviour, being made explicit in the semantic model. Likewise, the number of identifiers remains similar to the code in the abstracted model (53% increase) and in the Python translation (72% increase).

The high average completeness measure means that identifiers from the source are generally preserved in the translations, which helps software developers to understand the translations in terms of the source. Cases where source identifiers are lost are function identifiers such as $MOD$, which are replaced by binary operator symbols ($mod$ in OCL and $\%$ in the target programming languages) or symbolic constants such as $ZERO$, replaced by the actual value 0.

However, the consistency measures are lower, indicating that new identifiers are introduced in the translations. This is because single COBOL statements such as UNSTRING are interpreted as a sequence of OCL statements involving new temporary variables. Likewise, file buffer variables are made explicit in the OCL representation. In the Java and C# target programming languages, setter and getter operations for program variables are also introduced, this results in lower consistency measures for the mappings to those languages.

### B. Detailed comparison

To give a more precise characterisation of the differences between the two approaches, we evaluate them applied to the same examples and using the same metrics for accuracy, readability and alignment. We select as evaluation cases the five examples from [5], Appendix C. Table IV summarises the comparison of the refinement LLM and AgileUML translation results for these cases, considering translation from COBOL to Java. All figures are the averages taken over the five cases.

TABLE IV
AGILEUML AND REFINEMENT LLM COMPARISON

| Measure | Original code | AgileUML | Refinement LLM |
|---|---|---|---|
| *EA* | – | 1.0 | 0.8 |
| *LOC* | 42.4 | 57.2 | 30.6 |
| $\Delta$ *LOC* | – | +35% | -28% |
| *CC* | 4.2 | 4.2 | 2.8 |
| $\Delta$ *CC* | – | +0% | -33% |
| #identifiers | 11 | 34 | 12 |
| *Consistency* | – | 0.4 | 0.46 |
| *Completeness* | – | 0.93 | 0.49 |
| *Alignment* | – | 0.56 | 0.47 |

It can be seen from this table that the refinement LLM approach generally has higher readability measures than the AgileUML approach, producing target code with reduced average code size and CC compared to the source program, and compared to the AgileUML target code. The number of identifiers is marginally increased, whilst (for Java targets) AgileUML substantially increases the number of identifiers (mainly due to introduced setters/getters).

However for correctness, there is a significant error in the refinement LLM translation of the $caddi2018d$ program, which checks a series of $n$ input integer values to determine if any are odd. The LLM translation instead returns a program which adds up the number of even numbers from 1 to n. This is an case of LLM *hallucination* [15], where an LLM produces a plausible but incorrect result. The AgileUML translations are all semantically correct. The close structural correspondence between the source and target should help practitioners to verify this correctness.

In terms of alignment between the source and target, the refinement LLM approach can radically change the structure and set of identifiers between the source and target. Thus there is a lower completeness measure for this approach, compared to the AgileUML approach, which generally preserves the names of all program variables. The consistency measures of the approaches are similar.

### C. Summary of evaluation

Overall, we found that the accuracy of the precise translation approach of [23] is similar to or higher than the LLM approach of [5], whilst readability of the target is satisfactory, although generally lower than for the results of [5]. Alignment of the source and target, as measured by consistency and completeness, is generally higher for AgileUML translations.

Translation to multiple target languages is directly supported by AgileUML, whilst the approach of [5] needs to be customised for each different target language. The number of transformation steps applied by [5] may hinder traceability and developer's understanding of the correspondence between the target and source code versions, whilst in the absence of restructuring steps, the AgileUML approach maintains a consistent and close relation between the source and target code structures.

The completeness of the coverage of COBOL '85 or COBOL '74 language features is not specified in [5]. The examples described in [5] use only a restricted subset of COBOL: integer numerical computations and string and array processing. Coverage of 72% of COBOL '85 is claimed by [23], based on the Cobol85 grammar definition.

## V. Conclusions

In this paper we have provided the first detailed comparison of LLM-based and MDRE approaches to program translation, for translation of COBOL code. We found that the accuracy of the MDRE approach was generally comparable to or higher than the LLM approach, whilst readability was not significantly impaired. COBOL language coverage of the LLM approach is an open question.

Development of code LLMs with improved COBOL knowledge would improve the baseline performance of COBOL translation, and potentially lead to improved accuracy for DL translation approaches using these LLMs. Combination of LLM and MDRE approaches for program translation is also an important area to explore, by for example a MDRE translator utilising an LLM in cases that are outside the syntax input range of the MDRE tool. The MDRE approach could also be used to generate training datasets for LLM fine-tuning.

## Acknowledgment

The work presented here was funded by the EPSRC-IAA grant "Industrial demonstration case in verified re-engineering" with partners King's College London and CLMS UK Ltd.

## References

[1] S. Kerner, "COBOL language still in demand as application modernization efforts take hold," 2023. www.itprotoday.com.

[2] J. Bowen, P. Breuer, and K. Lano, "A compendium of formal techniques for software maintenance," *IEE/BCS Software Engineering Journal*, vol. 8, pp. 253 – 262, September 1993.

[3] A. D. Marco, V. Iancu, and I. Asinofsky, "COBOL to Java and newspapers still get delivered," in *Proceedings IEEE International Conference on Software Maintenance and Evolution*, pp. 583–586, IEEE Press, 2018.

[4] H. Sneed, "Migrating from COBOL to Java: A report from the field," in *IEEE Proc. of 26th ICSM*, pp. 1–7, IEEE Press, 2011.

[5] S. Gandhi, M. Patwardhan, J. Khatri, L. Vig, and R. Medicherla, "Translation of low-resource COBOL to logically-correct and readable Java leveraging high-resource Java refinement," in *LLM4Code*, 2024.

[6] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering*, ASE '15, p. 585–596, IEEE Press, 2015.

[7] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, (Red Hook, NY, USA), pp. 2552–2562, Curran Associates Inc., 2018.

[8] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised translation of programming languages," *arXiv:2006.03511v3*, 2020.

[9] A. Malyaya, K. Zhou, B. Ray, and S. Chakraborty, "On ML-based program translation: perils and promises," *arXiv:2302.10812v1*, 2023.

[10] D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, and M. Tufano, "GraphCodeBERT: Pre-training code representations with dataflow," in *ICLR 2021*, 2021.

[11] M. Szafraniec, B. Roziere, H. Leather, F. Charton, P. Labatut, and G. Synnaeve, "Code translations with compiler representations," *ArXiv:2207.03578v5*, 2023.

[12] P. Jana, P. Jha, H. Ju, G. Kishore, and A. Mahajan, "Attention, compilation, and solver-based symbolic analysis are all you need," *arXiv:2306.06755v1*, 2023.

[13] W. Ahmad, M. Tushar, S. Chakraborty, and K.-W. Chang, "AVATAR: a parallel corpus for Java-Python program translation," *arXiv:2108.11590v2*, 2023.

[14] M. Zhu, K. Suresh, and C. Reddy, "Multilingual code snippets training for program translation," in *AAAI*, 2022.

[15] W. Zhao, K. Zhou, J. Li, T. Tang, X. Wang, and Y. Hou, "A survey of large language models," *arXiv*, vol. 2303.18223v10, 2023.

[16] X. Li, S. Yuan, X. Gu, Y. Chen, and B. Shen, "Few-shot code translation via task-adapted prompt learning," *Journal of Systems and Software*, 2024.

[17] X. Liu, H. Yang, and H. Zedan, "Formal methods for the re-engineering of computing systems," in *Compsac '97*, 1997.

[18] H. Sneed and G. Jandrasics, "Inverse transformation of software from code to specification," in *IEEE Conf. Soft. Maintenance*, 1987.

[19] H. Siala, K. Lano, and H. Alfraihi, "Model-driven approaches for reverse engineering – a systematic literature review," *IEEE Access*, 2024.

[20] G. Deltombe, O. L. Goaer, and F. Barbier, "Bridging KDM and ASTM for model-driven software modernization," in *SEKE 2012*, 2012.

[21] R. Perez-Castillo, I. G.-R. de Guzman, and M. Piattini, "Knowledge discovery metamodel ISO/IEC 19506: A standard to modernize legacy systems," *Computer Standards and Interfaces*, vol. 33, pp. 519–532, 2011.

[22] I. Krasteva, S. Stavru, and S. Ilieva, "Agile software modernization to the service cloud," in *ICIW 2013*, pp. 1–9, 2013.

[23] K. Lano and H. Siala, "Using MDE to automate software language translation," *Automated Software Engineering*, vol. 31, 2024.

[24] K. Lano, H. Haughton, Z. Yuan, and H. Alfraihi, "Agile model-driven re-engineering," *Innovations in Systems and Software Engineering*, 2024.

[25] A. Fuhr, T. Horn, V. Riediger, and A. Winter, "Model-driven software migration into service-oriented architectures," *Comput. Sci. Res. Dev*, vol. 28, pp. 65–84, 2013.

[26] R. Fuentes-Fernandez, J. Pavon, and F. Garijo, "A model-driven process for the modernization of component-based systems," *Science of Computer Programming*, vol. 77, pp. 247–269, 2012.

[27] K. Lano, Q. Xue, and H. Haughton, "A concrete syntax transformation approach for software language processing," *Springer Nature Computer Science*, 2024.

[28] R. Jabangwe, J. Borstler, D. Smite, and C. Wohlin, "Empirical evidence on the link between object-oriented measures and external quality attributes: a systematic literature review," *Empirical Software Engineering*, vol. 20, no. 3, pp. 640–693, 2015.