TYPE Original Research
PUBLISHED 25 July 2025
DOI 10.3389/fcomp.2025.1516410

Check for updates

OPEN ACCESS

EDITED BY
Hongji Yang,
University of Leicester, United Kingdom

REVIEWED BY
Manolis Tzagarakis,
University of Patras, Greece
Kunal Rao,
NEC Laboratories America Inc, United States

*CORRESPONDENCE
Kevin Lano
✉ kevin.lano@kcl.ac.uk

RECEIVED 24 October 2024
ACCEPTED 08 July 2025
PUBLISHED 25 July 2025

CITATION
Siala H and Lano K (2025) A comparison of large language models and model-driven reverse engineering for reverse engineering. *Front. Comput. Sci.* 7:1516410.
doi: 10.3389/fcomp.2025.1516410

A comparison of large language models and model-driven reverse engineering for reverse engineering

Hanan Siala and Kevin Lano*

Informatics Department, King's College London, London, United Kingdom

Large language models (LLMs) have been extensively researched for programming-related tasks, including program summarisation, over recent years. However, the task of abstracting formal specifications from code using LLMs has been less explored. Precise program analysis approaches based on model-driven reverse engineering (MDRE) have also been researched, and in this paper we compare the results of the LLM and MDRE approaches on tasks of abstracting Python and Java programs to the OCL formal language. We also define a combined approach which achieves improved results.

KEYWORDS

program abstraction, reverse engineering, LLMS, model-driven reverse engineering (MDRE), Object Constraint Language

1 Introduction

Reverse-engineering is the extraction of design, specification or requirements level information from software applications, including abstractions or models of the application semantics, which may be expressed as textual explanations, formal specifications, or repositories of information about program elements (Bowen J. et al., 1993; Marco et al., 2018; Sneed and Jandrasics, 1987; Sneed, 2011).

Re-engineering applies forward engineering to the reverse-engineered abstractions of a source application, in order to produce a new target version, in a different programming language or on a modernized platform (Bowen J. P. et al., 1993), and may include design refactorings or design transformations for quality improvements (Lano and Malik, 1999).

In this paper we focus upon reverse engineering to derive formal specifications from programs, in particular, our goal is to produce specifications in the OCL formal language used within the UML (OMG, 2014). We evaluate and compare model-based and machine learning (ML) approaches for this reverse engineering task, in terms of their semantic quality and in terms of their utility as part of a re-engineering process. The outcome is a detailed analysis of the relative strengths and weaknesses of the approaches for the extraction of OCL specifications from Java and Python code.

Model-driven reverse engineering (MDRE) (Siala et al., 2024), is a reverse-engineering approach which extracts software models from programs, including visual models such as UML class diagrams and state machines. The Object Management Group (OMG) has defined standard modeling representations for MDRE as part of its Architecture-driven Modernization (ADM) re-engineering approach (Krasteva et al., 2013; Perez-Castillo et al., 2010, 2011). These include the Knowledge Discovery Metamodel (KDM) for representing source code elements and their inter-relationships, and the Abstract Syntax Tree Metamodel (ASTM) for source code representation as parse trees. The intention of the ADM process is that language-specific parsers produce ASTM models, which are then abstracted to KDM models by reverse-engineering model transformations between the respective models.

However, the use of ADM implies the establishment of a full model-driven engineering (MDE) process with supporting tools, requiring reverse-engineers to have MDE skills and knowledge of the specialized metamodels (KDM and ASTM) involved, and thus to date most of the use of ADM has been in research projects such as REMICS and MoDisco. An alternative to ADM is a lightweight MDRE process, such as AMDRE (Lano and Siala, 2024a; Lano et al., 2024), which performs an abstraction transformation from program code to UML and OCL specifications using pattern matching on abstract syntax trees (ASTs), without the need for metamodelling knowledge.

Machine learning (ML) approaches have been applied extensively for the task of *program translation*: translating a program in one programming language into a semantically equivalent version in another programming language. This can be regarded as a specialized form of re-engineering, however in such approaches the intermediate abstraction is usually not explicitly created or accessible to users. To date, there have been few reverse engineering approaches using ML for the production of a formal specification from program code (Siala et al., 2024). Although general code-aware LLMs such as GPT-4 and Mistral can be directly used for the task of extracting OCL specifications from code, they have limitations, such as imperfect knowledge of OCL, hallucinations (production of plausible but incorrect output) and unreliability (Ouyang et al., 2023; Zhao et al., 2023). In our initial work on the baseline capabilities of LLMs, we found that they also tend to produce abstract and implicit OCL specifications, which are not suitable for use in forward engineering without manual refinement (Siala and Lano, 2025).

To improve the quality and utility of OCL production by LLMs used for reverse engineering, we propose the use of LLM *fine-tuning* to adapt a general-purpose LLM to the reverse engineering task (Siala, 2024). The resulting fine-tuned model is termed LLM4Models. We apply this LLM for the reverse-engineering of OCL specifications from Java (versions 5 to 8) and Python (versions 2.7 and 3.*) source codes.

In Section 2 we define the MDRE and LLM-based techniques which we use for reverse engineering, and we identify the evaluation criteria used to compare them. The criteria evaluate both the semantic accuracy of the extracted OCL specifications, with respect to the source programs, and the utility of these specifications for forward engineering.

In Section 3 we compare the selected MDRE approach with general purpose and fine-tuned LLMs for the task of extracting OCL specifications from code.

In Section 4 we survey related work, and consider how the work presented here may be extended and how the specification extraction process can be further enhanced.

2 Material and methods

In this section we describe the specific reverse-engineering technologies used in our work, how they are applied, and the critera used to compare them.

2.1 LLMs for reverse engineering

Large Language Models (LLMs) are a specific type of machine learning (ML) technology that can learn a variety of knowledge representations from large and complex datasets (Zhao et al., 2023). Once trained, the LLM can be utilized to accomplish specific downstream tasks by using additional specialized training (*fine-tuning*) on demonstration examples of the specialized task. Widely known examples of LLMs include GPT3&4, Bard, LLaMA, Mistral, BERT, and T5. LLMs have had a major impact on many fields, including software engineering, where researchers have explored using them for a variety of tasks (Hou et al., 2023).

Code-aware LLMs such as Mistral and GPT4 can be used directly for reverse-engineering of Java and Python to OCL, however the results are often unsatisfactory for use in re-engineering, because the produced specifications contain errors in OCL usage, are too abstract and implicit, or make simplifying assumptions that are not present in the code (Siala and Lano, 2025). For example, a search algorithm that searches a specified portion of a list may be abstracted to a simplified version that only searches the entire list.

Thus we considered improvement of a pre-trained LLM by fine-tuning using datasets of (i) Python programs and corresponding OCL specifications; (ii) Java programs and corresponding OCL specifications.

LLMs come in different sizes, where larger models may be more accurate, but also require more processing power, memory, training times, and other resources. GPT4 is the most widely-used current LLM for SE tasks, and we therefore selected this as the example untrained general LLM for reverse engineering. However, fine-tuning of GPT4 requires significant resources. Therefore we selected **Mistral-7B-v0.3**[1] as the baseline LLM for fine-tuning because of limited financial and computational resources. In addition, the performance of Mistral-7B is better than Llama 2 13B (Jiang et al., 2023), which was the other main LLM we considered for fine-tuning.

We considered three main ways to fine-tune an LLM for the reverse engineering task using supervised fine-tuning (SFT), which are: (i) Full fine-tuning, (ii) LoRA (Hu et al., 2022), and (iii) QLoRA (Dettmers et al., 2024). Full fine-tuning is expensive, because it involves possible modification of all the weights in the pre-trained LLM by the re-training. In LoRA, some extra weights (adapters) in some layers are added and trained while freezing most of the parameters of the selected LLM to reduce the cost of training the original weights and to prevent *catastrophic forgetting* that may occur during full fine-tuning (Zhao et al., 2023). QLoRA is a quantized version of LoRA, which optimizes LoRA to reduce the resources used. We selected QLoRA as our approach to fine-tune the LLM.

Source Java and Python programs for the training dataset were collected from sources such as the CoTran (Jana et al., 2023) and AVATAR datasets.[2] The sources were pre-processed to put them in a consistent format. AgileUML was then applied to produce OCL specifications for each source program. The pairs of Java code and

1 https://huggingface.co/mistralai/Mistral-7B-v0.3

2 https://github.com/wasiahmad/AVATAR

corresponding OCL specifications formed the dataset for LLM fine-tuning for Java reverse engineering, and the pairs of Python code and corresponding OCL specifications formed the dataset for LLM fine-tuning for Python reverse-engineering.

Following standard practice in ML, each fine-tuning dataset was divided into three sets:

1. Training examples, ~80% of the total.
2. Validation examples, ~10% of the total.
3. Test examples, also 10% of the total.

For Java, there were 20,580 samples, divided into: training data = 16,670 cases; validation data = 1,852 cases; test data = 2,058 cases.

For Python, there were 25,684 samples, divided into: training data = 20,805 cases; validation data = 2,311 cases; test data = 2,568 cases.

The training set is directly used to fine-tune the model. The validation set is used to monitor and optimize the model performance during training for each epoch or for a specified number of iterations using metrics such as loss and accuracy. At this stage, changes may be applied to the model's hyperparameters or to the training process to improve performance. The test set is used to evaluate the LLM performance on unseen data.

The training was distributed over four GPUs where PyTorch FSDP and Q-LoRA were utilized. The resulting fine-tuned LLM is referred to as LLM4Models.

The LLM prompt used to abstract Java program code to OCL is:

```
"""<s>[INST] Below is an instruction that describes
a task,
paired with an input that provides further context.
Write a response that appropriately solves the
following Task:

### Instruction:
Generate an Object Constraint Language (OCL)
 specification for
the provided Java code. The output should:
1. Ensure no repeated or redundant operations or
classes.
2. Include only the OCL code for the provided
Java code.
3. Do not include statements for items not found in
the Java code.

### Input:
... source code ...
[/INST]

### Response: """"
```

The same prompt, with "Python" instead of "Java," is used for Python abstraction. These prompts were determined through a systematic process of prompt engineering to identify the most appropriate instructions to obtain accurate and high-quality results.

2.2 AgileUML

AgileUML is an Eclipse MDE toolset[3] that supports the agile methodology and MDRE. It provides visual and textual editing of specifications using UML class diagrams and OCL constraints. These specifications can be analyzed and forward engineered to generate code in Java, Python, Swift, C++, C#, Go, and ANSI C.

For MDRE, the AgileUML toolset uses a set of manually-coded pattern-matching rules in the *CSTL* notation to abstract UML class diagrams and OCL constraints from the abstract syntax trees of Java, Python, C, JavaScript, VB, COBOL, and Pascal programs (Lano and Siala, 2024a). Also, it defines various supporting OCL library components not found in standard OCL, in order to represent programming language facilities, such as dates, times, iterators, files, and exceptions. The OCL extensions and libraries supported by the AgileUML toolset are detailed in Lano et al. (2022) and Lano and Siala (2024b). The resulting specifications are expressed in an explicit style of OCL, with computational steps defined using a procedural extension of OCL, similar to the SOIL formalism (Buttner and Gogolla, 2014).

We selected AgileUML as the example MDRE approach because it is one of the most accurate and versatile MDRE tools, with superior performance compared to other approaches for program translation and other forms of re-engineering (Lano and Siala, 2024a; Lano et al., 2024).

In our work, AgileUML is utilized firstly to provide a baseline MDRE approach to compare with untrained and fine-tuned LLMs for the Java and Python reverse engineering tasks. Secondly, it is used to create training datasets for LLM fine-tuning for reverse engineering. The datasets are produced by applying the AgileUML reverse engineering process to the selected Java and Python programs. This provides two datasets: (i) of pairs of Java programs and their corresponding OCL specifications; (ii) of pairs of Python programs and corresponding OCL specifications.

These training datasets are then used to fine-tune a pre-trained LLM, in our case Mistral, in order that the LLM (i) can create specifications in the same explicit style as AgileUML, but for a wider range of input programs, and (ii) so that the LLM can use the AgileUML extended OCL and OCL libraries to produce simpler and more concise specifications than those expressed in standard OCL. In this work, we opted to use the extended OCL introduced by AgileUML in our approach because this provides many aspects needed to represent program semantics that are missing from standard OCL. For example, the OCL standard provides `Integer` and `Real` to represent mathematical numeric datatypes, but AgileUML provides `int`, `long`, and `double` as basic types, which are more aligned with program datatypes. Likewise, an extensive set of file operations are provided by the *OclFile* library, which are not provided by standard OCL.

3 https://projects.eclipse.org/projects/modeling.agileuml (Accessed August 1, 2024).

2.3 Object Constraint Language

The following are the elements of standard OCL (OMG, 2014) specifications:

1. Data types: Different types of data are included in OCL constraints, including:

    - Primitive types: `Integer`, `Real`, `String`, enumeration types and `Boolean`.
    - Collection types: `Bag`, `Set`, `Sequence`, and `OrderedSet`.
    - User-defined types: User-defined types represent classes and their attributes. For example:
      context Student

    These types correspond in general to types in Java and Python. For example, array or list types in Java and list types in Python can be represented using OCL `Sequence` types.
2. Operators and operations: OCL operations include:

    - Standard operators: Standard operators for numbers, strings, Booleans, and collections include +, -, *, / on numbers, *substring* and *concat* on strings, *and*, *or*, *not* on Booleans, and $\rightarrow exists()$, $\rightarrow forAll()$, and $\rightarrow size()$ on collections.
    - User-defined operations: User-defined operations of classes can be represented as, for example:
      context Person::walk()

    These elements also correspond to Python and Java elements. For example, `any Col` in Python corresponds to $Col \rightarrow exists(x \mid x = true)$ in OCL.
3. Expressions: Java and Python expressions can in principle be abstracted to OCL expressions. Program constraints and conditions may be expressed as arithmetic expressions (e.g., $self.balance + amount$), Boolean expressions (e.g., $self.age \geq 18$), or collection expressions [e.g., $self.retiredPeople \rightarrow forAll(c \mid c.age >= 65)$].

Standard OCL was found to be inadequate for use to express the semantics of program code, and various extensions were added in AgileUML to support MDRE (Lano and Siala, 2024b):

1. Map and function types, together with operators such as lambda abstraction: `lambda x : T in expr` and function application `f->apply(x)`.
2. Statements: procedural OCL statements based on those of SOIL (Buttner and Gogolla, 2014) (Table 1).
3. OCL extension libraries *OclFile*, *OclRandom*, *OclProcess*, *MathLib*, *StringLib*, *MatrixLib*, and others (Lano et al., 2022).

2.4 Program abstraction rules

The following rules (Tables 2, 3) are encoded in the *CSTL* scripts used by AgileUML for the MDRE of Python and Java programs to UML/OCL. They are also used to generate the datasets to fine-tune Mistral to produce the LLM4Models LLM.

TABLE 1 AgileUML structured activities: procedural OCL statements.

Statement	Meaning
$x := e$	Assignment
var $x : t$	Variable declaration
$s1$; $s2$	Sequencing
if e then $s1$ else $s2$	Conditional
while e do s	Unbounded loop
repeat s until e	Unbounded loop
for *ident* : e do s	Bounded loop
return	Return statement
return e	Return value statement
(s)	Statement group
break	Break statement
continue	Continue statement
e	Operation call, e is *obj.op*(*pars*), *ClassName.op*(*pars*) or *op*(*pars*)
skip	No-op

2.5 Correctness and quality measures

We evaluate the syntactic correctness of the results produced by each reverse-engineering approach by attempting to parse them using the ANTLR OCL parser.[4]

For semantic correctness, we evaluate firstly the *completeness* of each reverse engineering approach, where completeness is defined as the proportion of source code elements that are correctly represented as UML/OCL elements in the generated specifications. This is the ratio

$$\frac{TP}{TP + FN}$$

where a true positive (TP) is an element correctly translated from code to OCL, and a false negative (FN) is an element that is not translated, or is translated incorrectly. Completeness in this sense is also known as *recall*.

Secondly we evaluate the *consistency* of each approach, where this is the proportion of elements in the generated specifications which are correctly derived from elements in the source programs. Consistency is important for ensuring traceability and alignment of the derived specification with respect to the source.

Consistency is also referred to as *precision*, the ratio

$$\frac{TP}{TP + FP}$$

where a false positive (FP) is an element that appears in the specification that is not correctly derived from a source code element.

Completeness measures the extent to which the abstraction process correctly processes source elements, whilst consistency

4 https://github.com/antlr/grammars-v4

TABLE 2 Abstraction rules from Python to UML/OCL.

Python element	UML/OCL
Program	Package and main class
Class	Class
Inheritance	Inheritance
Global variable	Attribute of main class
Instance-scope attribute	Non-static attribute
Class-scope attribute	Static attribute
Local variable	Local variable
Function	Operation
Assignment	Assignment/declaration
Sequencing, blocks	Sequencing, blocks
Function calls	Operation calls
Object creation $x = C()$	Call $x := C.newC()$
If, for, while	if, for, while
Continue, break, pass	continue, break, skip
Raise, try, except	Error, try, catch
Assert	Assert
With	Try/catch
Int/float	Int/double
Str/bool	String/Boolean
List, tuple types	Sequence types
Numpy matrix	Nested sequences
Dict type	Map type
Set type	Set type
Files	OclFile

TABLE 3 Abstraction rules from Java to UML/OCL.

Java element	UML/OCL
Program	Specification
Package	Package
Class, abstract class	Class, abstract class
Interface	Interface
Extends, implements	Extends, implements
Nested classes	Associated classes
Generic classes	Generic classes
Field, static field	Attribute, static attribute
Method, static method	Operation, static operation
Generic method	Generic operation
Constructor C(pars)	Static operation newC(pars)
This, super	Self, super
Method overriding, overloading	Operation overriding, overloading
Assignment, sequencing, blocks	Assignment, sequencing, blocks
If, while, do, return	If, while, repeat, return
Expression statements	Operation calls/assignments
General for loop	For, while
Enhanced for loop	For
Switch	If
Simple continue, break	Continue, break
Int, short, byte	Int
Long	long
Double, float	Double
String, char, StringBuffer	String
Object, Class	OclAny, OclType
Simple enum definitions	Enumerated types
Classlike enum definitions	Classes
Array types, Vector, List, etc	Sequence types
Set, HashSet, TreeSet, etc	Set types
Map, HashMap, TreeMap, etc	Map types
Function, Predicate	Function types

measures the quality of the generated specification in terms of absence of spurious elements not derived from the source.

A combined measure is the $F1$ metric $\frac{2*recall*precision}{recall+precision}$.

The specifications produced by the reverse engineering of program code can also be expressed in different styles, and at different levels of formality. We distinguish several cases for UML/OCL specifications:

1. Informal—Specifications expressed using natural language.
2. Semi-formal—Specifications partly expressed using natural language and partly with formal OCL constraints.
3. Fully formal Specifications, which can be subdivided into:
   (a) Implicit—Formal specifications expressed at a high level of abstraction, specifying properties of a computational result, but not the process of constructing the result.
   (b) Explicit—Formal specifications which define explicitly how results are computed, using OCL and procedural OCL.

To facilitate re-engineering, fully formal and explicit specification forms are preferred, although for other purposes, such as code explanation or summarisation, implicit, semiformal or informal forms may be more appropriate.

3 Results

We evaluated and compared the AgileUML, untrained LLM and LLM4Models approaches for program abstraction on 20 randomly-selected Python and Java examples from the test sets of the training datasets. We evaluated the syntactic correctness, completeness and consistency of the approaches (Tables 4, 5). We also evaluated the quality of the produced specifications, in terms of their level of formality and explicitness. The percentage of explicit specifications is shown in the final columns of Tables 4, 5. All other produced specifications were implicit in style.

The AgileUML Java abstractions were produced by using the ANTLR version 4 command line

```
grun Java compilationUnit -tree
```

to produce an abstract syntax tree of a source Java program, and then by applying the Java to UML CSTL abstraction script via the command

```
cgtl cg/cgJava2UML.cstl output/ast.txt
```

where *ast.txt* contains the abstract syntax tree from the first step. Likewise, for Python, the commands

```
grun Python file_input -tree
```

and

```
cgtl cg/python2UML.cstl output/ast.txt
```

are used to obtain a UML/OCL specification from Python source code.

All examples, training data and analysis results can be obtained from the repository (https://zenodo.org/records/14988622).

Tables 4, 5 show that the LLM4Models LLM has substantially improved syntax correctness, completeness, consistency and F1 measures compared to the untrained LLM, for both Java and Python abstraction. In addition, the completeness improves on that of AgileUML for Java abstraction. The specifications produced by LLM4Models are all explicit in style, which represents a significant improvement over the untrained LLM.

Figures 1, 2 give a detailed comparison of the three reverse-engineering approaches for Python.

It can be seen that for 13 of 14 Python cases, the LLM4Models completeness and consistency are lower or equal to that obtained by AgileUML. However, for 12 of 14 cases, the LLM4Models completeness and consistency is at least as high as that obtained by the untrained LLM.

Figures 3, 4 give a detailed comparison of the three reverse-engineering approaches for Java.

It can be seen that for all Java cases, the LLM4Models completeness is at least as high as that obtained by AgileUML, whilst the LLM4Models consistency is always lower or equal to that obtained by AgileUML. For each Java case, the AgileUML and LLM4Models completeness and consistency are always greater than or equal to the values obtained using the untrained LLM.

An example of Java abstraction is given by the following CoTran dataset example (Java evaluation case number 3):

```
public class EQUILIBRIUM_INDEX_OF_AN_ARRAY_1 {
    static int M209(int arr[], int n) {
        int sum = 0;
        int leftsum = 0;
        for (int i = 0; i < n; ++i)
          sum += arr[i];
        for (int i = 0; i < n; ++i) {
            sum -= arr[i];
            if (leftsum == sum)
               return i;
            leftsum += arr[i];
        }
        return -1;
    }
}
```

AgileUML produces the following explicit specification in UML and procedural OCL:

```
class EQUILIBRIUM_INDEX_OF_AN_ARRAY_1 {
 static operation M209( arr : Sequence(int), n : int)
   : int
  pre: true
  post: true
  activity: (
    var sum : int ; sum := 0 ;
    var leftsum : int ; leftsum := 0 ;
    for i : Integer.subrange(0, (n)-1) do (
        sum := sum+(arr[i+1])
      ) ;
    for i : Integer.subrange(0, (n)-1) do (
      ( sum := sum-(arr[i+1]) ;
        if (leftsum = sum) then
          return i else skip ;
        leftsum := leftsum+(arr[i+1]) )
      ) ;
    return -1
 );
}
```

An almost identical specification (using *while* loops instead of *for* loops) is produced by LLM4Models. These specifications are explicit and semantically correct.

However, the untrained LLM (GPT4) produces an implicit-style specification with semantic errors:

```
context EQUILIBRIUM_INDEX_OF_AN_ARRAY_1::M209
(arr: Sequence(Integer),
                 n: Integer) : Integer
pre: n > 0 -- The array must have at least one element
post: if result <> -1 then
  -- If an equilibrium index is found, the sum of
   elements to the left
  -- of index i should equal the sum of elements
  to the right
         arr->subSequence(1, result - 1)->sum() =
            arr->subSequence(result + 1, n)->sum()
      else
 -- If no equilibrium index is found, the result is -1
         true
      endif
```

Here, the value of *result* is only expressed implicitly in the postcondition. Moreover, a precondition that $n \leq arr{\rightarrow}size()$ is needed in order that this specification has the same functionality as the source code.

For more complex cases, a simple implicit specification is not possible. For example, consider the following CoTran Java case (number 6 in the Java evaluation examples):

```
public class TOTAL_NUMBER_OF_NON_DECREASING_
NUMBERS_WITH_N_DIGITS_1{
static long aaa(int n){
  int N=10;
  long count=1;
  for(int i=1;i<=n;i++){
    count=(int)count*(N+i-1);
    count=(int)count/i;
  }
  return count;
}
}
```

The untrained LLM produces a procedural version using the OCL $\rightarrow$*iterate* operator:

TABLE 4 Evaluation of approaches for Python abstraction.

Approach	Syntax	Completeness	Consistency	F1	Explicit
AgileUML	86%	98%	97%	0.975	100%
Untrained LLM	36%	60%	68%	0.64	64%
LLM4Models	79%	88%	85%	0.865	100%

TABLE 5 Evaluation of approaches for Java abstraction.

Approach	Syntax	Completeness	Consistency	F1	Explicit
AgileUML	83%	87%	99%	0.93	100%
Untrained LLM	33%	65%	71%	0.68	83%
LLM4Models	83%	96%	96%	0.96	100%

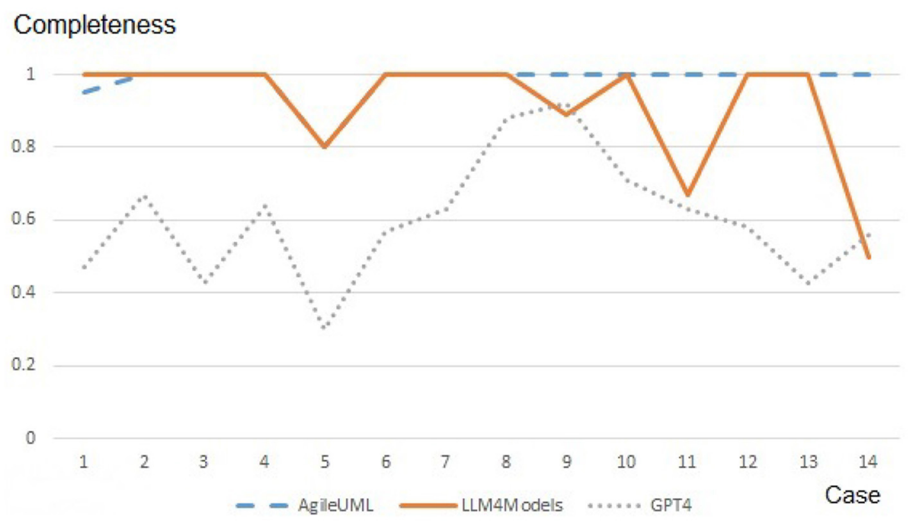

FIGURE 1
Completeness results for Python examples, numbered 1 to 14.

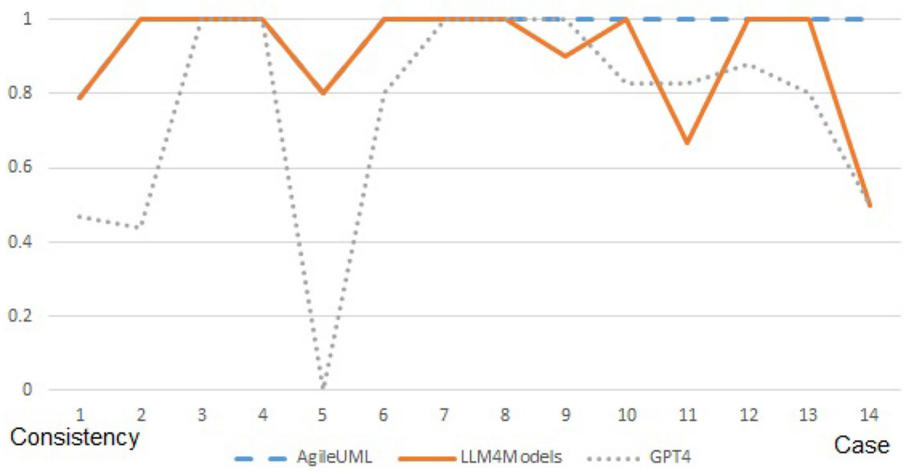

FIGURE 2
Consistency results for Python examples, numbered 1 to 14.

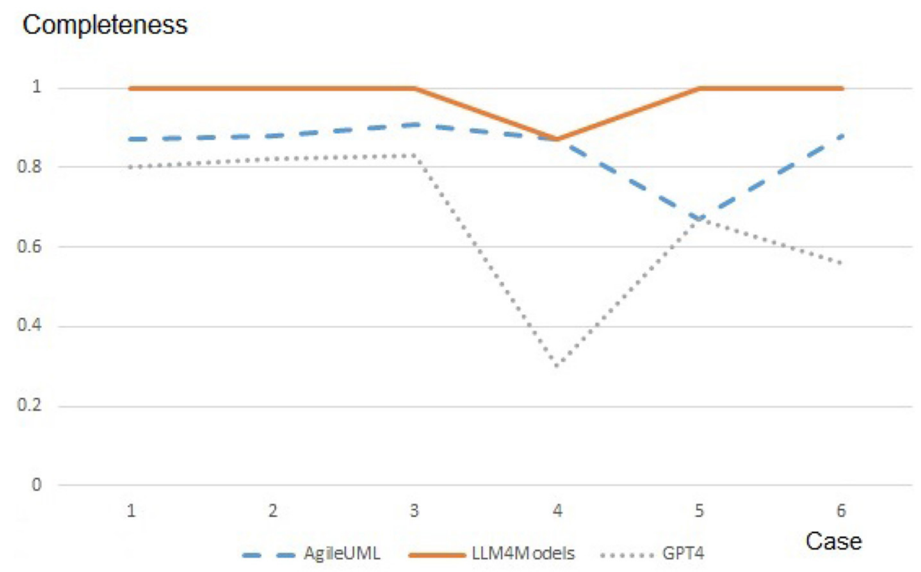

FIGURE 3
Completeness results for Java examples, numbered 1 to 6.

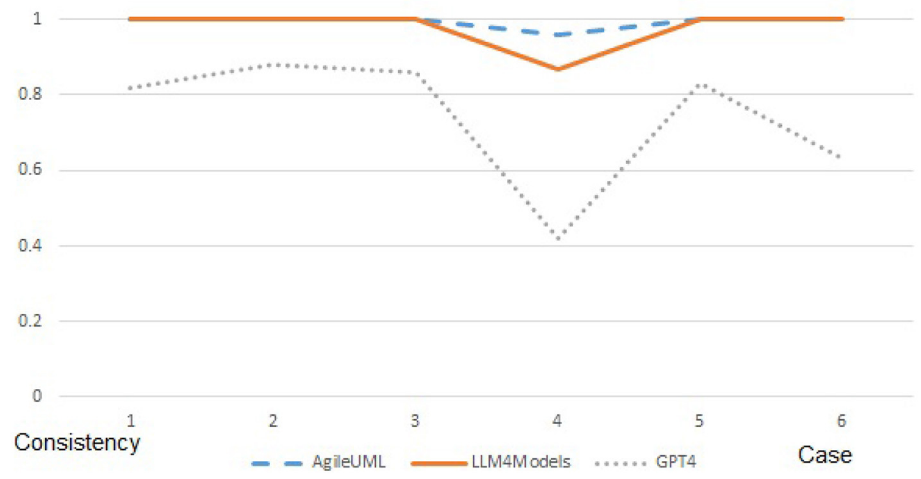

FIGURE 4
Consistency results for Java examples, numbered 1 to 6.

```
-- Context: TOTAL_NUMBER_OF_NON_DECREASING
_NUMBERS_WITH_N_DIGITS_1 class
context TOTAL_NUMBER_OF_NON_DECREASING_NUMBERS_
WITH_N_DIGITS_1

-- Operation: aaa(n: Integer): Long
inv:
  let N: Integer = 10 in
  let count: Long = 1 in
  count = self.aaa(n)

-- The operation specification in OCL
context TOTAL_NUMBER_OF_NON_DECREASING_
NUMBERS_WITH_N_DIGITS_1::
  aaa(n: Integer): Long
pre: n > 0 -- The number of digits (n) must
 be positive
post:
  result = (Sequence{1..n}->iterate(i; count:
  Long = 1 |
            (count * (N + i - 1)) / i)) -- Iterative
              calculation of count
```

Apart from the use of *iterate*, another issue with this specification is that the *let* statements are ineffective and redundant: *count* is initialized to 1 in the *iterate* loop, and *N* should be passed as a parameter of the *aaa* operation.

In contrast to the GPT-4 solution, the AgileUML and LLM4Models tools produce program abstractions which use *for* or *while* loop constructs, which are more familiar to software practitioners in general than *iterate*:

```
class TOTAL_NUMBER_OF_NON_DECREASING_NUMBERS_
WITH_N_DIGITS{
  static operation aaa( n : int) : long
  pre: true
  post: true
  activity: (
     var N : int ; N := 10 ;
     var count : long ; count := 1 ;
     for i : Integer.subrange(1, n) do (
     ( count := count->oclAsType(int) * (N + i - 1) ;
         count := count->oclAsType(int) / i )
     ) ;
     return count
  );
}
```

in the case of AgileUML, and:

```
class TOTAL_NUMBER_OF_NON_DECREASING_NUMBERS_
WITH_N_DIGITS_1 {
  static operation aaa( n : int) : long
  pre: true
  post: true
  activity: (
    var N : int ; N := 10 ;
    var count : long ; count := 1 ;
    var i : int ; i := 1 ;
    while i <= n do (
     count := count->oclAsType(int) * (N + i - 1) ;
      count := count->oclAsType(int) / i ;
      i := i + 1 ) ;
    return count
  );
}
```

in the case of LLM4Models.

Similar issues arise with Python program abstraction. For simple cases, the untrained LLM can produce accurate and useful abstractions, but more complex cases lead to errors or poorly-aligned specifications.

One example is the Python program:

```
i = 0
while i < 100 :
  i = i + 1
  print(i*i)
```

This is the Python evaluation case 13.

AgileUML and LLM4Models produce the output:

```
 class FromPython {

  operation initialise()
  pre: true post: true
  activity:
    var i : int := 0 ;
    while i < 100 do (
      i := i + 1 ;
      execute (i * i)->display());
}
```

This is a correct semantic representation of the source code, and can directly be used for forward engineering.

However the untrained LLM produces a more declarative and implicit specification of the same program:

```
-- Context: Describing the behavior of the loop
 and its constraints
context LoopOperation

-- Initial condition: i starts at 0
inv Initialization:
  i = 0

-- Loop invariant: While i is less than 100, the
 loop continues
inv LoopCondition:
  i >= 0 and i <= 100

-- Postcondition: After each iteration, i is
 incremented by 1
post Increment:
  i = i@pre + 1

-- Iteration constraint: For each iteration,
-- print the square of the current i
inv Iteration:
  i >= 0 and i <= 100 implies print(i * i)

-- Final state: Once i reaches 100, the
 loop terminates
post FinalState:
  i = 100
```

This describes the computation, rather than explicitly expressing its semantics, and is not in a form suitable for forward engineering.

AgileUML and LLM4Models tend to make similar mistakes to each other, for example, both fail to deduce the types of Python variables in many cases. However, LLM4Models can be more creative than AgileUML, for example, introducing new OCL operators (such as $\rightarrow$*capitalize* for strings or $\rightarrow$*mean* for collections) which are needed to represent the program semantics of examples that have not been seen in the LLM training data.

4 Discussion

In this section we give a detailed comparison of the reverse-engineering methods, survey related work, relate our work to the state of the art, consider threats to validity, summarize our contributions, and consider future areas for further development and extension of our research.

4.1 Detailed comparison of methods

The three reverse engineering approaches have different strengths and weaknesses for reverse engineering. Table 6 gives the frequencies of different types of errors that arise in reverse engineering of the Python test cases to OCL.

The principal cause of incompleteness in AgileUML and the trained LLM is the difficulty of deriving types for untyped variables in Python. The untrained LLM also omits other forms of information, particularly procedural information about the sequencing of processing. The untrained LLM is also more prone to introduce spurious elements, particularly additional invariants or constraints which are more restrictive than those in the source code. For example, introducing a spurious constraint that an integer-typed variable is non-negative. Overall, the untrained LLM

TABLE 6 Flaws in Python reverse-engineering.

Approach	Incompleteness	Hallucination	Spurious elements
AgileUML	0.14	0	0.14
Untrained LLM	0.36	0.14	0.43
LLM4Models	0.14	0.07	0.28

TABLE 7 Flaws in Java reverse-engineering.

Approach	Incompleteness	Hallucination	Spurious elements
AgileUML	0.33	0	0
Untrained LLM	0.33	0	0.67
LLM4Models	0.17	0	0

produces results which are further removed in structure and content from the source, compared to the other two approaches, which may impair traceability. However, in some cases the untrained LLM can infer higher-level abstractions, for example, it produces the declarative result

```
context A : Integer
inv: let B : Integer =
      if A > 10 then
          A * A
      else if A > 0 then
          A * A * A
      else
          -A
      endif
      endif
```

for the Python program

```
A = 11
if A > 10 :
  B = A*A
elif A > 0 :
  B = A*A*A
else :
  B = -A
print(B)
```

In contrast the other two approaches produce a procedural version that closely imitates the source:

```
operation initialise()
pre: true post: true
activity:
  var A : int := 11 ;
  if A > 10 then (
    var B : int := A * A
  )
  else (if A > 0 then
  (
    B := A * A * A
  )
  else (
    B := -A
   ) );
  execute (B)->display();
}
```

This version needs further post-processing in order to conform to OCL semantics (the declaration of *B* needs to be moved to the outer scope of the operation activity).

The situation for the abstraction of Java to OCL is similar. Table 7 lists the types of errors that arise in the different abstraction methods when applied to Java programs.

Again, the main problem with the untrained LLM is the production of spurious invariants, preconditions or postconditions. It is difficult to automatically detect such errors, and hence correction of the output would need to involve human expertise.

4.2 Related work

In this section we review related work in MDRE and ML-based program translation and code abstraction approaches.

4.2.1 Model-driven reverse engineering

Early work on MDRE considered the extraction of formal specifications from code, either represented in state-based formal languages such as Z, or in other formalisms such as algebraic specification languages (Bowen J. et al., 1993; Liu et al., 1997).

MDRE was combined with model-driven forward engineering to support *model-driven modernization* (MDM) (Bowen J. P. et al., 1993; Lano and Malik, 1999). MDM was standardized by the Object Management Group (OMG) in their Architecture-driven modernization (ADM) framework (Perez-Castillo et al., 2011). ADM supports in principle a "horseshoe modernization model" whereby the starting point is the legacy system source artifacts at a low level of abstraction, these are then reverse-engineered to higher levels of abstraction, e.g., Platform-specific, Platform-independent and Computation independent models (PSMs, PIMs, CIMs), and then forward-engineered to a target platform (Deltombe et al., 2012; Perez-Castillo et al., 2011). MDM in general enables multi-lingual translation between multiple programming languages, with some degree of assurance of semantic preservation (Krasteva et al., 2013; Lano and Siala, 2024a; Lano et al., 2024). However, as with other forms of model-driven engineering, MDM and MDRE requires the availability of practitioners with appropriate MDE knowledge and skills, including metamodelling and model transformation development. Thus, despite the potential of MDRE and MDM for supporting highly rigorous re-engineering, there has been limited use of MDRE and MDM in practice. Instead, customized re-engineering processes have been used, specifically created for each re-engineering project (Marco et al., 2018; Sneed, 2011).

4.2.2 ML approaches for program translation and abstraction

Based on the successful use of ML for neural machine translation (NMT) of natural languages, similar NMT approaches were applied to the translation of software languages (Nguyen et al., 2015; Chen et al., 2018). 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 modeling training to program translation in Transcoder (Lachaux et al., 2020). This enabled the use of large monolingual code datasets to train an ML model with knowledge of the individual programming languages together with knowledge of language correspondences. Specifically, masked language modeling and denoising auto-encoding learning objectives were used together with coupled target-to-source and source-to-target training. The result was a pre-trained language model (PLM) with significantly higher translation performance than manually-coded program translators. Nevertheless, such an approach has limitations: it utilizes common syntactic anchor points between programming languages, such as similar keywords *if*, *while*, etc., in different languages, and 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 Malyaya et al. (2023).

Subsequent PLM approaches to program translation include GraphCodeBERT (Guo et al., 2021), the Transcoder-IR approach of Szafraniec et al. (2023), the CoTran approach of Jana et al. (2023), and approaches utilizing fine-tuning of PLMs (Ahmad et al., 2023; Zhu et al., 2022). GraphCodeBERT uses data flow information to enable a PLM to learn more semantically-meaningful programming knowledge during pre-training. In Szafraniec et al. (2023), pre-training is enriched by the use of compiler intermediate representations, to increase the semantic awareness of the trained model. CoTran (Jana et al., 2023) incorporates semantic equivalence into the training objective by utilizing 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 Ahmad et al. (2023). A similar fine-tuning approach was applied with a dataset of 7 programming languages in Zhu et al. (2022). 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 (Zhao et al., 2023)) brought a further change to the program translation landscape. Although LLMs are typically pre-trained on large general purpose datasets, and with simple language modeling objectives such as next token prediction, they have proven to be capable of carrying out diverse tasks as well as, or better, than specialized smaller-scale PLMs (Zhao et al., 2023). 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 or refinement (Li et al., 2024; Gandhi et al., 2024).

ML approaches can also be used to derive specification-level documentation from programs by automated *code summarisation*, whereby natural language explanations are generated from program code (Zhang et al., 2022). Code summarisation techniques have mainly focussed upon modern programming languages, and there is a lack of datasets and tools for COBOL and VB (Gandhi et al., 2024). The approach of Boronat and Mustafa (2025) derives UML class diagram models from Java code using Retrieval-Augmented Generation (RAG) techniques to reduce LLM errors. This approach does not generate explicit semantic representations of the code, so it is mainly applicable to enhance code comprehension at a structural level, rather than as a basis for code migration.

4.3 Threats to validity

Threats to validity include bias in the construction of the evaluation, inability to generalize the results, inappropriate constructs and inappropriate measures.

4.3.1 Instrumental bias

This concerns the consistency of measures over the course of the analysis. To ensure consistency, all analysis and measurement of the results was carried out in the same manner by a single individual (the second author) on all cases. Analysis and measurement for the results of Tables 4, 5 were repeated in order to ensure the consistency of the results.

4.3.2 Selection bias

For the training datasets we chose Java and Python examples which covered the core statements and features of the languages. The source codes were taken from established datasets such as AVATAR (Ahmad et al., 2023) and CoTran (Jana et al., 2023) which have been used in other research. In addition we created our own examples to ensure completeness of grammar coverage. Care was taken to ensure that there was no duplication of examples within the training datasets.

4.3.3 Generalization to different samples

We found a high degree of consistency of our results across different examples, as shown in Figures 1–4. Since these were randomly-chosen examples, this consistency in the relative performance of the approaches also suggests that the same relations would hold between the approaches when applied to other random subsets of the test data.

4.3.4 Inexact characterization of constructs

We have used established metrics such as precision, recall and F1 measure in our evaluation, in order to measure the completeness, consistency and quality of the reverse-engineering process. These metrics however involve a subjective aspect (the human evaluator has to determine if program elements have been translated correctly or not). Alternative and more objective measures, such as equivalent behavior on a set of test cases, could also be considered for future work, however there is currently no means to directly execute AgileUML OCL specifications. Thus an execution-based measure such as computational accuracy (Lachaux et al., 2020) or runtime equivalence (Jana et al., 2023) could not be used.

4.3.5 Relevance

The LLM4Models reverse engineering approach has been shown to be applicable to the analysis of Java (Java versions from 5 to 8) and Python (versions 2.7 and 3.*). In principle, reverse engineering of other versions of these languages, or other programming languages, could be addressed by using the same model training procedures.

4.3.6 Representativeness

The Java and Python examples were selected to be representative of real-world Java and Python coding, rather than artificial examples constructed by the authors. The datasets used included programmer solutions from websites of programming tasks/solutions, such as AtCoder, used by AVATAR. The examples considered in Section 3, such as `EQUILIBRIUM_INDEX_OF_AN_ARRAY_1`, are of this kind.

4.3.7 Threats to conclusion validity

The consistency of our comparison results across different examples, as shown in Figures 1–4 demonstrates that there are consistent differences in the results of the three different reverse-engineering approaches.

4.4 Relation to the state of the art

AgileUML is one of the leading MDE approaches for program translation in terms of semantic accuracy, and it also exhibits higher translation accuracy than ML-based approaches such as CoTran (Lano et al., 2024). This indicates that AgileUML also has high semantic accuracy as a MDRE approach abstracting from code to UML/OCL. Our evaluation results in Section 3 support this claim, and also show that the AgileUML abstraction rules have been successfully transferred to the Mistral LLM via fine-tuning, to produce an LLM, LLM4Models, with similar abstraction quality to AgileUML, and substantially improving on the quality of an untrained LM.

Although the untrained LLM is able to produce formal specifications for simple source programs, we found that it had significant deficiencies for more general use in program abstraction:

- Creation of implicit-style specifications which are not well-suited for forward engineering.
- Incorrect use of OCL types and syntax. For example, introducing a type *Long*, which is not in the OCL standard, and using the notation

  ```
  x^p
  ```

  for exponentiation, which is also not in the OCL standard.
- Incompleteness, with only specific cases of the program functionality being expressed in the derived OCL, or with certain functions omitted. This particularly occurred with program aspects (such as file processing) not covered by the OCL standard.
- Spurious elements being introduced, particularly additional restrictions on functionality via preconditions or invariants.

These issues were all reduced significantly by the use of fine-tuning to produce the LLM4Models LLM.

LLM4Models has superior usability compared to AgileUML, because no MDE technical knowledge is needed to use the LLM, instead a user only needs to know how to query the LLM with the standard prompt for reverse-engineering and the specific program to be analyzed.

4.5 Summary of contributions

In this paper, we have:

1. Provided the first rigorous comparison of an MDRE program abstraction approach with LLM-based program abstraction.
2. Used an MDRE approach to fine-tune a general purpose LLM to enhance its accuracy and quality for reverse engineering in a re-engineering context.
3. Enabled reverse-engineering practitioners to use an LLM with a conversational interface for accurate program abstraction, in place of MDRE tools requiring specialized skills to utilize.

4.6 Future work and extensions

The work presented here may be extended in various directions:

- By using larger training datasets, a wider range of source programs may be brought into the scope of LLM4Models, in particular, Java or Python programs using specialized libraries and facilities could be handled.
- The same approach could be used to address other source programming languages, such as COBOL and VB, for which AgileUML provides abstractors, but for which LLM program translation or reverse-engineering support is currently lacking (Gandhi et al., 2024).
- Further post-processing checks and corrections could be applied to detect and remove spurious model elements produced by the LLM, thus improving the consistency results.

Data availability statement

The datasets presented in this study can be found in online repositories. The names of the repository/repositories and accession number(s) can be found in the article/supplementary material.

Author contributions

HS: Writing – original draft, Writing – review & editing. KL: Writing – original draft, Writing – review & editing.

Funding

The author(s) declare that no financial support was received for the research and/or publication of this article.

Acknowledgments

Hanan Siala would like to thank the Libyan Ministry of Higher Education and Scientific Research for its financial support.

Conflict of interest

The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Generative AI statement

The author(s) declare that no Gen AI was used in the creation of this manuscript.

Publisher's note

References

Ahmad, W., Tushar, M., Chakraborty, S., and Chang, K.-W. (2023). AVATAR: a parallel corpus for Java-Python program translation. *arXiv* [Preprint]. arXiv:2108.11590. doi: 10.48550/arXiv.2108.11590

Boronat, A., and Mustafa, J. (2025). "MDRE-LLM: a tool for analysing and applying LLMs in software reverse engineering," in *SANER '25* (Montreal, QC: IEEE). doi: 10.1109/SANER64311.2025.00090

Bowen, J., Breuer, P., and Lano, K. (1993a). A compendium of formal techniques for software maintenance. *IEE/BCS Softw. Eng. J.* 8, 253–262. doi: 10.1049/sej.1993.0031

Bowen, J. P., Breuer, P., and Lano, K. (1993b). Formal specifications in software maintenance: from code to Z++ and back again. *Inf. Softw. Technol.* 35, 679–690. doi: 10.1016/0950-5849(93)90083-F

Buttner, F., and Gogolla, M. (2014). On OCL-based imperative languages. *Sci. Comput. Program.* 92, 162–178. doi: 10.1016/j.scico.2013.10.003

Chen, X., Liu, C., and Song, D. (2018). "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: Curran Associates Inc), 2552–2562.

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

Dettmers, T., Pagnoni, A., Holtzman, A., and Zettlemoyer, L. (2024). "QLoRA: efficient fine-tuning of quantized LLMs," in *Advances in Neural Information Processing Systems, Vol. 36* (Cambridge, MA: MIT Press).

Gandhi, S., Patwardhan, M., Khatri, J., Vig, L., and Medicherla, R. K. (2024). "Translation of low-resource COBOL to logically-correct and readable Java leveraging high-resource Java refinement," in *LLM4Code* (New York, NY: IEEE). doi: 10.1145/3643795.3648388

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., and Liu, S., et al. (2021). "GraphCodeBERT: pre-training code representations with dataflow," in *ICLR 2021* (Cambridge, MA: Microsoft Research).

Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L., et al. (2023). LLMs for software engineering: a systematic literature review. *arXiv* [Preprint]. arXiv:2308.10620. doi: 10.48550/arXiv.2308.10620

Hu, E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., et al. (2022). "LoRA: low-rank adaptation of large language models," in *ICLR 2022* (Cambridge, MA: Microsoft Research).

Jana, P., Jha, P., Ju, H., Kishore, G., Mahajan, A., and Ganesh, V. (2023). Attention, compilation, and solver-based symbolic analysis are all you need. *arXiv* [Preprint]. arXiv:2306.06755. doi: 10.48550/arXiv.2306.06755

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D., et al. (2023). Mistral 7b. *arXiv* [Preprint]. arXiv:2310.06825. doi: 10.48550/arXiv.2310.06825

Krasteva, I., Stavru, S., and Ilieva, S. (2013). "Agile software modernization to the service cloud," in *ICIW 2013* (Rome), 1–9.

Lachaux, M.-A., Roziere, B., Chanussot, L., and Lample, G. (2020). Unsupervised translation of programming languages. *arXiv* [Preprint]. arXiv:2006.03511. doi: 10.48550/arXiv.2006.03511

Lano, K., Haughton, H., Yuan, Z., and Alfraihi, H. (2024). Agile model-driven re-engineering. *Innov. Syst. Softw. Eng.* 20, 559–584. doi: 10.1007/s11334-024-00568-z

Lano, K., Kolahdouz-Rahimi, S., and Jin, K. (2022). "OCL libraries for software specification and representation," in *OCL 2022, MODELS 2022 Companion Proceedings* (New York, NY: ACM). doi: 10.1145/3550356.3561565

Lano, K., and Malik, N. (1999). Mapping procedural patterns to object-oriented design patterns. *Autom. Softw. Eng.* 6, 265–289. doi: 10.1023/A:1008708927260

Lano, K., and Siala, H. A. (2024a). Using MDE to automate software language translation. *Autom. Softw. Eng.* 31:20. doi: 10.1007/s10515-024-00419-y

Lano, K., and Siala, H. A. (2024b). "Using OCL for verified re-engineering," in *MoDeVVa 2024, MODELS 2024* (New York, NY: ACM). doi: 10.1145/3652620.3687824

Li, X., Yuan, S., Gu, X., Chen, Y., and Shen, B. (2024). Few-shot code translation via task-adapted prompt learning. *J. Syst. Softw.* 212:112002. doi: 10.1016/j.jss.2024.112002

Liu, X., Yang, H., and Zedan, H. (1997). "Formal methods for the re-engineering of computing systems," in *Compsac '97* (Washington, DC: IEEE). doi: 10.1109/CMPSAC.1997.625024

Malyaya, A., Zhou, K., Ray, B., and Chakraborty, S. (2023). On ML-based program translation: perils and promises. *arXiv* [Preprint]. arXiv:2302.10812. doi: 10.48550/arXiv:2302.10812

Marco, A. D., Iancu, V., and Asinofsky, I. (2018). "COBOL to Java and newspapers still get delivered," in *Proceedings IEEE International Conference on Software Maintenance and Evolution* (Madrid: IEEE Press), 583–586. doi: 10.1109/ICSME.2018.00055

Nguyen, A. T., Nguyen, T. T., and Nguyen, T. N. (2015). "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* (Lincoln, NE: IEEE Press), 585–596. doi: 10.1109/ASE.2015.74

OMG (2014). *Object Constraint Language (OCL) 2.4 Specification*. Milford, MA.

Ouyang, S., Zhang, J., Harman, M., and Wang, M. (2023). LLM is like a box of chocolates: the non-determinism of ChatGPT in code generation. *arXiv* [Preprint]. arXiv:2308.02828. doi: 10.48550/arXiv.2308.02828

Perez-Castillo, R., de Guzman, I. G.-R., and Piattini, M. (2010). "Implementing business process recovery patterns through QVT transformations," in *ICMT 2010* (Cham: Springer). doi: 10.1007/978-3-642-13688-7_12

Perez-Castillo, R., de Guzman, I. G.-R., and Piattini, M. (2011). Knowledge discovery metamodel ISO/IEC 19506: a standard to modernize legacy systems. *Comput. Standar Interfaces* 33, 519–532. doi: 10.1016/j.csi.2011.02.007

Siala, H. A. (2024). "Enhancing model-driven reverse-engineering using machine learning," in *Doctorial Symposium, ICSE 2024* (New York, NY: ACM). doi: 10.1145/3639478.3639797

Siala, H. A., and Lano, K. (2025). "Using LLMs to extract OCL specifications from Java and Python programs: an empirical study," in *Proceedings of the STAF 2025 Workshops: OCL 2025* (CEUR; RWTH).

Siala, H. A., Lano, K., and Alfraihi, H. (2024). Model-driven approaches for reverse engineering-a systematic literature review. *IEEE Access* 12, 62558–62580. doi: 10.1109/ACCESS.2024.3394732

Sneed, H. (2011). "Migrating from COBOL to Java: a report from the field," in *IEEE Proc. of 26th ICSM* (Timisoara: IEEE Press), 1–7. doi: 10.1109/ICSM.2010.5609583

Sneed, H., and Jandrasics, G. (1987). "Inverse transformation of software from code to specification," in *IEEE Conf. Soft. Maintenance* (Scottsdale, AZ: IEEE). doi: 10.1109/ICSM.1988.10149

Szafraniec, M., Roziere, B., Leather, H., Charton, F., Labatut, P., and Synnaeve, G. (2023). Code translations with compiler representations. *arXiv* [Preprint]. arXiv:2207.03578. doi: 10.48550/arXiv.2207.03578

Zhang, C., Wang, J., Zhou, Q., Xu, T., Tang, K., Gui, H., et al. (2022). A survey of automated code summarization. *Symmetry* 14:471. doi: 10.3390/sym14030471

Zhao, W., Zhou, K., Li, J., Tang, T., Wang, X., Hou, Y., et al. (2023). A survey of large language models. *arXiv* [Preprint]. arXiv:2303.18223. doi: 10.48550/arXiv.2303.18223

Zhu, M., Suresh, K., and Reddy, C. (2022). Multilingual code snippets training for program translation. *Proc. AAAI Conf. Artif. Intell.* 36, 11783–11790. doi: 10.1609/aaai.v36i10.21434