Software language translation by example

Q. Xue[1], Kevin Lano[1]

[1]King's College London, Strand, London, UK

Abstract

In order to improve the usability, flexibility and agility of model-driven engineering (MDE), various approaches have been adopted to facilitate the creation and adaptation of MDE tools by end-user software practitioners, and to reduce the level of MDE skills necessary for such tasks. Two important techniques are the use of *examples* to specify MDE tools such as model transformations, and the use of specifications based on the *concrete syntax* of the software languages being processed by transformations, instead of the language metamodels. In this paper we combine these two techniques with symbolic machine learning, in order to derive language-to-language transformations from sets of examples expressed using concrete syntax. The approach is demonstrated on a program abstraction task (i.e., reverse engineering), a DSL-to-code code generation task, and a refactoring task.

Keywords

Specification by Example, Model-driven Engineering, Software Language Translation, Agile Development

1. Introduction

Model-driven engineering has become a widely-adopted practice across several application areas of software systems, however barriers to MDE use remain in many industries because of the specialised skills and knowledge needed to apply MDE or to use MDE tools [1, 2, 3, 4].

In [5] we introduced the idea of using *code generation by example* (CGBE) to derive MDE code generators (mapping from UML models to a target executable language such as Java) from paired source-target examples. CGBE is a search-based software engineering technique which searches for plausible model-to-code transformations which map the source element of each pair in a provided set of example translation pairs to the target element of the pair. This process is also a form of symbolic machine learning (ML) which induces precise mapping rules from training data (the example pairs). The resulting transformation is expressed in concrete textual syntax using the $\mathcal{CSTL}$ text-to-text transformation language [6].

For instance, the general OCL-to-Java mapping rule

```
_1->union(_2) |-->Ocl.union(_1,_2)
```

can be learnt from 3 paired examples of the mapping.

This process reduces the effort of manually constructing code generators, and reduces the knowledge required by the user: they only need to know the concrete syntax of the source and

AMDE 2024: Agile Model-driven Engineering Workshop, Part of the Software Technologies: Applications and Foundations (STAF) federated conferences, Eds. S. Tehrani, H. Alfraihi, S. Rahimi and J. Troya, 8–12 July 2024, Enschede, Netherlands.

qiaomu.xue@kcl.ac.uk (Q. Xue); kevin.lano@kcl.ac.uk (K. Lano)
0000-0002-9706-1410 (K. Lano)

target languages, in order to express the examples. Knowledge of the source/target language grammars, language metamodels or of $\mathcal{CSTL}$ syntax/semantics is not needed.

However, the CGBE process of [5] is mainly oriented to the case where the source language is UML/OCL and the target language is a programming language, with the source language syntax and grammar structure being less complex than that of the target language. In this paper we generalise the procedure to *language translation by example* (LTBE) to support the construction of a wider range of MDE transformation tools, in particular to support *reverse engineering* or *program abstraction* from programming language code to UML, design refactoring, and code synthesis from a DSL to programs.

Subsequently to [5], there have been significant advances in non-symbolic ML approaches for language transformation synthesis, using large language models (LLMs) such as GPT-3 [7]. These have been applied in particular to learn translations between programming languages, i.e., *program translation* [8, 9, 10]. Thus it is important to compare CGBE and LTBE with LLM approaches for language transformation.

In Section 2 we describe our generalised procedure, and the $\mathcal{CSTL}$ language, and in Section 3 show how this can be applied to a practical reverse engineering task of learning a Pascal-to-UML abstraction transformation. Section 4 applies LTBE to the synthesis of a DSL-to-Python code generator, Section 5 applies LTBE to the synthesis of a design refactoring transformation, and Section 6 compares the results of LTBE with manually-coded translators and application of the GPT-3.5 LLM. Section 7 considers limitations and future work.

2. Language translation by example (LTBE)

The goal of LTBE is to automatically derive a $\mathcal{CSTL}$ transformation τ mapping a software language L_1 to a language L_2 (possibly the same as L_1), based on a set D of examples of corresponding texts from L_1 and L_2. D should be functional from source to target, and each example should be valid according to its language grammar.

We require that the synthesised τ should be correct wrt D, i.e., it should correctly translate the source part of each example $d \in D$ to the corresponding target part of d.

In addition, τ should also be able to correctly translate the source elements of an independent validation dataset V of (L_1, L_2) examples, disjoint from D.

2.1. $\mathcal{CSTL}$ syntax and semantics

$\mathcal{CSTL}$ specifications or scripts consist of a set of named *rulesets* with the syntax

```
RulesetName::
(rule)+
```

I.e., each ruleset consists of one or more rules. The ordering of rules represents their relative priority: higher-priority and more specialised rules are listed before more general/default rules.

The names of rulesets are usually the same as the source language grammar non-terminal symbols (i.e., the names of the syntactic categories recognised by a parser for the grammar). However, user-defined rulesets can also be added to a $\mathcal{CSTL}$ script. These rulesets can define custom functions to derive specific information from source elements – e.g., to test if a certain kind of language item occurs anywhere within the source element, or to derive a list of identifiers used within the source element. Context-specific mappings can also be performed by user-defined rulesets.

Rulesets named by a grammar non-terminal are applied to parse trees whose outermost tag equals the name (i.e., they were produced as a result of parsing input text using a grammar production for that tag/non-terminal).

Each rule in a ruleset has the syntax

```
LHS |-->RHS (<when> conditions)? (<action> actions)?
```

The left-hand side (LHS) is schematic text in the source language concrete syntax, and the right-hand side (RHS) is schematic text in the target language syntax. E.g., to express a translation from the Pascal 'not equal' operator $<>$ to the OCL operator $/=$, we could write the rule

```
_1 <> _2 |-->_1 /= _2
```

The _1 and _2 are metavariables representing subparts of the source element being processed by the rule. In the result text produced by the rule (the RHS) the metavariables are replaced by the text which the entire transformation produces for these elements.

The optional $<$*when*$>$ and $<$*action*$>$ clauses contain *conditions* and *actions*, each of these are comma-separated sequences of pairs *argument predicate* which either (in the case of conditions) test if the *argument*, such as a metavariable, satisfies a condition *predicate* (such as being of a type named by the predicate, or being equal to a given value denoted by the predicate), or (for actions) assert that the *argument* does satisfy *predicate*. Condition predicates can also be parse tree tag names, to test if the source element is of the syntactic category named by the tag. All conditions of a rule must evaluate to true in order for the rule to be applied to a source element, which must also syntactically match the rule LHS.

For example, the rules:

```
_1 + _2 |-->_1->union(_2)<when> _1 Set
_1 + _2 |-->_1 + _2
```

map Pascal expressions $a + b$ either to $a' \rightarrow union(b')$ if a is known to be Set-typed, or to $a' + b'$ otherwise, where e' is the result of translating e.

Actions are mainly used when processing declarations of elements, such as variables, types or operations, in order to retain information about the elements for use in processing other parts of a source language text, such as behaviour specifications or code. Users may define their own predicate names to use in actions and conditions, but there are standard names *int*, *double*, *Set*, *Sequence*, *Map*, etc, corresponding to OCL datatype names.

User-defined rulesets f can be explicitly invoked (in the RHS of a rule) by the notation `_i‘f` on a metavariable $\_i$.

Rules can be ordered according to a specialisation partial order $<$. Rules satisfy $r1 < r2$ if:

- $r1$, $r2$ have the same conditions, but the LHS of $r2$ is more general than that of $r1$: the same pattern but with more metavariables. In other words, the LHS of $r1$ is an instantiation or special case of the LHS of $r2$

- The LHS of $r1$, $r2$ are the same, but the conditions of $r1$ include those of $r2$ and extend these.

Rules within a ruleset should be listed in $<$ order.

2.2. Extensions of CGBE for program abstraction and design refactoring

To learn $\mathcal{CSTL}$ scripts mapping from text examples in the concrete syntax of language L_1 to text examples in L_2 concrete syntax, these examples are parsed by parsers for the respective languages, producing *parse trees* of the texts. For example, parse trees for the Pascal example expression $3 \ <> \ 1$ and corresponding OCL expression $3\ / = 1$ are shown respectively on the left and right side of Figure 1.

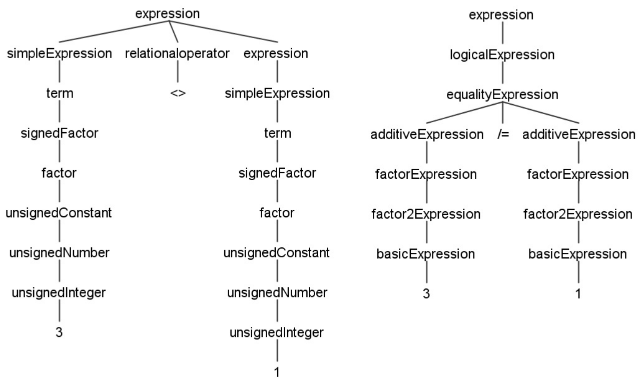

Figure 1: Pascal and OCL parse trees

The parse trees can be written in the form of nested lists, e.g., as

```
(equalityExpression
  (additiveExpression
    (factorExpression (factor2Expression (basicExpression 3)))) /=
  (additiveExpression
    (factorExpression (factor2Expression (basicExpression 1)))))
```

for the OCL version of the inequality expression. We refer to the terminal nodes of parse trees (such as ‘1’, ‘3’ and ‘/=’) as *symbols*: they are instances of terminal symbols of the language grammar.

The CGBE process defined in [5] used a series of six strategies to examine the parse trees s, t of paired source and target examples, to discover systematic rules to map the source tree structures to target tree structures. Table 1 summarises these strategies.

The strategies are oriented to the situation where the source parse tree is generally simpler in structure than the target: they search for embeddings of the source tree within the target tree. However, when learning language-to-language mappings such as program abstractions, the reverse situation may hold: the source (program) parse tree will often be more complex in structure than the target (UML/OCL) parse tree, as in Figure 1. Thus we need to include the case that one subelement of the source elements consistently maps to the entire target: $s_j \longmapsto t$.

Table 1
Tree-to-tree mapping strategies for CGBE

Strategy	*Conditions*	*Mapping*
1	All source trees s have same tag $tag1$ and same arity $k > 0$. All target trees t have same tag $tag2$ and same arity $n > 0$.	Target subtrees t_i at each position $i : 1..n$ in ts are constants or mapped from source trees s_j at some fixed $j : 1..k$, or are mapped from the entire s trees.
2	Source and target terms can have varying arity, but each s has same arity as paired t. Tags as for 1.	Target subtrees t_i at each position i in each t are mapped from source subtrees s_i at position i in each s.
3	As for 2, but s, t are only required to have the same number of non-symbol subtrees.	i-th non-symbol subtree of t corresponds to i-th non-symbol subtree of s. s symbols can be consistently replaced in t or deleted, new symbols can be inserted in the t.
4	Source s all have same tag and arity $k > 0$. Subtrees s_i at position i are always symbols, which may vary.	Conditional mapping based on s_i value, if mappings found for sets of s, t pairs which have each different s_i value.
5	As 3, but the s trees may have more non-symbol subtrees than the paired ts.	All non-symbol direct subtrees of t are mapped from non-symbol subtrees of s, in ascending order. Symbols of s may be deleted/replaced in t, and new symbols inserted.
6	Source s all have same tag, arity $k > 0$, t have same tag, possibly different arities.	Target subsequences $t_p..t_q$ correspond to some s_j, for fixed p, $q > p$, $j : 1..k$.

The learning of conditional mappings

```
LHS |-->RHS<when> _i p
```

also needs to be generalised. Strategy 4 of Table 1 only considers equality-test conditions where p denotes a value for a source symbol (e.g., to give different mappings for $a + b$ and $a - b$). But for general language mappings, the case of tests on typing predicates p are also needed (because the same syntactic source text may translate differently depending upon the types of its elements, as with the conditional example of Section 2.1). Similarly, tests on syntactic category (tag name) predicates need to be considered.

Thus we have extended and modified the strategies as follows (Table 2).

The analysis of [5] regarding the time complexity of the CGBE search procedure still holds true for the extended LTBE search. Thus the overall complexity of the search is of the order of

$$(N * n * m)^{max(P,Q)}$$

where N is the number of examples, m, n are the maximum tree widths at any level within the source, target examples, respectively, and P, Q are the maximum tree depths of the source, target examples.

Table 2
Extended mapping strategies for LTBE

Strategy	*Conditions*	*Mapping*
7	All source trees s have same tag $tag1$ and same arity $k > 0$. All target trees t have same tag $tag2$.	Entire target trees t are mapped from source trees s_j at some fixed position $j : 1..k$
8	Source s all have same tag, arity. Subtree s_i is always a non-symbol, with a tag, which may vary.	Conditional mapping based on s_i tag if mappings found for pairs with each different s_i tag value.

This result indicates that the simplest examples should be used which are sufficient to learn the required mappings.

Finally, in order to reduce the effort required by symbolic ML, it is possible to split a LTBE task into parts, whereby distinct sublanguages of a given source L_1 are processed separately: L_1 is partitioned into subsets LS_1, ..., LS_n, with the LS_i having disjoint sets of outermost tags in parse trees of L_1 elements, and the mappings LS_i to L_2 are learnt separately, resulting in $\mathcal{CSTL}$ scripts τ_1, ..., τ_n. Typical partitions could be the expression, type, statement and declaration sublanguages of a software language.

This process may result in rulesets with the same name occurring in two or more τ_i. Such rulesets should be merged by taking the union of the ruleset rules and listing them in $<$-order.

3. Program abstraction by example: Pascal to UML/OCL

This language translation task is a program abstraction or reverse-engineering transformation from Pascal programs to UML/OCL class diagram specifications. The key idea is to abstract Pascal record types P to UML classes consisting of attributes for each field of P, and containing operations for each Pascal procedure or function that operates on P instances. Pascal statements are translated to procedural OCL statements, Pascal expressions are translated to OCL expressions, and Pascal types translated to OCL types. Thus the abstraction can be naturally subdivided into parts for the expression, type, statement and declaration subparts of the Pascal language.

Table 3
Pascal to UML/OCL using LTBE

Dataset	*Size (examples)*	*ML time (ms)*	*Ruleset size (LOC)*	*Accuracy*
Expressions	102	1592	105	0.9
Statements	29	3603	100	1.0
Types	20	121	126	1.0
Declarations	20	455	71	0.7
Averages	43	1443	100.5	0.9

Table 3 summarises the results of LTBE for the Pascal to UML/OCL abstraction task. An example statement rule derived by LTBE is:

```
repeat _2 until _4  |-->( _2 ; while _4 do _2 )
```

ML training times are computed as the average of 3 complete training sessions, on an i7 64-bit laptop running Windows 10 OS. Accuracy is computed as the proportion of correct translations on the validation dataset. This dataset consists of 51 examples split into 4 parts for the separate language subparts.

The accuracy of the expression translation could be increased to 100% by adding two more examples for the *set_* non-terminal.

4. DSL code-generation by example: Collection expressions to Python

This language translation task is a refinement, mapping from a simple OCL sublanguage for collection expressions of the forms

```
s->select( x | P )
s->reject( x | P )
s->collect( x | e )
```

to the corresponding Python expression forms

```
[ x for x in s where P ]
[ x for x in s where not P ]
[ e for x in s ]
```

Table 4 shows the results for this application of LTBE. Training times are calculated as for Table 3.

Table 4
OCL collection expressions to Python using LTBE

Size (examples)	*ML time (ms)*	*Ruleset size (LOC)*	*Accuracy*
12	557	115	1.0

5. Refactoring by example: Design reduction

Program reduction [11] is an established technique to simplify code in order to facilitate program analysis. A similar transformation can be specified at the design model level, either by manually-coded rules in $\mathcal{CSTL}$ or using LTBE. Unlike the previous transformation examples, this is a case of an *endogenous* transformation, where the source and target languages (UML/OCL) are the same.

An example rule of the transformation is:

```
if true then _1 else _2 |-->_1
```

Table 5 gives the results of applying LTBE to learn this transformation.

Table 5
Learning OCL statement refactoring using LTBE

Size (examples)	*ML time (ms)*	*Ruleset size (LOC)*	*Accuracy*
26	118	44	0.7

6. Evaluation and comparisons

The results of Tables 3, 4, 5 show that the LTBE procedure is able to learn accurate abstraction, refactoring and refinement language mappings in a practical time, from relatively small sets of examples. However, as Table 6 shows, the LTBE-synthesised Pascal abstractor is less complete and less accurate than the manually-constructed $\mathcal{CSTL}$ transformation *pascal2UML* of [12], while consuming less resources to create and execute. Completeness is measured as the proportion of source language grammar productions which have corresponding $\mathcal{CSTL}$ rules in the transformation. Accuracy is measured as the percentage of correct translations by each transformation, applied to the same validation set of 51 Pascal examples used for Table 3. Performance is the total execution time taken for processing the validation examples.

Table 6
Pascal to UML/OCL transformations

Transformation	*Completeness (coverage)*	*Effort (person days)*	*Accuracy*	*Performance (ms)*
LTBE	40%	2	0.94	1029
Manual [12]	95%	25	0.96	2243

Table 7 compares the manually-produced and LTBE synthesised versions of the refactoring transformation. As with the Pascal abstraction case, the learnt transformation is inferior to the manual version in terms of completeness and accuracy, but took significantly less time to develop. Accuracy is measured as the percentage of correct translations of the validation examples used in Table 5. Performance is also measured on these examples.

Table 7
OCL refactoring transformations

Transformation	*Completeness (coverage)*	*Effort (person days)*	*Accuracy*	*Performance (ms)*
LTBE	60%	0.5	0.7	123
Manual [13]	90%	5	1.0	183

We conclude from these results that the LTBE production of $\mathcal{CSTL}$ transformations can be useful to quickly build initial versions of a complex transformation, but that manual refinement remains necessary to achieve high accuracy and completeness. In this respect complete automation is not yet possible.

Large language models (LLMs) have been used for code generation, program translation and many other kinds of software processing [14]. Thus it is interesting to evaluate their performance on tasks such as reverse engineering and refactoring. We evaluated the ability of GPT-3.5[1] to correctly abstract Pascal functions to OCL operations, and compared the results to those of the LTBE-synthesised and manually-created Pascal to UML translators (second column of Table 8). The instruction

> “Translate the following Pascal function to an OCL operation definition"

was used as a prompt for the abstraction task.

We also evaluated the capability of the LLM for the refactoring task. The instruction

> “Refactor this Pascal code fragment into a simplified equivalent form"

was used as a GPT prompt for the refactoring task. The results for this task are shown in the third column of Table 8.

Table 8
Comparison of LLM results with rule-based translators

Translator version	*Abstraction accuracy*	*Refactoring accuracy*
LTBE synthesised	0.7	0.8
Manually coded	1.0	1.0
GPT-3.5 LLM	0.8	0.6

Unlike the rule-based translators, GPT-3.5 is ‘creative’ in its responses – the responses may differ if the same instruction and input example are re-submitted at different times, and spurious details may be added in the result, or assumptions made about the input, which the rule-based approaches do not make.

As an example, for the abstraction task, the input Pascal function

```
function findpi : Real ;
begin
  findpi := arctan(0.5)
end
```

is translated to

```
context AnyClass::findpi() : Real
pre: true
post: result = (3.141592653589793 / 180) *
  (180 - (180 * atan(0.5) / 3.141592653589793))
```

by GPT-3.5.

The refactoring task illustrates that GPT 3.5 has partial but not complete understanding of program semantics. For example, the input

```
for j := 2 to 1 do x := x+1
```

is recognised as a loop that never executes, but the answer to the prompt is the incorrect `x := x+1`. We found similar results with the MetaAI (LLama3), Gemini and Mistral LLMs. All artefacts used in this evaluation are provided at zenodo.org/records/11654783.

[1]chat.openai.com

7. Limitations and future work

As with CGBE, the LTBE process does not learn rule *actions*, however default actions could be added to each rule that processes a source language declaration, for example:

```
_1 : _2 |-->RHS<action> _1 _2
```

in the case of a Pascal parameter declaration.

For the learning of rule conditions (strategy 4 in Table 1 and 8 in Table 2), only syntactic conditions based on the value or syntactic category (tag name) of argument places can be learnt. Semantic conditions based on type predicates cannot be directly learnt, because this information is not available in the training dataset. A possible solution would be to use *triples* as training data instead of pairs, with an additional first item that gives the contextual type of the identifiers in the source example. This would lead to more precise learnt rules, but at the cost of a more complex training process and more complex datasets.

Instead, we recommend the use of syntactic *proxies* for semantic categories where possible. This means adopting a particular tag as a marker for a semantic category in training data. Thus examples involving Pascal sets would use explicit set values (which have the tag *set_*). The above example would therefore be expressed as

```
[x]+t     Set{x}->union(t)
```

The induced rule would use the syntax category in its condition:

```
_1 + _2 |-->_1->union(_2)<when> _1 set_
```

Because the syntactic predicate `set_` is a proxy for the semantic predicate *Set*, we can also derive the rule

```
_1 + _2 |-->_1->union(_2)<when> _1 Set
```

Summary

In this paper we have considered how the use of example-based specification and concrete-syntax transformation can lead to simpler use of MDE. We improved the CGBE process to generalise and extend this to a process, LTBE, to build a wider range of text-to-text transformations from examples, including program abstraction and refactoring transformations. The evaluation results show that LTBE can reduce the effort required to construct transformations, however full automation of complete and highly-accurate transformations is not achieved. We found that while LLMs are able to partly perform abstraction and refactoring tasks, they also have deficiencies in terms of accuracy and reliability.

References

[1] H. Alfraihi, K. Lano, Trends and insights into the use of model-driven engineering: a survey, in: SAM/MODELS, 2023.

[2] S. Abrahão, F. Bourdeleau, B. Cheng, S. Kokaly, R. Paige, H. Stöerrle, J. Whittle, User experience for model-driven engineering: Challenges and future directions, in: 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems (MODELS), IEEE, 2017, pp. 229–236.

[3] A. Bucchiarone, J. Cabot, R. Paige, A. Pierantonio, Grand challenges in MDE: an analysis of the state of the research, SoSyM 19 (2020) 5–13.

[4] S. Hoppner, Y. Haas, M. Tichy, K. Juhnke, Advantages and disadvantages of (dedicated) model transformation languages, Empirical Software Engineering 27 (2022).

[5] K. Lano, Q. Xue, Code generation by example using symbolic machine learning, Springer Nature Computer Science (2023).

[6] K. Lano, Q. Xue, Agile specification of code generators for model-driven engineering, in: 2020 15th International Conference on Software Engineering Advances (ICSEA), 2020, pp. 9–15.

[7] W. Zhao, et al., A survey of large language models, arXiv 2303.18223v10 (2023).

[8] W. Ahmad, M. Tushar, S. Chakraborty, K.-W. Chang, AVATAR: a parallel corpus for Java-Python program translation, arXiv:2108.11590v2 (2023).

[9] X. Li, et al., Few-shot code translation via task-adapted prompt learning, Journal of Systems and Software (2024).

[10] A. Malyaya, et al., On ML-based program translation: perils and promises, arXiv:2302.10812v1 (2023).

[11] C. Sun, et al., Perses: Syntax-guided program reduction, in: ICSE 2018, 2018.

[12] K. Lano, H. Siala, Using MDE to automate software language translation, Automated Software Engineering 31 (2024).

[13] K. Lano, Q. Xue, H. Haughton, A concrete syntax transformation approach for software language processing, Springer Nature Computer Science (2024).

[14] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, H. Wang, Llms for software engineering: a systematic literature review, arXiv 2308.10620 (2023).