SN Computer Science (2023) 4:170
https://doi.org/10.1007/s42979-022-01573-4

ORIGINAL RESEARCH

Code Generation by Example Using Symbolic Machine Learning

Kevin Lano[1] · Qiaomu Xue[1]

Received: 20 May 2022 / Accepted: 16 December 2022 / Published online: 17 January 2023

Abstract
Code generation is a key technique for model-driven engineering (MDE) approaches of software construction. Code generation enables the synthesis of applications in executable programming languages from high-level specifications in UML or in a domain-specific language. Specialised code generation languages and tools have been defined; however, the task of manually constructing a code generator remains a substantial undertaking, requiring a high degree of expertise in both the source and target languages, and in the code generation language. In this paper, we apply novel symbolic machine learning techniques for learning tree-to-tree mappings of software syntax trees, to automate the development of code generators from source–target example pairs. We evaluate the approach on several code generation tasks, and compare the approach to other code generator construction approaches. The results show that the approach can effectively automate the synthesis of code generators from examples, with relatively small manual effort required compared to existing code generation construction approaches. We also identified that it can be adapted to learn software abstraction and translation algorithms. The paper demonstrates that a symbolic machine learning approach can be applied to assist in the development of code generators and other tools manipulating software syntax trees.

Keywords Code generation · MDE · Machine learning · MTBE · CGBE

Introduction

Model-driven engineering (MDE) has many potential benefits for software development, as a means for representing and managing core business concepts and rules as software models, and thus ensuring that these business assets are retained in a platform-independent manner over time. Despite the long-term advantages of MDE, many businesses and organisations are discouraged from adopting it because of the high initial costs and specialised skills required. Our research is intended to remove these obstacles by enabling general software practitioners to apply MDE techniques, via the use of simplified notations and by providing AI support for MDE processes.

This article is part of the topical collection "Advances on Model-Driven Engineering and Software Development" guest edited by Luís Ferreira Pires and Slimane Hammoudi.

✉ Kevin Lano
kevin.lano@kcl.ac.uk

Qiaomu Xue
qiaomu.1.xue@kcl.ac.uk

1 Department of Informatics, King's College London, Strand, London WC2R 2LS, UK

One area where there have been particular problems for industrial users of MDE is in the definition and maintenance of code generators [32]. Code generation in the model-driven engineering (MDE) context involves the automated synthesis of executable code, usually for a target third-generation language (3GL), such as Java, C, C++, Go or Swift, from a software specification defined as one or more models in a modelling language such as the Unified Modelling Language (UML) with formal Object Constraint Language (OCL) constraints [28] or in a domain-specific modelling language (DSL). MDE code generation has potentially high benefits in reducing the cost of code production, and in improving code quality by ensuring that a systematic architectural approach is used in system implementations. However, the manual construction of such code generators can involve substantial effort and require specialised expertise in the transformation languages used. For example, several person-years of work were required for the construction of one UML to Java code generator [7].

To reduce the knowledge and human resources needed to develop code generators, we define a novel symbolic machine learning (ML) approach to automatically create code generation rules based on translation examples. We term this approach *code generation by example* or CGBE. The basis of CGBE is the learning of tree-to-tree mappings between the abstract syntax trees (ASTs) of source language examples and those of corresponding target language examples. A set of search strategies are used to postulate and then check potential tree-to-tree mappings between the language ASTs. Typically, the source language is a subset of the Unified Modelling Language (UML) and Object Constraint Language (OCL), and the target language is a programming language, such as Java or Kotlin. However, the technique is applicable in principle to learning mappings between any software languages which have precise grammar definitions.

The CGBE approach for UML/OCL was evaluated on a wide range of target programming languages, and the results showed that a large part of the relevant code generators could typically be synthesised from examples, thus considerably reducing the manual effort and expertise required for code generator construction.

To summarise our contribution, we have provided a new technique (CGBE) for automating the construction of code generators, via a novel application of symbolic machine learning. We have also evaluated CGBE on realistic examples of code generation tasks, to establish that it is effective for such tasks.

In "Code Generation Technologies", we survey the existing MDE code generation approaches and research, and "Code Generation Idioms" describes common idioms which arise in code generator processing. These idioms are used to guide and restrict the search strategies of CGBE. "Research Questions" presents the research questions that we aim to answer.

"$\mathcal{CSTL}$" describes the $\mathcal{CSTL}$ code generation language which we use to express code generator rules. "Synthesis of Code Generators from Examples" describes the detailed process of CGBE using symbolic ML. We provide an evaluation of the approach in "Evaluation", a summary of related work in "Related Work", threats to validity in "Threats to Validity", and conclusions and future work in "Conclusions and Future Work".

Code Generation Technologies

There are three main approaches to code generation in the MDE context. Code generators may directly produce target language text from models, i.e., model-to-text approaches (M2T), or produce a model that represents the target code, i.e., model-to-model (M2M) approaches [9,23]. More recently, text-to-text (T2T) code generation languages have been defined [24].

At present, code generation is often carried out by utilising template-based M2T languages such as the Epsilon Generation Language (EGL)[1] and Acceleo.[2] In these, templates for target language elements such as classes and methods are specified, with the data of instantiated templates being computed using expressions involving source model elements. Thus, a developer of a template-based code generator needs to understand the source language metamodel, the target language syntax, and the template language. These three languages are intermixed in the template texts, with delimiters used to separate the syntax of different languages. The concept is similar to the use of JSP to produce dynamic Web pages from business data. Figure 1 shows an example of an EGL script combining fixed template text and dynamic content, and the resulting generated code. The dynamic content in the template is enclosed in `[%` and `%]` delimiter brackets.

An M2M code generation approach separates code generation into two steps: (i) a model transformation from the source language metamodel to the target language metamodel [9] and (ii) text production from a target model. In this case, the author of the code generator must know both the source and target language metamodels, the model transformation language, and the target language syntax. Figure 2 shows an example of M2M code generation in QVTr from [9].

A T2T approach to code generation specifies the translation from source to target languages in terms of the source and target language concrete syntax or grammars, and does not depend upon metamodels (abstract syntax) of the languages. A T2T author needs to know only the source language grammar and target language syntax, and the T2T language.

An example of T2T code generation is the UML to Java8 code generation script of AgileUML,[3] which includes rules such as

```
BinaryExpression::
_1 * _2 |-->_1 * _2
_1 / _2 |-->_1 / _2
_1 mod _2 |-->_1 % _2
_1 div _2 |-->((int) (_1/_2))
_1->pow(_2) |-->Math.pow(_1,_2)
```

Another T2T language is Antlr's StringTemplate[4].

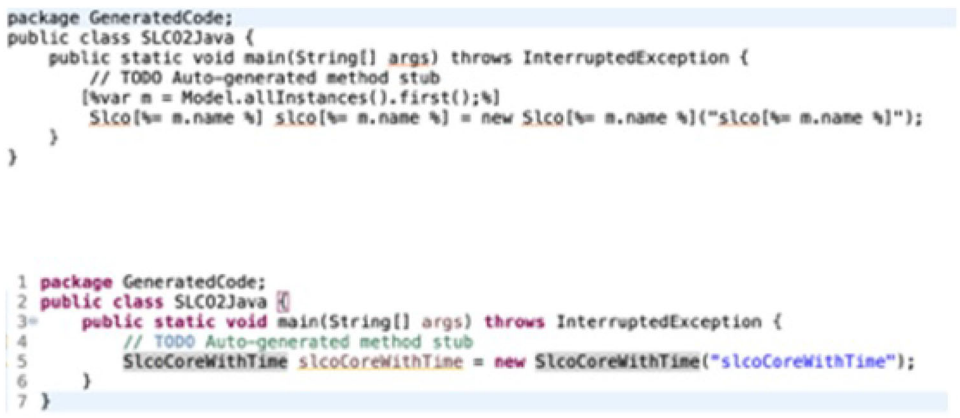
M2M and M2T approaches have the advantage of being able to use a semantically rich source representation (a model with a graph-like structure of cross-references between elements), whilst a T2T approach uses tree-structured source data (the parse tree of source text according to a grammar for the source language). However, M2M and M2T approaches require the definition of a source metamodel, which may not exist, for example, in the case of a DSL defined by a gram-

[1] https://projects.eclipse.org/projects/modeling.epsilon.

[2] https://www.eclipse.org/acceleo.

[3] https://projects.eclipse.org/projects/modeling.agileuml.

[4] http://www.stringtemplate.org/about.html.



```
package GeneratedCode;
public class SLCO2Java {
    public static void main(String[] args) throws InterruptedException {
        // TODO Auto-generated method stub
       [%var m = Model.allInstances().first();%]
        Slco[%= m.name %] slco[%= m.name %] = new Slco[%= m.name %]("slco[%= m.name %]");
    }
}
```

```
package GeneratedCode;
public class SLCO2Java {
    public static void main(String[] args) throws InterruptedException {
        // TODO Auto-generated method stub
        SlcoCoreWithTime slcoCoreWithTime = new SlcoCoreWithTime("slcoCoreWithTime");
    }
}
```

Fig. 1 Example of EGL template and generated code

```
relation Package2Package_top{
    packName : String;

    enforce domain uml uml_model : uml::Model{
          nestedPackage = uml_pack: uml::Package{
                nestingPackage = uml_model
                , name = packName
          }
    };
    enforce domain java java_model : java::Model{
          ownedElements = java_pack : java::Package{
                model = java_model
                , name = packName
                , proxy = false
                --make sure no proxy package matches
          }
    };
    where{
          AuxPackage2Package(uml_pack, java_pack);
          Package2Package_nested(uml_pack, java_pack);
    }
}
```

Fig. 2 Example of QVTr M2M code generation

mar. For these reasons, we decided to focus on learning T2T code generators, rather than M2M or M2T generators, as the goal of our research.

Code Generation Idioms

Code generation involves specific forms of processing and transformation: typically, a model or syntax tree representing the source version of a software system is traversed to construct a model or text representing the target version of the software system. Multiple traversals of the source data may be necessary, together with the construction of internal auxiliary data, in a manner similar to the processing carried out by compilers [10].

By examining a number of different M2M, M2T, and T2T code generators, we identified a set of characteristic idioms used in their processing:

Elaboration: a source element is mapped to a target element plus some additional structure/constant elements. For example, for each concrete UML class C, we could generate an additional global constructor operation $createC()$ to create instances of C.

Rearrangement: Elements of the source are re-ordered in the target. E.g., a UML declaration

```
attribute name : String;
```

becomes

```
private String name;
```

in Java.

Simplification: Elements of the input are discarded when producing the output. For example, the pre : and $post$: specification clauses of an OCL operation could be discarded when translating from OCL to Java, and only the explicit behavioural definition of the operation used for code production.

Replacement: Parts of the source are replaced in a functional manner by target parts. For example, statement group symbols (and) could be consistency replaced by block symbols { and } in mapping from procedural OCL statements to Java.

Conditional generation: Translate individual elements in alternative ways based on their properties. For example, a numeric division expression a/b would be translated to $a//b$ in Python if both a and b are of integer type, otherwise to a/b.

Context-sensitive generation: Translate individual elements in alternative ways based on their context. For example, the attributes and operations of an interface would be translated to Java in a different way to those of a general class.

Iterative generation: Process the elements of a list by successively applying the same translation to each list item in turn. For example, translating the literals of an enumeration definition in UML to corresponding literals in a programming language enum.

Accumulation: Iterate over the source model structure, gathering together all items with certain properties which are encountered. For example, gathering together all the directly owned and recursively inherited features of a UML class.

Horizontal splitting: Perform alternative processing on elements in a collection depending on their properties, partitioning them into disjoint groups in the target. For example in translation of UML class features to Go, static attributes are defined as global variables using var, outside of any class (struct), whilst non-static attributes become fields within a struct.

Vertical splitting: Generate two or more target elements from one source element. For example, producing corresponding C header and code files for a UML class.

There are also language-specific idioms for certain kinds of source/target languages:

Express types by initialisation: In cases where the source language is fully typed but the target language does not have explicit typing (e.g., Python or JavaScript), type information for source variables can be expressed by the choice of initialisation value of the variable in the target. For example, in mapping

```
var name : String;
```

to Python, we could write

```
 name = ""
```

Integers would be initialised with 0, doubles with 0.0, and Booleans with False, etc.

Perform type inference: In the opposite situation where the source language has implicit types and the target has explicit types, a type-inference strategy could be used to identify the actual type of data items, where possible.

Replace inheritance by association: If the source language has inheritance but the target language does not, one technique to represent inheritance is to embed a reference $super$ in a subclass instance, linking it to an instance of the superclass. Accesses and updates to superclass attributes f are implemented as accesses and updates to $super.f$. Likewise operation calls of inherited operations are performed via $super$. This strategy is used in the UML2C translation of [23].

Pre-normalisation: To facilitate code generation, the source model/specification may need to be rewritten into a specific form. For example, factoring out common subexpressions in arithmetic expressions [10].

Tree-to-sequence: This idiom flattens the source tree structure, so that individual subtrees of a source tree become subsequences of the result sequence. It is used in cases where the target language has a flat structure, such as an assembly language or PLC language, and the source is block-structured.

Research Questions

The paper aims to answer the following research questions:

RQ1: Can CGBE be used to learn code generation rules involving the common code generation idioms listed in "Code Generation Idioms"?
RQ2: Can CGBE be used to learn efficient and accurate code generators for practical code generation tasks?
RQ3: Are there clear benefits in using CGBE, compared to other approaches, such as manual generator construction?
RQ4: Can CGBE be adapted to the learning of abstraction and translation transformations?

We address these questions by defining a systematic process for CGBE ("Synthesis of Code Generators from Examples") and evaluating its application for a wide range of code generation tasks ("Evaluation"). As a running example, we use the translation of OCL unary expressions to code. These expressions are of two kinds: prefix unary expressions such as $-x$, $+x$, $not\ p$, and postfix arrow operator forms, such as $sq\rightarrow size()$, $str\rightarrow reverse()$, etc.

$\mathcal{CSTL}$

$\mathcal{CSTL}$ is a T2T transformation language, which was originally created to facilitate the manual construction of code generators from UML/OCL to 3GLs [24]. $\mathcal{CSTL}$ transformations map elements of a source language L_1 into the textual form of elements of a target language L_2. A $\mathcal{CSTL}$ specification consists of a set of *script* files, each script has a set of rulesets, which have a name and an ordered list of rules (Fig. 3). Rulesets usually correspond to the syntactic categories of L_1, although additional rulesets can be defined to specify auxiliary functions.

$\mathcal{CSTL}$ rules have the form

```
LHS |--> RHS <when> condition
```

where the LHS (left-hand side) and RHS (right-hand side) are schematic text representations of corresponding elements of L_1 and L_2, and the optional $condition$ can place restrictions on when the rule is applicable.

Neither the source or target metamodel is referred to, instead, a rule LHS can be regarded as a pattern for matching nodes in a parse tree of L_1 elements (such as types, expressions, or statements). When the transformation is applied to a

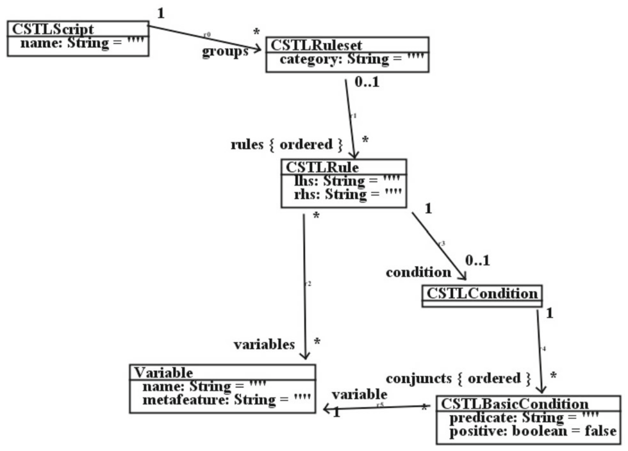


Fig. 3 $\mathcal{CSTL}$ metamodel

particular parse tree s, rule left-hand sides are tested to determine if they match s; if so, the first matching rule is applied to s.

For example, some rules for translating OCL [28] types to Java 7+ could be

```
OclType::
Integer |-->BigInteger
Real |-->BigDecimal
OclAny |-->Object
Boolean |-->boolean
String |-->String

Set(_1) |-->HashSet<_1>
Sequence(_1) |-->ArrayList<_1>
Map(_1,_2) |-->HashMap<_1,_2>
```

Metavariables _1, ..., _99 represent subnodes of an L_1 syntax tree node s. If s matches an LHS containing metavariables, these metavariables are bound to the corresponding subnodes of s, and these subnodes are then translated in turn, to construct the subparts of the RHS denoted by the metavariable.

Thus, for the above ruleset $OclType$, applied to the OCL type $Map(Integer, String)$, the final rule matches against the type, with _1 bound to $Integer$ and _2 bound to $String$. These are translated to $BigInteger$ and $String$, respectively, and hence, the output is $HashMap{<}BigInteger, String{>}$.

The special metavariable $\_*$ denotes a list of subnodes. For example, a rule

```
Set{_*} |-->Ocl.initialiseSet(_*)
```

translates OCL set expressions with a list of arguments, into a corresponding call on the static method $initialiseSet$ of the Java Ocl.java library. Elements of the list bound to $\_*$ are translated according to their own syntax category, and separators are preserved.

Rule conditions are a conjunction of basic conditions, separated by commas. Individual basic conditions have the form

```
_i S
```

or

```
_i not S
```

for a stereotype S ($predicate$ in Fig. 3), which can refer to the properties of the element bound to $\_i$. For example, the type of an element can be tested using stereotypes $Integer$, $Real$, $Boolean$, $Object$, $Sequence$, etc. Rules may also have $actions$, which have the same syntax as conditions, and are used to record information about elements which can be subsequently tested in conditions.

A ruleset r can be explicitly applied to metavariable $\_i$ representing an individual source element by the notation `_i`r`. The notation `_*`r` denotes the application of r to each element of a list $\_*$. This facility enables the use of auxiliary functions within a code generator. In addition, a separate set of rulesets in a script file $f.cstl$ can also be invoked on $\_i$ by the notation `_i`f`.

By default, if no rule in a ruleset applied to source element s matches to s, s is copied unchanged to the result. Thus, the rule $String \longmapsto String$ above is not necessary. Because rules are matched in the order of their listing in their ruleset, more specific rules should precede more general rules. A transitive partial order relation $r1 \sqsubset r2$ can be defined on rules, which is true iff $r1$ is strictly more specific than $r2$. For example, if the LHS of $r2$ and $r1$ are equal, but $r1$ has stronger conditions than $r2$.

$\mathcal{CSTL}$ is a simpler notation than template-based code generation formalisms, in the sense that no reference is made to source or target language metamodels, and no interweaving of target language text and code generation language text is necessary. The target language syntax and the structure of the source language grammar need to be known, to write and modify the rules. $\mathcal{CSTL}$ scripts are substantially more concise than equivalent Java or model transformation code [24].

$\mathcal{CSTL}$ has been applied to the generation of Swift 5 and Java 8 code, to support mobile app synthesis [20]. It has also been used to generate Go code from UML, and applied to natural language processing [22] and reverse-engineering tasks [18]. However, a significant manual effort is still required to define the $\mathcal{CSTL}$ rules and organise the transformation structure. In the next section, we discuss how this effort can be reduced by automated learning of a $\mathcal{CSTL}$ code generator from pairs of corresponding source language, target language texts. This reduces or removes the need for developers to understand the details of the source language grammar or $\mathcal{CSTL}$ syntax.

Synthesis of Code Generators from Examples

The goal of our machine learning procedure is to automatically derive a $\mathcal{CSTL}$ code generator g mapping a software language L_1 to a different language L_2, based on a set D of examples of corresponding texts from L_1 and L_2. D should be valid, i.e., functional from source to target, and each example should be valid according to its language grammar.

The generated g 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, g should also be able to correctly translate the source elements of a validation dataset V of (L_1, L_2) examples, disjoint from D.

We term this process *code generation by-example* or CGBE.

Thus, from a dataset

```
x->front()          Ocl.front(x)
sq->front()         Ocl.front(sq)
x->tail()           Ocl.tail(x)
sq->tail()          Ocl.tail(sq)
y->first()          Ocl.first(y)
arr->first()        Ocl.first(arr)
```

it should be possible to derive a specification equivalent to

```
OclUnaryExpression::
_1->front() |-->Ocl.front(_1)
_1->tail()  |-->Ocl.tail(_1)
_1->first() |-->Ocl.first(_1)
```

The datasets D will be organised in the same manner as datasets of paired texts for ML of natural language translators:[5] each line of D holds one example pair, and the source and target texts are separated by one or more tab characters $\backslash t$.

Because software languages are generally organised hierarchically into sublanguages, e.g., concerning types, expressions, statements, operations/functions, classes, etc., D will typically be divided into parts corresponding to the main source language divisions.

Symbolic Versus Non-symbolic Machine Learning

Machine learning (ML) can be defined as a technology which aims to automate the learning of knowledge from the instances of a training set, such that the learned knowledge can then be applied to derive information about other instances. A wide range of techniques are encompassed by this definition, including statistical approaches, traditional and recurrent neural networks, and symbolic learning approaches such as decision trees [29] or inductive logic programming (ILP) [26]. Machine learning typically uses a *training* phase, during which knowledge is induced from the training set, and a *validation* phase, where the accuracy of the learned knowledge is tested against a new dataset, for which the expected results are known.

The training phase can be *supervised* or *unsupervised* (or a combination of the two). Supervised learning occurs when the training data are labelled with the expected result for each item. E.g., the classification of an image as representing a person or not. Unsupervised learning occurs when such information is not given; instead, the classification categories or other results are constructed by the ML algorithm from the dataset. Our CGBE approach also adopts the division into training and validation phases, and uses supervised learning: each source language example is provided with a corresponding target language example which is the expected result of the trained translation mapping.

A major division between ML approaches is between non-symbolic approaches such as neural nets, where the learned knowledge is only implicitly represented, and symbolic approaches, where the knowledge is explicitly represented in a symbolic form. There has been considerable recent research into the use of non-symbolic ML techniques for the learning of model transformations and other translations of software languages, for example [1,3,4,12,16,27]. These approaches are usually adapted from non-symbolic ML approaches used in the machine translation of natural languages, such as LSTM neural networks [14] enhanced with various attention mechanisms. These approaches are not suitable for our goal, since the learned translators are not represented explicitly, but only implicitly in the internal parameters of the neural net. Thus, it is difficult to formally verify the translators, or to manually adapt them. In addition, neural-net ML approaches typically require large training datasets (e.g., over 100MB) and long training times. This impairs agility and also has resource and environmental implications.

Symbolic ML approaches have been applied to model transformation by-example (MTBE) by [2] (using ILP) and by [21] (using search-based techniques). These typically use considerably smaller (i.e., KB-scale) training datasets compared to non-symbolic ML, and produce explicit rules. One disadvantage of ILP is that counter-examples of a relation to be learned need to be provided, in addition to positive examples. The approach of [21] uses only positive examples. It is able to learn individual String-to-String functions and Sequence-to-Sequence functions from small numbers (usually under 10) of examples of the function. It is a very general transformation synthesis tool, which can generate M2M transformations in multiple target languages (QVTr, QVTo, ETL, and ATL). In this paper, we adapt and extend this MTBE approach to learn functions relating software language parse trees, and hence to synthesise T2T code generators.

[5] http://www.tensorflow.org/text/tutorials/nmt_with_attention.

The approach of [21] takes as input metamodels for the source and target languages of a transformation, and an initial outline mapping of source metaclasses to target classes, expressed in the abstract model transformation language $\mathcal{TL}$ [19]. In our adaption of MTBE, we use the language grammar categories of L_1 and L_2 as the source and target metamodels. The initial mapping is defined to indicate which L_1 categories map to L_2 categories. For example, in a transformation mapping UML/OCL to Java, there could be OCL language categories $OclLambdaExpression$ and $OclConditionalExpression$ (subcategories of $OclExpression$), and a Java category $JavaExpr$, with the outline language category mappings

$$OclLambdaExpression \longmapsto JavaExpr$$
$$OclConditionalExpression \longmapsto JavaExpr$$

In practice, we use language-independent categories $ProgramExpression$, $ProgramStatement$, etc. for the target language, so that the same outline mapping can be used for different target languages.

The MTBE process also takes as input a model m containing example instances from the source and target languages, and a mapping relation $\mapsto$ defining which source and target elements correspond. For example, using text representations of elements

```
x1 : OclLambdaExpression
x1.text = "lambda x : String in x+x"

y1 : JavaExpr
y1.text = "x->(x+x)"

x1 |-> y1
```

To infer functional mappings from source data such as $OclLambdaExpression::text$ to type-compatible target data such as $JavaExpr::text$, at least two examples of each $SC \longmapsto TC$ language category correspondence must be present in the model. String-to-String mappings of several forms can be discovered by the MTBE approach of [21]: where the target data are formed by prefixing, infixing, or appending a constant string to the source data; by reversing the source data; by replacing particular characters by a fixed string, etc.[6] Similarly, functions of other datatypes can be proposed based on relatively few examples.

MTBE operates by postulating source–target functions based on the examples, and then checking if the postulated function is valid for the examples. The proposed functions are selected from a repertoire of functions which have been found to occur in practice in transformation specifications.

[6] In this respect, it is similar to Microsoft's FlashFill [11].

We use the same principle for CGBE, and postulate code generation mappings based on the idioms of "Code Generation Idioms". These mappings are then checked against the training data.

Software Language Representations

Different forms of software representation could be used for CGBE:

- Text, eg.: "sq[i] + k"
- Sequences of tokens, eg.: 'sq', '[', 'i', ']', '+', 'k'
- Abstract syntax/parse trees (ASTs):

```
(OclBinaryExpression
 (OclBasicExpression sq [ (OclBasicExpression i) ])
 +
 (OclBasicExpression k))
```

- Instance model data of a language metamodel:

```
ba0 : OclBasicExpression
ba0.type = Integer
ba0.data = "i"
ba1 : OclBasicExpression
ba1.type = Integer
ba1.data = "sq"
ba1.arrayIndex = ba0
ba2 : OclBasicExpression
ba2.type = Integer
ba2.data = "k"
be1 : OclBinaryExpression
be1.type = Integer
be1.operator = "+"
be1.left = ba1
be1.right = ba2
```

These representations are progressively richer and more detailed. Compared to text or token representations, parse trees express more detailed information about the software element, specifically its internal structure in terms of the grammar of the language. As [4] discuss, this representation therefore provides a more effective basis for ML of language mappings, compared to token sequences or raw text. For example, it is difficult to infer a String-to-String mapping between OCL expressions and Java expressions represented as raw text,[7] as the lambda expression example of the preceding section demonstrates.

Figure 4 shows the metamodel which we use for parse trees (i.e., AST terms). This metamodel is of wide applicability, not only for representation and processing of software language artefacts, but also for natural language artefacts. $ASTSymbolTerm$ represents individual symbols such as '[' in the above example. $ASTBasicTerm$

[7] Because of arbitrary text formatting differences in examples.

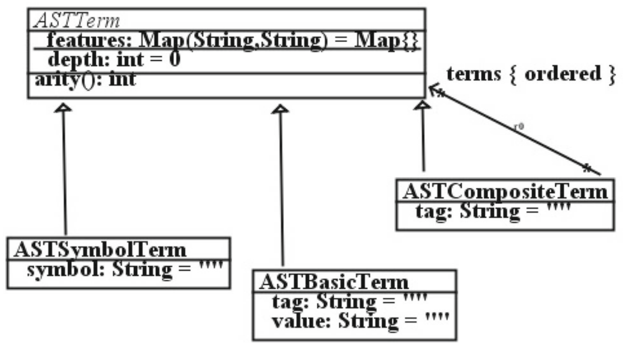


Fig. 4 Metamodel of parse trees

represents other terminal parse tree nodes. For example, `(OclBasicExpression i)`. *ASTCompositeTerm* represents non-terminal nodes, such as the root *OclBinaryExpression* node of the example. The *tag* of a basic or composite term is the identifier immediately following the initial (, in the text representation of parse trees. The *arity* of a symbol is 0, of a basic term is 1, and of a composite term is the size of *terms* (the direct subnodes of the tree node). The tag is used as the syntactic category name of the tree, when the tree is processed by a $\mathcal{CSTL}$ script—i.e., rules in the ruleset with this name are checked for their applicability to the tree. The *depth* of a term is 0 for symbol terms, 1 for basic terms, and $1 + max(terms \rightarrow collect(depth))$ for composite terms.

To utilise MTBE for CGBE, we derive a model file m from the text examples in dataset D. Each of the source and target examples ex of D are expressed as model elements mx in m, with these elements having an $ast : ASTTerm$ attribute, whose value is a parse tree (according to the appropriate language grammar) of the software element ex. For example the model element $x1$:

```
x1 : OclLambdaExpr
x1.ast = (OclUnaryExpression
          lambda x :
          (OclType String) in
          (OclBinaryExpression
             (OclBasicExpression x) +
             (OclBasicExpression x)))
```

represents $lambda\ x : String\ in\ x + x$.

Syntax Tree Mappings

There are many different possible mappings between parse trees; for example, a composite parse tree s could be mapped to a tree t with $t.terms$ being the reverse list of $s.terms$. The total number of tree-to-tree mappings is extremely high. Even restricting to sequence-to-sequence mappings gives

$$(K^n)^{K^n}$$

possible mappings between the set of sequences of length n over an alphabet of size K. However, only a subset of the possible mappings are relevant to code generation, or more generally, to software language translation. Thus, the search for mappings which are consistent with a given set of examples can be restricted to those mappings which are plausible for code generation.

From examination of different cases of code generation and language translation, we identified the following categories of relevant tree-to-tree mappings, based on the idioms of "Code Generation Idioms":

- *Structural elaboration*: the target trees contain subparts derived from corresponding source trees, and additional constant subparts/structures. For example, a transformation from expressions of the form $op()$ to $M.op()$.
- *Structural simplification*: Some subparts of the source terms are omitted from the target terms.
- *Structural rearrangement*: the target subparts are derived from subparts of corresponding source trees, but these parts occur in different orders and positions in the target than in the source. E.g., a transformation from postfix unary expressions $x \rightarrow op()$ to prefix forms $op\ x$.
- *Structural replacement* via functional mappings of symbols: a functional relation exists between source tree symbols and symbols in corresponding target trees. E.g: symbol changes for binary operators, so that $x = y$ translates to $x == y$ and $x\ mod\ y$ to $x\ \%\ y$.
- *Iterative mappings*: for variable-arity source terms, the subterms are mapped in the same order to target subterms of a variable-arity target term. For example: mapping individual statements in the statement sequence of a block statement.
- *Conditional mapping/horizontal splitting*: the mapping has alternative cases based on some property of the source trees, such as their arity or the specific symbol at a given subterm position of the trees.
- *Selection or filter mappings*: for variable-arity source terms, only those subterms satisfying a given property are mapped to subterms of a variable-arity target term. E.g: only instance attributes of a UML class E are mapped to fields of a corresponding C struct E. Other class contents are not included in the struct.

We do not address the $accumulation$ idiom. Vertical splitting and context-sensitive generation are handled by learning separate mappings for each part/case of the transformation.

We describe tree-to-tree mappings by writing pairs of schematic parse trees containing metavariables _$*$ or _i for $i : 1..99$ to represent subparts of trees which vary. A mapping

```
(stag st1 ... stn) |--> (ttag tt1 ... ttm)
```

means that source terms with tag $stag$ and n subterms of the forms denoted by $st1$, ..., stn will map to target terms of the form denoted by the RHS of the mapping. As with $\mathcal{CSTL}$, a metavariable $\_v$ occurring on the RHS denotes the translation of the source tree part bound to $\_v$.

A typical example of structural elaboration arises when mapping OCL basic expressions to Java expressions. An OCL parse tree

```
(OclBasicExpression i)
```

representing an identifier i maps to

```
(expression (primary i))
```

as a Java expression parse tree. As a mapping from the source language category $OclIdentifier$ to $JavaExpr$, this is denoted

$$(OclBasicExpression\ \_1) \longmapsto (expression\ (primary\ \_1)).$$

In other words, this translation of source trees to target trees is assumed to be valid for all $OclIdentifier$ elements with the same structure, regardless of the actual identifier in the first subterm of the source element parse tree. In general, any number of additional single nesting levels can be introduced in the target terms relative to the source.

New subterms of the target trees can be introduced when a simple OCL expression is represented by a more complex Java expression. For example, an array access $sq[i]$ in OCL becomes $sq[i-1]$ in Java, because Java arrays are 0-indexed. As parse trees, this means that

```
(OclBasicExpression
    (OclBasicExpression sq)
      [ (OclBasicExpression i) ])
```

maps to:

```
(expression (expression (primary sq)) [
  (expression (expression (primary i)) -
  (expression (primary
    (literal (integerLiteral 1)))) ) ])
```

Figure 5 shows the corresponding OCL and Java parse trees. The outlined subtrees also correspond.

Functional mappings of symbols arise when operators are represented by different symbols in the source and target languages. For example, OCL unary prefix expressions

```
(OclUnaryExpression op expr)
```

map to

```
(expression f(op) expr')
```

in Java, where $expr$ maps to $expr'$ and $f(not)$ is !, $f(-)$ is $-$, $f(+)$ is $+$.

Schematically, this can be represented as

$$(OclUnaryExpression\ \_1\ \_2) \longmapsto (expression\ f(\_1)\ \_2),$$

where f is defined by a separate rule.

More complex cases involve re-ordering the subterms of a tree; embedding the source tree as a subpart of the target tree, etc. Tree-to-tree mappings can be constructed and specified recursively.

Learning code generation rules as mappings between parse trees has the advantage that the target text produced by the rules is correct (by construction) with respect to the target grammar. This is not the case for template-based M2T code generators, where the fixed texts in templates can be arbitrary strings, and may be invalid syntax in the target language.

Strategies for Learning Tree-to-Tree Mappings

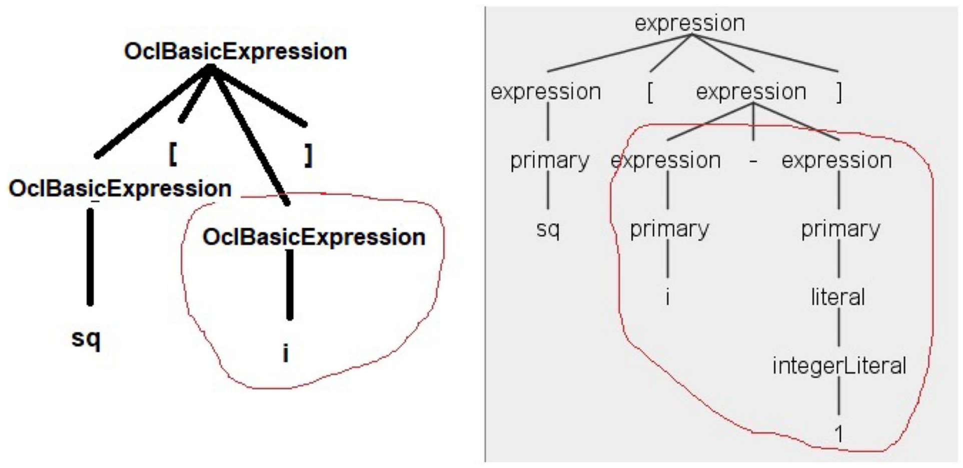
To recognise syntax tree mappings between two language L_1 and L_2, the CGBE procedure postulates and checks possible forms of mapping for each category-to-category correspondence $SC \longmapsto TC$ between source category SC of L_1 and target category TC of L_2. Given a set of corresponding examples of SC and TC, the ast values of all corresponding instances $x : SC$ and $y : TC$ are compared, i.e., for all such pairs with $x \mapsto y$, the values $x.ast$ and $y.ast$ are checked to see if a consistent relationship holds between the value pairs. At least two instances x_1, x_2 of SC with corresponding TC elements must exist for this check to be made.

The search procedure for possible mappings uses the following six main search strategies over the pairs $s = x.ast$, $t = y.ast$ of AST values of corresponding $x \mapsto y$ from $SC \longmapsto TC$:

Strategy 1: Matching constant-arity trees If all considered target terms $t = y.ast$ with tag $tag2$ have the same arity n: $(tag2\ t_1\ ...\ t_n)$ and their corresponding source terms $s = x.ast$ all have the same tag $tag1$ and arity k: $(tag1\ s_1\ ...\ s_k)$, then $s \longmapsto t$ if for each $i : 1..n$, each t_i is either:

1. A constant K for each considered t
2. A symbol equal to or a function of the s_j for a particular fixed j
3. A corresponding term of s_j for a particular fixed j, i.e.: $s_j \mapsto t_i$
4. Mapped from some s_j: $s_j \longmapsto t_i$ for a particular fixed j (recursive check using any of the strategies)

Fig. 5 Corresponding parse trees



5. Mapped from the entire source term: $s \longmapsto t_i$ (recursive check using any of the strategies).

*Strategy 2: *-arity trees, subterm correspondences* If all considered target terms $t = y.ast$ with tag $tag2$ have the same arity as their corresponding source terms $s = x.ast$: $t = (tag2\ t_1 \dots t_n)$ and $s = (tag1\ s_1 \dots s_n)$, then there is a schematic mapping $(tag1\ \_*) \longmapsto (tag2\ \_*)$ if each $s_i \longmapsto t_i$ for each s, t via a recursive check.

*Strategy 3: *-arity trees, symbol changes* As for strategy 2, except that s, t can have different arities, but the same number of non-symbol subterms, which must correspond via a mapping. Symbol subterms of s can be consistently removed or replaced by symbol subterms in t, and new symbols can be inserted. The schematic mapping in this case is $(tag1\ \_*) \longmapsto (tag2\ f(\_*))$ for a function f that performs the deletion/replacement/insertion.

Strategy 4: conditional mappings If the set st of source terms under consideration all have the same tag and arity, but different values for one subterm s_i which is always a symbol, then st can be subdivided into disjoint sets sub_k which have a specific value v_k for s_i, and the search for a mapping proceeds for each of these specialised sets and their corresponding sets of target trees, in turn.
The mappings are expressed as conditional mappings $s \longmapsto t \ \ when \ \_i = v_k$.
Other discrimination conditions could also be used to identify conditional mappings, such as different source term arities for the same tag.

Strategy 5: selection mappings As for strategy 3, except that s can have more non-symbol subterms than t. All non-symbol subterms of t must have a matching non-symbol source subterm of s.
This allows for selection or filtering of source subterms. The schematic mapping in this case is $(tag1\ \_*) \longmapsto (tag2\ f(\_*))$ for a function f that performs the subterm selection and symbol deletion/replacement/insertions.

Strategy 6: tree-to-sequence mappings In this case, the source terms have the same tag and arity, and the target terms have the same tag, but different arities. Subterms s_i of source terms s correspond to subsequences of terms of t. t terms may also have constant prefix, suffix, or insert term subsequences, P, X and I. For source terms of arity 2, the general form of the mapping could be

$$(tag1\ \_1\ \_2) \longmapsto (tag2\ P\_1I\_2X)$$

or

$$(tag1\ \_1\ \_2) \longmapsto (tag2\ P\_2I\_1X).$$

Stategy 1 can be formally expressed in OCL:

```
operation strategy1(ssq : Sequence(ASTCompositeTerm),
    tsq : Sequence(ASTCompositeTerm)) : ASTTerm
pre: ssq->size() = tsq->size() &
 ssq->size() > 1 &
 ssq->forAll( sx1, sx2 | sx1.tag = sx2.tag &
        sx1.arity() = sx2.arity() ) &
 tsq->forAll( tx1, tx2 | tx1.tag = tx2.tag &
        tx1.arity() = tx2.arity() )
activity:
 var m : int := ssq[1].arity() ;
 var n : int := tsq[1].arity() ;
 var mapping : Map(int, ASTTerm) := Map{} ;

 for tind : 1..n
 do
   var tvals : Sequence(ASTTerm) ;
   tvals := tsq->collect(terms[tind]) ;
   if tvals->forAll(tx | tx = K) where K constant
   then
     mapping[tind] := K
   else
     for sind : 1..m
     do
       var svals : Sequence(ASTTerm) ;
       svals := ssq->collect(terms[sind]) ;
       if tvals symbols,  function F(svals) of svals
       then
```

```
28            mapping[tind] := F(_sind)
29          else if i : 1..tvals->size() => svals[i]|->tvals[i
                ]
30          then
31            mapping[tind] := _sind
32          else
33            var submap : ASTTerm ;
34            submap := correspondence(svals,tvals) ;
35            if submap /= null
36            then
37              mapping[tind] := f(_sind) where f is a new
                    function
38            else
39              submap := correspondence(ssq,tvals) ;
40              if submap /= null
41              then
42                mapping[tind] := submap
43              else skip ;
44    if mapping->size() = n
45    then
46      return (tsq[1].tag mapping[1] ... mapping[n])
47    else
48      return null
```

In the fourth case (lines 33–37 above), the new function f is defined as a schematic mapping from the generalised form(s) of the $svals$ terms to the $submap$ schematic term. $correspondence(sterms, tterms)$ attempts to find a mapping from the $sterms$ to $tterms$ by each strategy successively, returning the non-null definition of the $tterms$ as a schematic term based on metavariables for each argument place in the $sterms$ data if successful, and returning $null$ if no mapping can be found. $strategy1$ is only successful if each of the target term argument places can be derived either as a constant or as a consistent mapping of some source data.

It is immediate to show that strategy 1 succeeds for a tag renaming $(stag\ x) \longmapsto (ttag\ x)$ for any ASTs x, and likewise for an embedding $x \longmapsto (ttag\ x)$.

As a more complex example of strategy 1, unary arrow operator OCL expressions such as

```
(OclUnaryExpression expr ->front ( ))
```

are consistently mapped to the Java parse trees

```
(expression (expression (primary Ocl)) .
 (methodCall front
   ( (expressionList expr') ) ) )
```

where $expr \longmapsto expr'$. The Java expression represents a call $Ocl.front(e)$ of a Java library for the OCL collection operations. Here, the first and second terms `(expression (primary Ocl))` and '.' of the target trees are constant, and the third term has four subterms. The first subterm is the method name, e.g., 'front', and this is a function of the second source term, the operation name such as '→front', of the source tree; the second subterm is the constant '('; the third subterm is mapped from the first term of the source; and the fourth subterm ')' is constant.

Generalising over all unary arrow operators, the schematic form of this mapping is

$$(OclUnaryExpression\ \_1\ \_2\ (\ )\ ) \longmapsto$$
$$(expression\ (expression\ (primary\ Ocl))\ .$$
$$(methodCall\ F(\_2)\ (\ (expressionList\ \_1)\ )\ )\ ).$$

This mapping involves embedding of the source tree as a subtree of the target (strategy 1, case 5), together with reordering of subterms (strategy 1, case 4: the nested mapping is $f : \_1 \longmapsto (expressionList\ \_1)$), plus a functional mapping F of the operator symbols (strategy 1, case 2) and constant target symbols (strategy 1, case 1).

The other strategies can be formalised in a similar manner. An example of the second strategy is the mapping of lists of arguments within compound expressions. For example, in Java, a parse tree `(expressionList ...)` representing a comma-separated list of expressions can have any number $n \geq 1$ of direct subnodes t_i.

In particular, OCL collection expressions of the form `(OclSetExpression Set { (OclElementList ...) })` will map to `(expression (expression (primary Ocl)) . (methodCall initialiseSet ( (expressionList ...) ) ) )` provided that the arities of the source and target list terms are equal for all corresponding source and target elements. Schematically, this mapping is represented as

$$(OclSetExpression\ Set$$
$$\{\ (OclElementList\ \_*)\ \})\ \longmapsto$$
$$(expression\ (expression\ (primary\ Ocl))\ .$$
$$(methodCall\ initialiseSet\ (\ (expressionList\ \_*)\ ))).$$

Strategy 3 allows the consistent replacement or removal of separator symbols and other symbols when mapping from source lists to target lists. In this case, the mapping has the form

$$(slistTag\ \_*) \longmapsto (tlistTag\ f(\_*))$$

for a suitable function f. For example, the replacement of ; by $\backslash n\backslash t$ in mapping a sequence of OCL statements to Python.

Strategy 4 enables the learning of mappings of different forms, based on the syntax of source terms. For example, binary expressions of forms $a+b$ and $a*b$ have the same AST structure in OCL, but map to different target tree structures in the C grammar [15].

Strategy 5 enables the selective mapping of source elements to target elements, e.g., attributes of a UML class are mapped to struct fields in C, but operations of the class are mapped to external functions.

Strategy 6 enables tree-to-sequence mappings to be recognised, where target terms are essentially sequences. Source tree structure is 'flattened' into sequences. An important case where this occurs is in the generation of assembly language code (Sect. 4.2)

The time complexity of the combined CGBE search strategy $correspondence(sasts, tasts)$ can be estimated as a function $\phi(N, P, Q, n, m)$ of the number N of examples, which is the size of the $sasts$ and $tasts$ sequences, and the maximum term width n (at any level) and depth P of the $tasts$ and maximum term width m and depth Q of the $sasts$

$$\phi(N, P, Q, n, m) = \Sigma_{r=1}^{6} \phi_r(N, P, Q, n, m),$$

where $\phi_r(N, P, Q, n, m)$ is the time complexity of strategy r. By examination of the above algorithm for strategy 1, it can be seen that a check for $correspondence(svals, tvals)$ is potentially made for each subterm of each target term, and for each subterm of each source term, resulting in a recursive complexity for $\phi_1(N, P, Q, n, m)$ of the order

$$N * n * m * \phi(N, P - 1, Q - 1, n, m)$$

when $P > 0$, $Q > 0$. Similar time costs apply for the other strategies, so that overall the complexity ϕ has the order

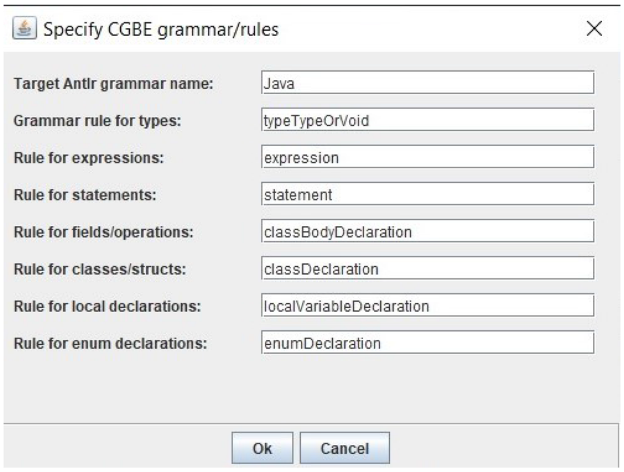
$$(N * n * m)^{max(P,Q)}.$$

This means that the depth of nesting in the example terms is the most significant time cost factor. Deep terms occur with complex grammars which have many non-terminal syntactic categories (the C and Python3 grammars are examples of this situation). In general, the simplest possible examples should be used, to reduce search costs.

Implementation of CGBE

The MTBE inference of tree-to-tree mappings described in "Strategies for Learning Tree-to-Tree Mappings" is the core step of CGBE. Prior to the CGBE process, a test dataset T of examples is prepared, disjoint from the main training dataset D and validation dataset V. The CGBE process between arbitrary source and target languages L_1 and L_2 then involves the iteration of three stages (Fig. 7):

1. Creating or modifying the training dataset D, preprocessing this to generate parse trees of the L_1 and L_2 elements, and storing these in a model file m of examples for input to the MTBE step. The user needs to specify the grammar rules to be used to parse the source and target language categories, using the interface of Fig. 6. This step also generates source and target metamodels sm, tm representing the language categories, and an initial mapping f of categories;
2. Recognition of tree-to-tree mappings between the parse trees of corresponding examples, using MTBE applied to sm, tm, m and f;
3. Conversion of the tree-to-tree mappings to $\mathcal{CSTL}$ rules.


Specify CGBE grammar/rules

Target Antlr grammar name:	Java
Grammar rule for types:	typeTypeOrVoid
Rule for expressions:	expression
Rule for statements:	statement
Rule for fields/operations:	classBodyDeclaration
Rule for classes/structs:	classDeclaration
Rule for local declarations:	localVariableDeclaration
Rule for enum declarations:	enumDeclaration

Fig. 6 Dataset pre-processing dialog

In the case that L_1 is UML/OCL, the AgileUML toolset is used to parse the UML/OCL examples and generate parse trees, and the Antlr[8] toolset with a program grammar file is used to parse the target program examples and generate program parse trees. A metamodel $mmCGBE.txt$ for the L_1 and L_2 categories sm, tm, and initial mapping $forwardCGBE.tl$ for f have already been defined.

The resulting $\mathcal{CSTL}$ script is tested against the dataset D and the test dataset T, and the steps 1 to 3 repeated until all examples in D are correctly processed, and the required level of accuracy is obtained on T.

The third step involves generating the $\mathcal{CSTL}$ textual form of the schematic source to target parse tree mappings produced by MTBE. This is carried out by a left to right traversal of the source and target schematic trees, discarding tags and returning the text content of symbol and basic terms. Spaces are inserted between successive source terms. Applications $f(\_v)$ of functions f to schematic metavariables are expressed as `_v`f` in $\mathcal{CSTL}$ notation.

For example, the schematic tree-to-tree mapping

$$(OclUnaryExpression\ \_1\ \_2\ (\,)\,) \longmapsto$$
$$(expression\ (expression\ (primary\ Ocl))\ .$$
$$(methodCall\ f(\_2)\ (\ (expressionList\ \_1)\ )\ )\ )$$

becomes the $\mathcal{CSTL}$ rule

[8] https://www.antlr.org.

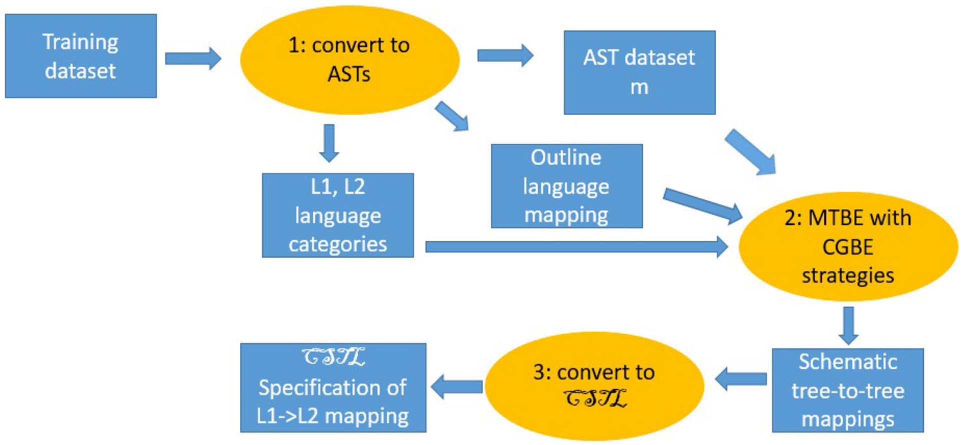

Training dataset
1: convert to ASTs
AST dataset m
L1, L2 language categories
Outline language mapping
2: MTBE with CGBE strategies
Schematic tree-to-tree mappings
3: convert to $\mathcal{CSTL}$
$\mathcal{CSTL}$ Specification of L1->L2 mapping

Fig. 7 CGBE general process steps

```
_1 _2 ( ) |-->Ocl._2‘f(_1)
```

Rules $l \longmapsto r$ are inserted into a ruleset SC for the syntactic category SC of l. They are inserted in $\sqsubset$ order, so that more specific rules occur prior to more general rules in the same category. New function symbols f introduced for functional symbol-to-symbol mappings are also represented as rulesets with the same name as f, and containing the individual functional mappings $v \longmapsto f(v)$ of f as their rules.

In the dataset,[9] we provide batch files $cgbeTrain.bat$ and $cgbeValidate$ which enable the execution of CGBE from the command line. $cgbeTrain$ is called as

```
cgbeTrain config.txt
```

where $config.txt$ holds the information about the target parser and parser rules to be used, corresponding to Fig. 6. The script performs steps 1, 2, 3 from Fig. 7.

$cgbeValidate$ is called as

```
cgbeValidate f.cstl asts.txt
```

where $f.cstl$ is the code generator, and $asts.txt$ is a list of source language ASTs, one per line. The script applies $f.cstl$ to each AST in $asts.txt$ and computes statistics of the average size and translation time of the examples.

The derived $\mathcal{CSTL}$ specification g does not necessarily translate the D dataset correctly, because some syntactic categories of L_1 may have insufficient numbers of examples in D for the derivation of correct rules. However, we can argue by induction on the depth of target terms that the derived rules of g do produce valid L_2 syntax according to the L_2 grammar.

In the case of strategy 1, target terms t of depth 1 produced by this strategy are of form $(ttag\ tt_1\ ...\ tt_n)$ where each of the tt_i are symbol terms, and hence, t must have been produced by cases 1 or 2 of the strategy. In each case, the tt_i are either constant symbols K which occurred as target terms in the example data, or are functions $F(\_j)$ of source data, with the range of F being restricted to valid target symbol values. Similar reasoning applies to other strategies.

By induction, if the result holds for target terms of depth P, then a target term $t = (ttag\ tt_1\ ...\ tt_n)$ of depth $P + 1$ produced by strategy 1 has that each tt_i is of depth at most P, and hence is a syntactically valid target term, and t itself has a correct arity and tag (since these are the same as the target examples for the syntax category TC of $ttag$). Likewise, for terms produced by other strategies. Thus, the result holds for target terms of any depth.

Evaluation

We evaluate the approach by answering the research questions of “Research Questions”. For RQ1, we identify to what extent CGBE can learn the code generation idioms of “Code Generation Idioms”. For RQ2, we evaluate the effectiveness of CGBE using it to construct code generators from UML/OCL to Java, Kotlin, C, JavaScript, and assembly language, based on text examples. We also construct a code generator from a DSL for mobile apps to SwiftUI code. For RQ3, we compare the effort required for construction and maintenance of code generators using CGBE with manual code generator construction/maintenance. We also compare the application of CGBE on the FOR2LAM case of [4] with their neural-net solution, and compare CGBE with the use of MTBE on instance models of language metamodels. For RQ4, we consider how CGBE can be generalised to apply to the learning of software language abstractions and translations.

All the datasets and code used in this paper are available at https://www.zenodo.org/record/7230107.

A tutorial video is available at youtu.be/NRY_sBlNfnY.

[9] https://www.zenodo.org/record/7230107.

Table 1 Support for code generation idioms

Idiom	Expressible in $\mathcal{CSTL}$	Learnable Using CGBE
Elaboration	$\surd$	$\surd$ (strategy 1)
Rearrangement	$\surd$	$\surd$ (strategy 1)
Simplification	$\surd$	$\surd$ (strategy 1)
Replacement	$\surd$	$\surd$ (strategy 3)
Conditional generation	$\surd$	? (strategy 4)
Context-sensitive generation	$\surd$	$\surd$ (strategy 3)
Iterative generation	$\surd$	$\surd$ (strategy 2)
Accumulation	$\surd$	$\times$
Horizontal splitting	$\surd$	$\surd$ (strategy 5)
Vertical splitting	$\surd$	$\surd$
Express types by initialisation	$\surd$	$\surd$ (strategy 1)
Perform type inference	?	?
Replace inheritance by association	$\surd$	$\times$
Pre-normalisation	$\surd$	?
Tree-to-sequence	$\surd$	$\surd$ (strategy 6)

RQ1: Coverage of Code Generation Idioms

To answer this research question, we consider to what extent $\mathcal{CSTL}$ can express the code generation idioms of "Code Generation Idioms", and to what extent these can be learnt by CGBE.

Table 1 summarises the level of support for code generation idioms. $\surd$ indicates full support, ? partial support, and $\times$ no support.

The recognition of elaboration, rearrangement, and simplification mappings is addressed by strategy 1. Replacement mappings are recognised by strategy 3. Only certain kinds of conditional generation are currently supported: based on distinctions of the arity and symbols in source terms. In general, there is no access to global/contextual information when processing a parse tree; thus, the types of the variables x, y in an expression x/y cannot generally be determined from its parse tree.

Context-sensitive generation is handled by the construction of new local functions in strategy 3; these functions are specific to the particular mapping context.

Horizontal splitting is supported by strategy 5. For vertical splitting, in cases where multiple separate output texts need to be produced from a single source, as with C header and code files, a separate CGBE process should be applied to learn each mapping as a specific $\mathcal{CSTL}$ script.

Accumulation can be expressed in $\mathcal{CSTL}$, but none of the mapping search strategies currently cover this mechanism. In principle, a strategy could be added to recognise this form of mapping.

Type inference can be partly performed in $\mathcal{CSTL}$ using the *actions* mechanism of $\mathcal{CSTL}$ to record types of variables at the points where these can be deduced [17]. There is currently no means to learn rules which involve the use of actions; however, appropriate actions could be added as part of the translation of learnt rules to $\mathcal{CSTL}$, for source elements which are declarations. For example:

```
var _1 : _2 |-->Rhs[_1,_2]<action> _1 _2
```

in the case of a learnt rule for local variable declarations.

Normalisation rules can be expressed in $\mathcal{CSTL}$, but only certain forms can be learnt. The normalisation mapping would need to be learnt as a separate script, distinct from the main translation script. Tree-to-sequence mappings can be recognised by strategy 6.

Some expressible forms of $\mathcal{CSTL}$ rules involving metafeatures such as *front*, *tail*, *reverse* applied to term sequences are not currently handled by CGBE; however, it would be possible to add strategies to recognise such mappings.

RQ2: UML/OCL to 3GL Code Generation

In this section, we evaluate the effectiveness of CGBE by applying it to learn code generators for several target languages: Java 8, ANSI C, JavaScript, Kotlin, and assembly language. The UML/OCL language used is a subset of UML class diagrams and use cases, together with OCL 2.4 and some procedural extensions to express executable behaviour. A grammar of the source language is available on the Antlr grammars github.[10]

For these code generator cases, the example dataset D is separated into four files: (i) type examples, (ii) expression examples, (iii) statement examples, and (iv) declaration examples, including operation and attribute declarations and declarations of complete classes. Table 2 shows the size of the training dataset in terms of number of examples, and the number of generated rules and the accuracy of the synthesised code generator, in terms of the proportion of validation cases (57 new examples) which were correctly translated by the generator.

It is noticeable that the training time for C is significantly higher than for the Java or Kotlin code generators, due to the high complexity of the C grammar and the large structural difference between UML/OCL and C; however, the time remains practicable.

Both Java and C have explicit typing for variables. To test the hypothesis that CGBE can be used to learn the "Express types by initialisation" code generation idiom ("Code Generation Idioms"), we use CGBE to learn the mapping from UML/OCL to JavaScript, which has implicit typing. The

[10] https://github.com/antlr/grammars-v4/tree/master/ocl.

Table 2 Evaluation of CGBE on UML/OCL to program code generation cases

Target language	Training size (#examples)	Training time (s)	Generator size (LOC)	Accuracy
Java	167	87.3	211	0.95
Kotlin	227	357	168	0.98
JavaScript	168	3.8	120	0.91
C	224	954	165	0.98
Assm	282	10	102	0.82

Table 3 Code generator efficiency: translation time per AST

Code generator	Average AST size				
	5	10	20	40	80
UML2C	4	10.4	29	37	84
UML2Java	3.35	7.9	25.5	40	76
UML2Kotlin	3.55	7.5	23	41	91
UML2JS	3.6	10.3	29	47.3	139
UML2Assm	2.7	7.5	31.3	50.5	112

accuracy of this translation is lower than for Java and C, but is still at a high level.

We also investigated if CGBE can be used to learn a mapping from UML/OCL to assembly language. We adopt an extension of the generic assembly language used in [10]. This has the usual assembly language facilities of reserving memory space, moving data between registers, an accumulator and main memory, and instructions for branching, subroutine calls, and arithmetic computations. We wrote a parser in Antlr for this language, which we call *Assm*. In contrast to the preceding cases of generating 3GL code in Java, Kotlin, C, and JavaScript, the structure of source and target code in this case is radically different. The target code is a sequence of instructions and labels, and hence, many of the mappings to be learnt are based on sequence composition functions such as prefixing and suffixing of sequences. For example, computation of a logical negation can be performed by a sequence of instructions evaluating the argument, followed by instructions to negate the result (interchange the values 0 and 1). Thus, $not(p)$ for variable p is

```
LOAD p
SUB 1
ABS
```

The general $\mathcal{CSTL}$ rule to be learnt for *not* is

```
OclUnaryExpression::
not _1 |-->_1 \n SUB 1 \n ABS
```

The accuracy of the generator produced for Assm is significantly lower than for the cases of 3GL targets, due to the wide syntactic and semantic distance between UML/OCL and assembly code. In particular, a pre-normalisation step on expressions is needed, and this cannot be learnt. Interpretations of general collection expressions, loops over collections, and general operation definitions and calls cannot be given. Nonetheless, some meaningful rules, such as the above case of *not*, can be successfully learnt from examples.

RQ2: Efficiency of Synthesised Code Generators

We evaluated the synthesised Java, C, Kotlin, JavaScript, and Assm code generators by executing them on datasets of UML/OCL ASTs with different average sizes (number of tokens in an AST). The average time of three executions is taken. Table 3 shows the results; all times are in ms. These results are comparable to the performance of manually constructed $\mathcal{CSTL}$ code generators, and show an approximately linear growth in execution time with respect to input data size.

The results are shown graphically in Fig. 8.

RQ2: Code Generation from DSLs

To demonstrate the application of CGBE for learning DSL to code mappings, we implemented a translation from a simple DSL for defining mobile UI screens, to the SwiftUI platform. The source language has constructs for buttons, text fields, selection lists, etc., together with styling elements similar to cascading style sheets (CSS). An example UI screen description in this language is

```
view
 textView tv1 { label "Weight" }
 textView tv2 { label "Height" }
 textView tv3 { label "Person" }
```

One of the leading mobile implementation platforms is iOS, which supports the SwiftUI language for defining mobile screens. The equivalent SwiftUI code for the above DSL specification is

```
VStack(alignment: .leading, spacing: 20) {
 Text("Weight")
 Text("Height")
 Text("Person")
}
```

To learn the translation, we created 46 paired examples of DSL and SwiftUI, parsed these into ASTs using the respective Antlr parsers (MobileDSL.g4 and Swift5.g4), and applied the AgileUML toolset option ‘LTBE from ASTs’.

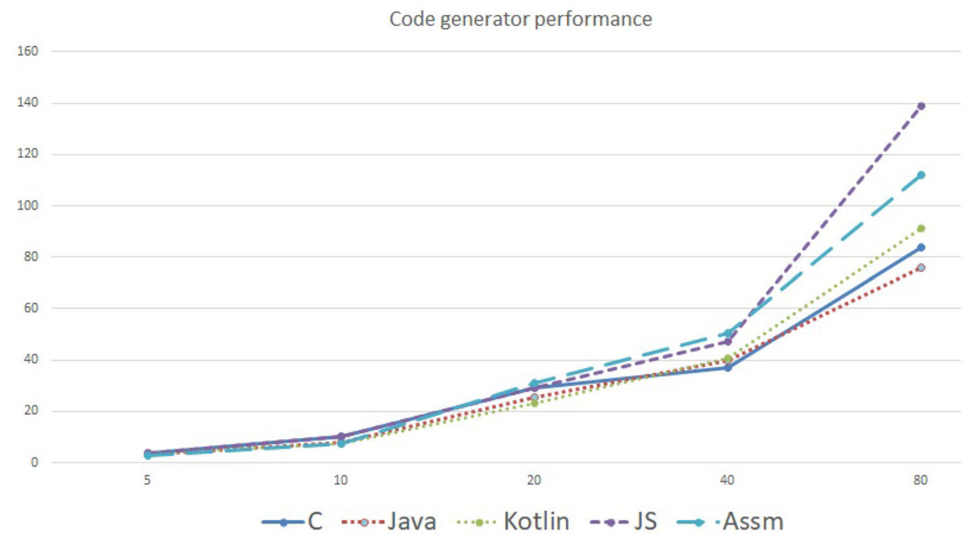


Fig. 8 Code generator performance

The training time was 5.3 s, and the generated CSTL contained 81 LOC and scored 100% accuracy on an independent validation set.

RQ3: Benefits of CGBE

To answer this question, we compare the development effort required to apply CGBE for the example code generators of Sect. 4.2 to the effort required for manual development of similar code generators. We also consider the skills and technical resources required for different code generator construction approaches.

Construction of a code generator typically involves three kinds of activity:

1. Identification of an appropriate representation of the source language in the target language, and performing tests to validate this choice;
2. Defining a support library of additional types and operations in the target language (e.g., to implement OCL collection operators);
3. Defining a code generator to implement the chosen representation/translation approach, and testing this generator.

Only the final step involves coding the generator in a specific transformation language (or in a 3GL), and we only consider this aspect of code generation construction in the following.

Table 4 compares the person-hours expended in code generator implementation/testing for the CGBE approach for code generation with that required for manually coded $\mathcal{CSTL}$ generators. All generators were constructed by the first author (expert in MDE) except for UML2Kotlin by the second author (a Ph.D. student with 1 year's experience of MDE). The implementation and testing effort is significantly reduced by the use of CGBE, because the extent of $\mathcal{CSTL}$ coding is much lower, or is even eliminated if an entirely correct script can be learnt from examples. The average time for development of a generator using CGBE is 19 person-hours, about a third of the average cost of manual coding of generators (53 h). In this respect, it is a 'low code' programming-by-example (PBE) approach requiring no explicit programming.

Table 4 Effort of manual/CGBE code generators

Code generator	Approach	Effort (person-hours)
UML2Java	$\mathcal{CSTL}$ (CGBE)	18
UML2Kotlin	$\mathcal{CSTL}$ (CGBE)	19
UML2C	$\mathcal{CSTL}$ (CGBE)	22
UML2JavaScript	$\mathcal{CSTL}$ (CGBE)	16
UML2Assm	$\mathcal{CSTL}$ (CGBE)	20
UML2Java8	$\mathcal{CSTL}$ (manual)	44
UML2Swift	$\mathcal{CSTL}$ (manual)	56
UML2Go	$\mathcal{CSTL}$ (manual)	58

Instead of explicitly coding the generator in $\mathcal{CSTL}$, a user of CGBE iteratively defines a dataset of examples and applies the CGBE procedure until it produces a generator which is correct wrt the example dataset, and which has the required level of accuracy and completeness on the test and validation datasets. The approach does not ensure 100% accuracy; however, it reduces the developer's workload by synthesising the large majority of the code generator script from examples. Testing and validation are supported by a toolset option which automatically applies a given script to a set of AST examples. This is also available as the $cgbeValidate$ batch file.

Table 5 compares CGBE with other code generation approaches with regard to the generator structure and development knowledge needed.

Table 5 Code generator structure and development

Approach	Generator structure	Skills and knowledge needed	Needs compilation
M2M	Based on L_1 metamodel	M2M language	√
		L_1 metamodel	
		L_2 metamodel	
		L_2 syntax	
M2T	Based on L_2 syntax	M2T language	√
		L_1 metamodel	
		L_2 syntax	
T2T (manual)	Based on L_1 grammar	T2T language	×
		L_1 grammar	
		L_2 syntax	
T2T (CGBE)	Based on L_1 grammar	L_1 syntax	×
		L_2 syntax	

Table 6 Relative time cost of maintenance changes

Maintenance action	Manually coded M2T (uml2py)	Manually coded T2T (UML2Java8)	CGBE T2T *(UML2Java)*
Introduce <>= operator for OCL	30 min	10 min	10 min
Introduce try-catch-finally Statements in OCL	7 h	1 h	30 min

It can be seen that CGBE has the lowest requirements on developer expertise. In terms of technological requirements, an Antlr parser for the target language is required, in addition to AgileUML. Both Antlr and AgileUML are lightweight tools, with no further technology dependencies except for Java. This is in contrast to tools such as EGL and Acceleo, which require significant technology stacks.

The effort required to maintain code generators can be substantial. Table 6 shows two maintenance examples and the relative costs in terms of developer time of performing these on manually coded M2T and T2T generators, and using CGBE.

RQ3: Comparison with [4]

The paper [4] defines a neural-net ML approach for learning program translators from examples. One case they use to evaluate their approach is a mapping from an imperative language, FOR, to a functional language, LAM. This could be considered as an example of program abstraction.

We applied our CGBE approach to this case, learning a $\mathcal{CSTL}$ script *for2lam.cstl* of 28 lines from 31 examples. The rules are non-trivial; for example, there is the following rule to abstract a *for* loop to a recursive function:

```
For::
for _2 = _4 ; _6 ; _8 do _10 endfor  |-->
```

Table 7 CGBE of FOR2LAM compared to [4]

	Training dataset	Model size	Training time	Accuracy
CGBE	31	1KB	1.9s	100% S
				100% L
NN	100,000	9.8MB	8+ h	99.99% S
				99.6% L

```
letrec func _2 = if _6 then let _ = _10
   in func _8 else ( ) in func _4
```

This rule was learnt from four examples.

We then tested the script with 1000 test cases, divided into 500 small cases S (average 20 tokens) and 500 large cases L (average 50 tokens) as in [4]. Table 7 summarises the results and compares these to the results of [4] for this case, where the time figures relate to execution on the same hardware, a standard Windows 10 laptop without GPU.

Unlike the neural-net approach of [4], there is no deterioration of accuracy in our approach with larger inputs, because a precise and correct algorithm has been learnt. The execution time for translation grows linearly with input size (24 ms per example for S examples, 50 ms per example for L examples), whereas the NN model has less consistent time

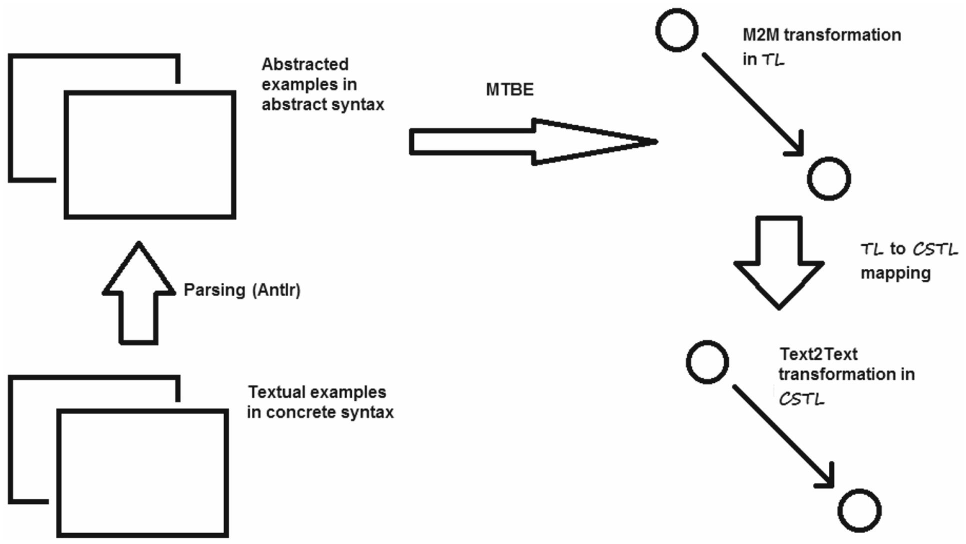

Abstracted examples in abstract syntax
MTBE
M2M transformation in $\mathcal{TL}$
$\mathcal{TL}$ to $\mathcal{CSTL}$ mapping
Parsing (Antlr)
Text2Text transformation in $\mathcal{CSTL}$
Textual examples in concrete syntax

Fig. 9 Code generator synthesis via MTBE between metamodels

performance (360 ms per example for S examples, over 2 s per example for L examples).

RQ3: Comparison with MTBE Between Language Metamodels

Instead of learning tree-to-tree mappings using MTBE, an alternative would be to use the MTBE approach of [21] to learn a M2M transformation between metamodels of the source and target languages, using example source and target instance models of the metamodels, instead of example ASTs. The synthesised abstract syntax rules in $\mathcal{TL}$ would then be translated down to the concrete syntax level (Fig. 9), or used as a M2M transformation.

We implemented these steps as follows:

1. Abstract syntax models are generated from parse trees of corresponding source and target text examples. The principal difference between such models and tree values is that they have a richer graph-like structure, with elements possessing attribute values and with cross-referencing between elements.
2. MTBE at the abstract syntax level can then be applied to the abstract syntax models as described in [21].
3. The abstract syntax rules in $\mathcal{TL}$ can then be converted to $\mathcal{CSTL}$ rules. Alternatively, the $\mathcal{TL}$ could be converted to an M2M transformation language such as QVTo, to produce an M2M code generator.

We applied this idea to learning parts of the UML to Java transformation, using the Java metamodel of [9] as the target metamodel.

In Table 8, we give the results of using metamodel-based MTBE for this case.

Table 8 MTBE transformation synthesis for UML2Java

Language part	MTBE accuracy, time, model size
Types	1.0,17s, 22 objects
Expressions (part)	1.0, 432ms, 24 objects
Expressions (full)	0.91, 21.5s, 66 objects
Statements	1.0, 5.3s, 74 objects

It can be seen that the time for training and the number of examples needed are comparable to the AST-based approach for the UML to Java translation (Table 2). Unlike CGBE based on tree-to-tree mappings, the approach using general metamodels can produce M2M transformation rules which utilise full semantic information about the source examples, e.g., detailed type information. Hence, the accuracy can be higher. However, the approach requires (i) that appropriate language metamodels are available; (ii) the definition of a mapping from example texts or graphical models to instance models of these metamodels; (iii) the definition of a general mapping from $\mathcal{TL}$ to $\mathcal{CSTL}$ (or the use of an M2M transformation as a code generator).

For (i), there are metamodels available for several leading 3GLs; [11] however, often, these have been developed for specific MDE projects, and lack the comprehensive coverage and wide usage of published Antlr grammars and parsers. The translation (ii) requires considerable coding effort, which is specific to a given grammar and metamodel. In a sense, the AST metamodel of Fig. 4 acts as a 'universal' metamodel for software languages, thus avoiding the need to define customised mappings from specific language source code to a specific language metamodel. The translation (iii) from $\mathcal{TL}$

[11] http://www.eclipse.org/MoDisco.

to $\mathcal{CSTL}$ can only be carried out for restricted forms of $\mathcal{TL}$ rules. Use of a M2M transformation has disadvantages in terms of usability, as discussed in Sect. 1, and requires an additional M2T transformation to produce target program text.

Thus, overall we consider that CGBE using tree-to-tree learning is more generally applicable and more usable for practitioners than an approach using metamodels.

RQ4: Generalisation to Abstraction and Translation Transformations

Abstraction transformations map a software language at a lower level of abstraction to one at a higher level of abstraction, e.g., C to UML/OCL [18]. Translation transformations map languages at the same abstraction level, e.g., Java to Python [16].

The idioms of "Code Generation Idioms" also apply to such transformations; however, there are also specialised idioms for abstraction:

- *Merging of content:* Contents from different parts of the source artefact are combined in a single element in the target. For example, operation declarations and implementations in C are merged into a single operation definition in UML.
- *Merging of categories:* An abstract language may merge categories such as expressions and statements into a single expression category. E.g., the FOR2LAM transformation involves this kind of merging.
- *Sequence-to-tree mapping:* The inverse of tree-to-sequence mapping, this form of correspondence maps a linear subsequence of subterms of a source term to a single term in the target

  $$(stag\ TS1\ TS2) \longmapsto (ttag\ t1\ t2),$$

  where $TS1$ and $TS2$ are sequences of subterms of the source term.

The CGBE strategies described in "Strategies for Learning Tree-to-Tree Mappings" are oriented towards the situation of *refinement* of a source language to a target language, such as the code generation case. In this situation, the source syntax trees are typically less elaborately structured than the target trees. However, the strategies can also be used for translation and abstraction cases, as the FOR2LAM case of Sect. 4.6 demonstrates. We have to take into consideration that increasing the number of CGBE strategies will also impact the performance of CGBE; thus, we do not currently support additional strategies for the above abstraction idioms.

Table 9 Evaluation of CGBE on abstraction/translation cases

Translation/ abstraction	Training size (#examples)	Training time (s)	$\mathcal{CSTL}$ size (LOC)	Accuracy
FOR2LAM	31	1.9	28	1.0
JS2OCL	28	3	25	1.0
JS2Python3	28	0.5	29	1.0

Abstractions from a 3GL to UML/OCL can be partly derived as the inverses of code generation transformations. If a learnt code generator g has rules

```
LHS   |-->RHS
```

in the category mapping $SC \longmapsto TC$, then its inverse $g^{\sim}$ has rules

```
RHS   |-->LHS
```

in category mapping $TC \longmapsto SC$. The mapping g should be injective, and only restricted forms of rule condition are possible, such as tests on metavariable type.

An example is the mapping of equality relations from OCL to Java

```
_1 = _2 |-->_1.equals(_2)<when> _1 String
_1 = _2 |-->_1.equals(_2)<when> _1 Sequence
_1 = _2 |-->_1.equals(_2)<when> _1 Set
_1 = _2 |-->_1 == _2<when> _1 Object
_1 = _2 |-->_1 == _2<when> _1 Real
_1 = _2 |-->_1 == _2<when> _1 Integer
_1 = _2 |-->_1 == _2<when> _1 Boolean
```

Clearly, arrays are not in the range of this mapping, and the case of == between strings and between collections does not occur on the Java side. Thus, these cases need to be added for the inverse abstraction mapping.

An example abstraction from JavaScript expressions to OCL is provided in the dataset. This has a training set of 28 paired expression examples, and CGBE produced a correct script of 25 LOC in 3s.

Language translation may involve a combination of abstraction and refinement. An example translation from JavaScript expressions to Python3 expressions is given in the dataset. This mapping was learnt by CGBE from 28 paired expression examples, producing a correct $\mathcal{CSTL}$ translation mapping of size 29 LOC in 0.5s.

We provide AgileUML tool options "LTBE from texts" and "LTBE from ASTs" to support the use of our CGBE techniques for abstraction and translation. Table 9 summarises the results for abstraction and translation cases.

Limitations of the Approach

The basic requirement on the training dataset D for CGBE is that it provides at least two examples for each source syntactic category. To learn a context-sensitive mapping f of symbols, one example of use of each different source symbol in the context must be provided. E.g., to learn the mapping of unary prefix operators

```
func::
- |-->-
+ |-->+
not |-->!
```

an example of each case must be included in D:

```
-x      -x
+1      +1
not p   !p
```

In addition to these numeric constraints on D, the source and target examples must be syntactically correct for their languages and category. There should also be sufficient diversity in the examples, so that rules of sufficient generality to process new examples are induced. Specifically, not all examples of the same category should have the same element in any one argument place. Thus

```
-x      -x
+x      +x
not x   !x
```

would be an insufficiently diverse example set for prefix unary expressions: the rule

```
_1 x |-->_1`func x
```

would be produced, instead of

```
_1 _2 |-->_1`func _2
```

In addition, in a set of examples of the same category, no (non-constant) argument place should be a function of another argument place. Thus, in learning the mapping of a binary expression, examples such as

```
x mod y         x % y
x mod z         x % z
w mod z         w % z
```

should be used.

One current limitation is that training model data should be organised, so that examples of composite elements only use subelements which have been previously introduced. Thus, an example

```
if b then 0 else 1 endif
```

of a conditional expression should use expressions (here, basic expressions b, 0, 1) which have been included as examples in their own category.

Currently, the approach is oriented to the production of code generators from UML/OCL to 3GLs. It is particularly designed to work with target languages supported by Antlr Version 4 parsers. Antlr parsers are available for over 200 different software languages, so this is not a strong. restriction[12] To apply CGBE for target language T, the user needs to identify the T grammar rules that correspond to the general language categories of expressions, statements, etc. (Fig. 6). The metamodel *mmCGBE.txt* of syntactic categories, and the outline mapping *forwardCGBE.tl* of syntactic categories may also need to be modified. Errors may be present in the Antlr parsers (we discovered that each of the Java, C, VisualBasic6, and JavaScript parsers contain at least one significant grammar rule error), which may result in the learning of erroneous translation rules.

As noted above in Sect. 4.8, some strategies for abstraction and translation are explicitly not supported by CGBE at present, for efficiency reasons. This means that some mappings cannot be learnt by CGBE; in particular, sequence-to-tree mappings and accumulation mappings are not learnable.

Related Work

Our work is related to model transformation by-example (MTBE) approaches such as [2,3], to programming-by-example (PBE) [6,11], and to program translation work utilising machine learning [4,12,16]. The approach of [2] uses inductive logic programming (ILP) to learn model transformation rules. ILP appears to be appropriate for the task of learning tree-to-tree mappings, since trees are naturally representable as Prolog terms. Similar to CGBE, ILP postulates and checks possible facts and rules to explain the example data. However, it is a general-purpose learning system, whereas CGBE is restricted to learning code generation mappings. Thus, CGBE can utilise a more focussed search process. In contrast to our approach, ILP requires the user to manually provide counter-examples for invalid mappings. In experiments with ILP, we found that it was unable to discover complex tree mappings which our approach could recognise. PBE also typically requires negative examples. As is pointed out by [11], successful cases of PBE usually require a strong restriction on the search space for candidate solutions. In our case, the search space is restricted to those tree-to-tree functions which we have found to be widely used in code generation.

Neural network-based ML approaches using training on bilingual datasets have achieved successful results for MTBE [3] and software language translation [4] tasks. These adapt established machine translation approaches for natural languages to software languages. In contrast to our approach, these techniques do not produce explicit transformation

[12] https://github.com/antlr/grammars-v4.

or translation rules, and they also require large training datasets of corresponding examples. Neural-net approaches encounter difficulties with novel inputs (not seen in the training sets) due to the *dictionary problem* [3], and tend to become less accurate as inputs become larger in size [4]. In contrast, because the rules we learn are symbolic, they can be independent of the specific input data (such as variable names) and apply regardless of the size of these data. They can in principle be 100% accurate.

The Transcoder language translation approach developed by Facebook [16,30] uses monolingual training datasets. The approach is based on recognising common aspects of different languages, e.g., common loop and conditional program keywords and structures. As with the bilingual neural-net approaches, large datasets are necessary, and only implicit representations of learnt language mappings are produced. This approach may not be applicable in cases where the source and target languages have a large syntactic distance, such as a 3GL and assembly language.

Another related AI-based approach is intelligent code completion, as supported by tools such as Github Copilot,[13] AlphaCode[14] and Polycoder.[15] These use large datasets of programming solutions to identify completions of partially coded functions. The quality of the resulting code depends critically on the quality of code in the base dataset, and this can be of highly variable quality (e.g., code taken from Github repositories).

In our view, neural-net machine learning approaches are suitable for situations where precise rules do not exist or cannot be identified (for example, translation between natural languages). However, for discovering precise translations, such as code generation mappings, a symbolic ML approach seems more appropriate.

Our approach utilises the MTBE approach of [21], by representing collections of language categories as simple metamodels, and by mapping paired text examples of source and target languages to instance models of these metamodels. We have extended and generalised the MTBE approach with the capability to recognise tree-to-tree and tree-to-sequence mappings, and the facility to translate the resulting mappings to $\mathcal{CSTL}$. An earlier version of the present paper is appeared in [25]. The present paper substantially extends and develops the CGBE approach introduced in [25], applies it to additional code generation cases, and provides a detailed rationale for the approach in terms of the processing requirements and idioms of code generation.

[13] https://copilot.github.com/.

[14] https://alphacode.deepmind.com/.

[15] https://nixsolutions-ai.com/polycoderopensourceai/.

Threats to Validity

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

Threats to Internal Validity

Instrumental Bias

This concerns the consistency of measures over the course of the experiments. To ensure consistency, all analysis and measurement was carried out in the same manner by a single individual (the first author) on all cases. The comparison with the results of [4] used the same approach and a similar set of test cases to the evaluation in [4]. Analysis and measurement for the results of Table 2 were repeated to ensure the consistency of the results.

Selection Bias

We chose UML to Java, Kotlin, and C translations as typical of the code generation tasks faced by practitioners. Java and C are common targets for MDE code generation, e.g., [7–9,23]. DSLs have been widely used for mobile app specification [5,13,31], and the synthesis of SwiftUI is a typical task in this domain.

Threats to External Validity

Generalisation to Different Samples

Code generation of Java, Kotlin, and C involves many of the problems encountered in code generation of other object-oriented and procedural languages. However, these languages use explicit typing. As a representative of languages with implicit typing we also considered code generation of JavaScript.

Threats to Construct Validity

Inexact Characterisation of Constructs

Our concepts of AST mappings and code generation are aligned to widely used concepts in language engineering. We have given a precise characterisation of ASTs via a metamodel (Fig. 4), defined $\mathcal{CSTL}$ via a metamodel (Fig. 3), and given a precise characterisation of CGBE (Sect. 3). The principal CGBE algorithm has been precisely specified, and analysed to justify its correctness ("Strategies for Learning Tree-to-Tree Mappings").

Threats to Content Validity

Relevance

We have shown that the CGBE approach can learn the majority of idioms commonly used in code generators (Sect. 4.1). We have also shown that it is effective for learning common code generation tasks, and can be applied for some translation and abstraction tasks (Sect. 4.8).

Representativeness

The 3GL code generation tasks we have examined (generation of Java, Kotlin, C, and JavaScript) are representative of typical code generation tasks for 3GL targets in MDE. We also considered the case of assembly code generation. The mobile DSL to SwiftUI task of Sect. 4.4 is also representative of typical code generation tasks from DSL specifications.

Threats to Conclusion Validity

We used the proportion p of correct translations of an independent validation set to assess the accuracy of synthesised code generators. This measure is widely used in machine learning. The measure $1 - p$ also expresses the proportion of additional effort needed to correct the code generators.

Conclusions and Future Work

We have described a process for synthesising code generator transformations from datasets of text examples. The approach uses symbolic machine learning to produce explicit specifications of the code generators.

We have shown that this approach can produce correct and effective code generators, with a significant reduction in effort compared to manual construction of code generators. We also showed that it can offer reduced training times and improved accuracy compared with a neural net-based approach to learning program translations.

Future work could include extending CGBE with a wider repertoire of search strategies, and by the combination of other forms of mapping with tree-to-tree mappings, e.g., to enable string-to-string mappings embedded in a tree-to-tree mapping to be discovered.

Optimisation of code generators so that they produce code satisfying various quality criteria is another important area of future work. Quality criteria could be low code size, complexity, or redundancy. CGBE strategies would need to be designed to favour the production of code generation rules which result in generated code satisfying the criteria.

An important area of application is program translation, where there is active research both in industry and academia to address legacy code problems via automated code migration. We will apply CGBE to the discovery of abstraction mappings from 3GLs to UML/OCL, as part of a program translation process.

Declarations

Conflict of interest Q. Xue was funded by the King's China Scholarship award. K. Lano declares that he has no conflict of interest. Q. Xue declares that she has no conflict of interest. This article does not contain any studies with human participants or animals performed by any of the authors.

References

1. Aggarwal K, Salameh M, Hindle A. Using machine translation for converting Python 2 to Python 3 code. PeerJ Preprints. 2015.
2. Balogh Z, Varro D. Model transformation by example using inductive logic programming. SoSyM. 2009;8:347–64.
3. Burgueno L, Cabot J, Gerard S. An LSTM-based neural network architecture for model transformations. In: MODELS '19. pp. 294–9. 2019.
4. Chen X, Liu C, Song D. Tree-to-tree neural networks for program translation. In: 32nd conference on neural information processing systems (NIPS 2018). 2018.
5. Derakhshandi M, Kolahdouz-Rahimi S, Troya J, Lano K. A model-driven framework for developing android-based classic multiplayer 2D board games. Autom Softw Eng. 2021;28(2):1–57.
6. Desai A, Gulwani S, Hingorani V, Jain N, Karkare A, Marron M, Sailesh R, Roy S. Program synthesis using natural language. ICSE. 2016;2016:345–56.
7. Eclipse project, 2020. Eclipse UML2Java code generator. https://git.eclipse.org/c/umlgen/. Accessed 18 Aug 2020.
8. Funk M, Nysen A, Lichter H. From UML to ANSI-C: an eclipse-based code generation framework, RWTH. 2007.
9. Greiner S, Buchmann T, Westfechtel B. Bidirectional transformations with QVT-R: a case study in round-trip engineering UML class models and Java source code. In: Modelsward 2016. INSTICC, SCITEPRESS; 2016. pp. 15–27.
10. Gries D. Compiler construction for digital computers. Wiley, New York; 1971.
11. Gulwani S. Programming by Example, Microsoft Corp. 2016.
12. Guo D, et al. GraphCodeBERT: pre-training code representations with dataflow. In: ICLR 2021, 2021.
13. Heitkotter H, Majchrzak T, Kuchen H. Cross-platform MDD of mobile applications with MD^2. In: SAC 2013. ACM Press; 2013. pp. 526–33.

14. Hochreiter S, Schmidhuber J. Long short-term memory. Neural Comput. 1997;9:1735–80.
15. Kernighan B, Ritchie D. The C programming language. 2nd edn. Prentice Hall; 1988.
16. Lachaux M-A, Roziere B, Chanussot L, Lample G. Unsupervised translation of programming languages. 2020. arXiv:2006.03511v3.
17. Lano K. Using the code generator language CSTL. 2022. https://agilemde.co.uk/cgrules.pdf.
18. Lano K. Program translation using Model-driven engineering. In: ICSE 2022 companion Proceedings, 2022. pp. 362–63.
19. Lano K, Fang S, Kolahdouz-Rahimi S. TL: an abstract specification language for bidirectional transformations. In: MoDeVVa 2020, MODELS 2020. 2020. pp. article 77, 1–10.
20. Lano K, Kolahdouz-Rahimi S, Alwakeel L. Synthesis of mobile applications using AgileUML. ISEC. 2021;2021:1–10.
21. Lano K, Kolahdouz-Rahimi S, Fang S. Model transformation development using automated requirements analysis, metamodel matching and transformation by-example. ACM TOSEM. 2021;31(2):1–71.
22. Lano K, Xue Q. Lightweight software language processing using antlr and cgtl. In: Proceedings of the 11th international conference on model-driven engineering and software development (MODELSWARD), 2023.
23. Lano K, Yassipour-Tehrani S, Alfraihi H, Kolahdouz-Rahimi S. Translating from UML-RSDS OCL to ANSI C. In: OCL 2017, STAF 2017. 2017. pp. 317–30.
24. Lano K, Xue Q. Agile specification of code generators for model-driven engineering. In: 2020 15th international conference on software engineering advances (ICSEA). 2020. pp. 9–15.
25. Lano K, Xue Q. Code generation by example. In: Proceedings of the 10th international conference on model-driven engineering and software development (MODELSWARD). 2022. pp. 84–92.
26. Muggleton S, de Raedt L. Inductive logic programming: theory and methods. J Logic Programm. 1994;19–20:629–79.
27. Nguyen AT, Nguyen TT, Nguyen TN. 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. IEEE Press; 2015. pp. 585–96.
28. OMG. Object constraint language (OCL) 2.4 specification. 2014.
29. Quinlan J. C4.5: programs for machine learning. Morgan Kaufmann; 1993.
30. Roziere B, Zhang J, Charton F, Harman M, Synnaeve G, Lample G. Leveraging automated unit tests for unsupervised code translation, 2021. CoRR, vol. abs/2110.06773.
31. Vaupel S, Taentzer G, Gerlach R, Guckert M. Model-driven development of mobile applications. Sosym. 2018;17(1):35–63.
32. Whittle J, Hutchinson J, Rouncefield M, Burden H, Heldal R. A taxonomy of tool-related issues affecting the adoption of MDE. Sosym. 2017;16:313–31.