Using the code generator language $\mathcal{CSTL}$

K. Lano

December 22, 2023

1 Introduction

$\mathcal{CSTL}$ is a simple text-based language for defining code generators and code abstractors for the AgileUML toolset. This enables users to define and add their own code generators/abstractors to the tool, and to modify existing code generators/abstractors. It is a scripting language executed within AgileUML, and does not require compilation.

Section 2 describes the use of $\mathcal{CSTL}$ for code generation, and Section 3 its use for program abstraction. Section 4 describes how symbolic machine learning can be used to produce $\mathcal{CSTL}$ scripts from examples.

2 Code generators

Three generators are currently available in the *cg* directory:

1. For Java 8, in files *cgJava*8*.cstl*, *cgmain.cstl* and *cginterface.cstl*. The generated code uses the OCL library *Ocl.Java*.

2. For Swift 5, in files *cgSwift.cstl*, *cgswiftmain.cstl* and *cgprotocol.cstl*. OCL library *Ocl.swift*.

3. For Go 15.1, in files *cgGo.cstl*, etc, with OCL library *ocl.go*.

The Java 8 rules are used for Android app generation, and the Swift 5 rules for iOS app generation (AgileUML tool *Build* menu options for app generation from the current model). The Go rules are executed via the *Build* menu option "Generate Go".

Usually the simplest way to create a new code generator is to copy and adapt an existing generator for a similar language: the Go translator was developed from the Swift generator in this way.

It is also possible to use the *antlr*2*cstl* script to generate an outline CSTL script for an ANTLR grammar *L.g*4: use the ANTLRv4 parser for ANTLR to produce an AST for *L.g*4 (parser rules only) in *output*/*ast.txt*, and invoke *antlr*2*cstl* to generate an outline script in *cstlFromAntlr.cstl*.

The *Build* menu option "Use CSTL specification" applies a selected *.cstl* file to the current model, to enable experimentation with alternative code generators.

Rules in $\mathcal{CSTL}$ have the form:

```
Source syntax |--> Target syntax <when> Condition
```

The left side of the rule is some text syntax of a UML class, expression or statement, where KM3 text syntax is used for class declarations, and OCL syntax for expressions. The statement syntax of AgileUML is used for statements. The right side of the rule is written in the syntax of the target language. The *Condition* is expressed in terms of the source language syntax categories and stereotypes.

Rules are grouped based on the source syntax category: types, expressions, etc.

As an example, some rules from UML to Java could be written as:

```
Package::
package _1 { types: _2 classes: _3 usecases: _4 }  |-->package _1;\n\n_2\n\n_3

Class::
abstract class _1 { _2 }   |-->abstract class _1\n{\n_2\n}\n\n

class _1 extends _2 { _3 }   |-->class _1 extends _2 {\n_3\n}\n\n

class _1 { _2 }  |-->class _1\n{\n_2\n}\n\n

Attribute::
attribute _1 : _2; |-->  _2 _1;\n

reference _1 : _2; |-->  _2 _1 = new _2();\n


Type::
Set(_1) |-->HashSet<_1>
Sequence(_1) |-->ArrayList<_1>


Operation::
operation _1(_2) : _3
pre: _4 post _5 activity: _6;  |-->  public _3 _1(_2)\n { _6 }\n
```

The $_i$ for positive integer i from 1 to 99 denote variables which hold concrete syntax fragments. They are termed 'metavariables'. On the LHS of a rule, $_i$ represents some UML fragment, such as a name, or the contents of a class. On the RHS $_i$ denotes the corresponding code fragment derived by applying the mapping rules recursively. Metavariables used in the LHS of a rule should be named left to right in increasing order _1, _2, _3 etc. A metavariable $_*$ may also be used to denote a list of 1 or more elements. A second metavariable for lists is $_+$. $_*$ should precede $_+$ on the LHS of a rule if both are used.

Specialised rules are listed before more general rules. Given a source text element *elem*, in category *Cat*, the first rule of ruleset *Cat* whose LHS matches *elem* is applied to *elem*, with metavariables $_i$ of the LHS being bound to fragments within *elem*. These fragments are then themselves mapped by the rulesets appropriate for their categories, and the result of transformation substituted for $_i$ on the RHS of the rule. If no rule of *Cat* applies, the element is mapped to itself. Thus in the above rule set, types such as *int*, *double* are mapped without change from UML to Java. However, for Swift, the rules would be:

```
Type::
int |-->Int
long |-->Int64
boolean |-->Bool
double |-->Double
Sequence(_1) |-->[_1]
Set(_1) |-->Set<_1>
```

As an example of code generation, the following KM3 text of a UML package:

```
package App
{ class CDO
  { reference sectors : Set(Sector); }

  abstract class Sector
```

```
  { attribute probDefault : double;
    attribute lossAmount : int;
  }

  class IndustrySector extends Sector
  { reference companys : Set(Company); }

  class BankingSector extends Sector
  { reference banks : Set(Bank); }

  class Bank
  { attribute name : String; }

  class Company
  { attribute name : String; }
}
```

Would be converted by the above Java rules into Java code beginning:

```
package App;

class CDO
{
  HashSet<Sector> sectors = new HashSet<Sector>();
}

abstract class Sector
{
  double probDefault;
  int lossAmount;
}
...
```

2.1 Writing $\mathcal{CSTL}$ specifications

The following categories of rules are available for UML/OCL to 3GL code generation:

- Type rules, defining the target language equivalents of usages of UML types *int*, *double*, *long*, *String*, *boolean*, enumerations, class types and collection types. These rules are listed together, following the header *Type* ::.

  For example:

  ```
  Type::
  Set(_1) |-->HashSet<_1>
  Sequence(_1) |-->ArrayList<_1>
  ```

  map UML collection types to Java 7+ templated types.

- Enumeration rules, defining the interpretation of enumerated type definitions and their literal values.

- Expression rules, defining the interpretation of OCL expressions. These are divided into categories of basic expressions, binary expressions, unary expressions, conditional expressions and set expressions. For example:

```
BasicExpression::
_1._2(_3) |-->_1._2(_3)
_1(_2) |-->_1(_2)
_1._2 |-->_1.get_2()
```

define rules for operation calls with and without navigation in the function, and navigation of data features. These would map a call $obj.op(x.att)$ into $obj.op(x.getatt())$.

- Package and class rules, which define how the top-level structure of a class diagram maps to a program.
- Attribute and operation rules, defining how class data and operation features are mapped. Attribute rules either have the format

  ```
  attribute _1 : _2;
  ```

  for data features of string, numeric, boolean or enumerated type, or the format

  ```
  reference _1 : _2;
  ```

  for data features of collection or class types. The keywords *identity* or *static* can be used for the *attribute* case.
- Parameter rules for formal parameter and actual parameter (argument) lists in operation declarations and calls.
- Statement rules, defining how the high-level activity statements of AgileUML are interpreted as program code.
- Use case rules defining how use cases are interpreted as program code.
- Text rules, defining pattern matching and replacement based on unstructured text, eg.:

  ```
  createByPK_1(_2) |-->_1.createByPK_1(_2)
  ```

  These are listed with the basic expressions. Three formats are available on the LHS: $text_1()$; $text(_1)$ and $text_1(_2)$.

Rules are applied recursively in order to transform arbitrarily complex expressions, statements and classes, each application replaces a metavariable $_i$ on the LHS with the transformed text of whatever source text was in the $_i$ place. Thus, given some binary and unary expression rules:

```
UnaryExpression::
-_1 |-->-_1
+_1 |-->+_1
_1->log() |-->Math.log(_1)
_1->exp() |-->Math.exp(_1)
_1->sin() |-->Math.sin(_1)
_1->cos() |-->Math.cos(_1)
_1->tan() |-->Math.tan(_1)
_1->sqr() |-->(_1)*(_1)
_1->sqrt() |-->Math.sqrt(_1)
_1->cbrt() |-->Math.cbrt(_1)
_1->floor() |-->((int) Math.floor(_1))
_1->ceil() |-->((int) Math.ceil(_1))
_1->round() |-->((int) Math.round(_1))
```

```
_1->size() |-->_1.size()
_1->first() |-->Ocl.first(_1)
_1->last() |-->Ocl.last(_1)
_1->tail() |-->Ocl.tail(_1)
_1->front() |-->Ocl.front(_1)
_1->reverse() |-->Ocl.reverse(_1)

_1->max() |-->Ocl.max(_1)
_1->min() |-->Ocl.min(_1)
_1->sum() |-->Ocl.sum(_1)
_1->sort() |-->Ocl.sort(_1)
not(_1) |-->!(_1)
_1->isEmpty() |-->(_1.size() == 0)
_1->notEmpty() |-->(_1.size() > 0)
_1->display() |-->    System.out.println(_1 + "");
_1->asSet() |-->Ocl.asSet(_1)
_1->flatten() |-->Ocl.flatten(_1)

BinaryExpression::
_1 & _2 |-->_1 && _2
_1 or _2 |-->_1 || _2
_1 xor _2 |-->((_1 || _2) && !(_1 && _2))
_1 + _2 |-->_1 + _2
_1 - _2 |-->_1 - _2<when> _1 numeric, _2 numeric
_1 - _2 |-->Ocl.stringSubtract(_1,_2)<when> _1 String, _2 String
_1 - _2 |-->Ocl.setSubtract(_1,_2)<when> _1 Set, _2 collection
_1 - _2 |-->Ocl.sequenceSubtract(_1,_2)<when> _1 Sequence, _2 collection
_1 mod _2 |-->_1 % _2
_1 * _2 |-->_1 * _2
_1 / _2 |-->_1 / _2
```

$x \rightarrow exp() * x \rightarrow cbrt()$ is rewritten to $Math.exp(x) * Math.cbrt(x)$ by applying the binary expression rule for $*$, then applying the unary expression rules for *exp* and *cbrt* to the two arguments of the $*$ expression. Rules for operators of lower precedence (such as *or*) should be listed before those of higher precedence (such as $+$).

2.2 Rule conditions

Rules may have *conditions* of the form `metavariable property`, for example the rules for $-$ above distinguish between numeric subtraction and other meanings of the $-$ symbol by inspecting the type of elements in the metavariables. If _1 and _2 are both numeric (have *int*, *long* or *double* type) then the rule

```
_1 - _2 |-->_1 - _2<when> _1 numeric, _2 numeric
```

applies, however if the metavariable elements have String type, the next rule applies:

```
_1 - _2 |-->Ocl.stringSubtract(_1,_2)<when> _1 String, _2 String
```

Conditions are evaluated prior to selection of a rule, and they test the properties of source elements bound to variables $_i$.

Possible condition properties for expression metavariables are:

```
numeric
integer
String
```

```
Set
Sequence
collection
object
classid
value
variable
attribute
enumerationLiteral
```

The first 7 conditions concern the type of the expression, the last 5 concern the syntactic category of a basic expression: *classId* is true for an identifier which is a class name, *value* is true for a literal value, etc. *integer* means either *long* or *int*, whilst *numeric* also includes the case of *double*.

Conditions can be negated, eg:

```
_1 = _2 |-->_1 == _2<when> _1 not String, _1 not object, _1 not collection
```

But it is more concise to use the implicit order of rule checking to avoid using negations.

Conditions on the _* or _+ variables are:

```
_* any stereo
_* all stereo
```

These are quantifiers over the elements bound to _* or _+: the first is true if any of these elements satisfies predicate *stero*, and the second is true if every element satisfies *stereo*.

Possible condition properties for type rules are:

```
enumerated
class
Sequence
Set
Collection
Function
Map
Ref
```

Class means the metavariable is bound to a class type, Sequence to a sequence type, etc. The condition *collection* can be used for reference rules, to distinguish many-valued reference declarations from single-valued references.

A condition for an attribute is *identity*, meaning that it is an identity attribute (there should normally be at most one such attribute in each class).

Stereotypes can also be used as condition properties, for classes and attributes. For example:

```
Attribute::
_1 : _2 |-->  let _1 : _2<when>_1 readOnly
```

for a mapping from UML to Swift.

2.3 Metafeatures

The RHS of rules can also use *metafeatures*, which compute some function of the source element held in a metavariable. The notation is *_i`f* for metafeature *f*. Supported metafeatures are *defaultValue* for *Type*; *type*, *typename*, *elementType*, *upper* (multiplicity), *lower* (multiplicity) for expressions, attributes/references and operations; *owner* and *ownername* for attributes/references and operations, and *name* for classes, types, operations, use cases, attributes and references. *defaultSubclass* is the first concrete subclass of a given class. For an expression, *formalName* gives the formal parameter name corresponding to an actual parameter expression.

An example rule with a metafeature is:

```
for _1 : _2 do _3 |-->for (_2‘elementType _1 : _2) { _3}
```

In principle, this mechanism could provide access to any feature of the metamodel of UML class diagrams. However in most cases it is possible and preferable to use the pattern-matching facilities of rule LHS or rule condition tests.

The *var*‘*name* notation can also be used when *name.cstl* is an alternative/additional $\mathcal{CSTL}$ file which can be used to map *var*. For example, the Swift code generation rule:

```
Class::
class _1 { _2 } |-->protocol _1 { _2‘cgprotocol }\n<when> _1 interface
```

maps the content of an interface using the rules of *cgprotocol.cstl*, instead of the *cg.cstl* file. This is necessary, because Swift interface definitions are restricted in their formats. If an auxiliary file *f.cstl* is invoked by *_i*‘*f*, then *_i* and all subterms involved in computing the result for *_i* are also processed using *f.cstl* by default. The invocation relation between $\mathcal{CSTL}$ files should be acyclic.

Another point to notice in the above example rules is the use of a library, called *Ocl*, which contains definitions of specialised operations such as *String stringSubtract*(*String s*1, *String s*2). The developer of the code generator needs to provide definitions of these operations in the target language – or reuse them from a similar language. For example, the *stringSubtract* operation is already defined in *Ocl*.∗ libraries for Java 6, Java 7 and other languages.

2.4 Defining new functions

The language can also be used to define new functions, via the metafeatures facility. For example, a function to obtain all inherited and direct data features of a class could be expressed as a $\mathcal{CSTL}$ script called *allDataFeatures.cstl*:

```
Class::
class _1 { _2 }   |-->_2

class _1 extends _2 { _3 }   |-->_2‘allDataFeatures _3

abstract class _1 { _2 }   |-->_2

abstract class _1 extends _2 { _3 }   |-->_2‘allDataFeatures _3

class _1 implements _2 { _3 }   |-->_3

class _1 extends _2 implements _3 { _4 }   |-->_2‘allDataFeatures _4

abstract class _1 implements _2 { _3 }   |-->_3

abstract class _1 extends _2 implements _3 { _4 }   |-->_2‘allDataFeatures _4

Attribute::
identity attribute _1 : _2; |-->  identity attribute _1 : _2;\n
static attribute _1 : _2; |-->  static attribute _1 : _2;\n
attribute _1 : _2; |-->  attribute _1 : _2;\n
reference _1 : _2; |-->  reference _1 : _2;\n

Operation::
query _1(_2) : _3 pre: _4 post: _5 activity: _6 |-->

operation _1(_2) : _3 pre: _4 post: _5 activity: _6 |-->
```

```
static query _1(_2) : _3 pre: _4 post: _5 activity: _6 |-->

static operation _1(_2) : _3 pre: _4 post: _5 activity: _6 |-->
```

This script invokes itself recursively on the ancestors of each class. Attributes and references of the classes are accumulated, whilst operations are discarded. This function is useful for code generation into languages such as Python and Go which do not have object-oriented inheritance.

2.5 Using $\mathcal{CSTL}$ specifications

The LHS of rules for UML/OCL code generation have a fixed format and these cannot be varied. The files *cg/cgJava8.cstl* and *cg/cgSwift.cstl* give examples of all the available rules. The RHS and conditions of rules can be varied to generate text in many different languages.

Write your main translation rules for target language L in a file *cg/cgL.cstl*. The syntax of this can be checked by loading it into the Agile UML tools via the File menu option *Load transformation* ⇒ *Load CSTL*. Warning messages are given if rules cannot be parsed. Additional $\mathcal{CSTL}$ files in the *cg* directory are also loaded by this step and can be invoked within *cgL.cstl* by their names.

If you have defined a class diagram model, this can be translated to code via *cgL.cstl* by selecting the *Build* menu option *Use CSTL specification*. The output text is written to *output/cgout.txt*.

For some languages, multiple output files need to be produced from the same source model, for example, C header and code files. Separate code generator $\mathcal{CSTL}$ files should be written to produce each output.

3 Program abstraction

$\mathcal{CSTL}$ scripts can also be written to process input data of any source language in the form of abstract syntax trees/parse trees according to the metamodel of Figure 1. We refer to this more general form of the language as CGTL.

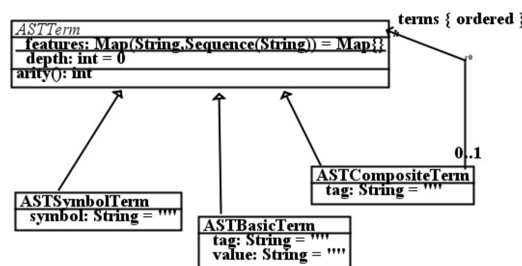

Figure 1: Metamodel of parse trees

In textual syntax, parse trees are written as nested LISP-style lists, eg.:

```
(unaryExpression ++ (primaryExpression x))
```

for the C expression `++x`.

Such parse trees can represent program code, specifications in UML/OCL, or even syntax trees of natural language texts.

The same structure of scripts as described in Section 2 can be used to process parse trees, with three main differences:

- The left hand sides of rules separate successive subterms by a space, eg:

```
unaryExpression::
++ _1 |-->  _1 := _1 + 1
```

- Rulesets are named by the tag of the terms they are intended to process. This is usually the same name as the grammar rule of the source language grammar, and should be distinct from the UML/OCL tags such as Type, BasicExpression, etc.
- Functions local to a script can be defined as rulesets within the script, instead of as separate script files.
- Java external functions can be invoked from scripts.

Lowercase initial letters are typically used for the names of CGTL rulesets.

3.1 Metafeatures for CGTL

The following metafeatures on terms can be used:

- $_i$‘*first* – the first subterm of the term bound to $_i$. Also *second*, *third*, up to *seventh*.
- $_i$‘*last* – the last subterm of the term bound to $_i$
- $_*$‘*first* – the first term of the term list $_*$ (also for $_+$)
- $_*$‘*last* – the last term of the term list $_*$ (also for $_+$)
- $_*$‘*front* – the front sublist of the list of terms $_*$ (or of $_+$)
- $_*$‘*tail* – the tail sublist of the list of terms $_*$ (or of $_+$). Also *tail*2 and *tail*3 for 2 and 3 applications of *tail*
- $_*$‘f for ruleset f – apply f to each element of the list of terms $_*$, concatenating the strings that result (or of $_+$)
- $_*$‘*recurse* – apply the script to (r *terms*) where r is the ruleset containing this expression, and *terms* are the terms in the list bound to $_*$.

3.2 Dynamic term information

The *features* map in Figure 1 provides a means to record information from one part of an AST for use elsewhere in processing the AST. Typically, type information is stored during processing variable declarations, which can then be used in processing expressions and statements involving the variables. For example, $x + y$ in Python has different semantic representations depending on the types of x, y. The *features* map associates the literal string form of an AST term with a list of stereotypes and meta-attributes or tag=value pairs. For example we could have the mapping

$$\text{“}x\text{”} \mapsto [\text{“}int\text{”}, \text{“}defined = true\text{”}]$$

for an integer variable x, to assert that it has the type *int* and has been explicitly defined. Such information can be queried by rule conditions and modified by rule actions.

3.3 CGTL rule conditions

The condition test

```
<when> _i tag
```

returns true if $_i$ is bound to a term with tag *tag*, or if the literal form of $_i$ has stereotype *tag*, otherwise false. Condition tests can also check the apparent type of a term:

```
<when> _i T
```

for T being *character*, *integer*, *real*, *boolean*, *String*, *Sequence*, *Set*, *Map*, *Function* and custom types represented as stereotypes. Metafeatures, matching, negation and quantifier conditions can also be used, as described in Table 1. Note that conditions are generally evaluated on the input terms before they are processed by CSTL/CGTL. Processing by a term by a ruleset f needs to be explicitly indicated by $_i`f$.

Condition	*Meaning*	*Example*
$_i\ t$	The $_i$ term has tag t	`_1 recordType`
$_i\ P$	The literal form of term $_i$ has type/stereotype P	`_1 String`
$_*\ all\ t$	All terms in $_*$ have tag t	
$_*\ any\ t$	Some term in $_*$ has tag t	`_+ any picOccurs`
$_i$ *matches* re	Literal form of $_i$ matches regular expression re	`_1 matches .capitalize()`
$_i`f\ P$	$_i$ term literal form has tagged value $f = P$ or ruleset f applied to $_i$ term evaluates to/has stereotype P	`` _2`isObjectPointer true `` `` _1`ocltype int ``

Table 1: Condition tests in CGTL

Properties P can be literal values, metavariables, or $_i`f$ for metavariable $_i$ and tagged value or ruleset name $f$. If $f$ is a ruleset name, $_i`f$ is the result of applying that ruleset to the term bound to $_i$, otherwise $_i`f$ is the tagged value of the literal form a of the term bound to $_i$. That is, if

$$features[a] = vect$$

it is the value v associated with f in an element "f=v" of *vect*.

It is recommended to use disjoint sets of names for tagged value meta-attributes, stereotypes, and ruleset names. Rulesets cannot be dynamically modified at runtime. Only tagged values and stereotypes and global variables can be dynamically modified.

3.4 CGTL rule actions

In place of/in addition to a <when> clause, an <action> clause can be defined:

```
<action> _i T
```

This records a stereotype for the translation of $_i$ which can then be tested by subsequent <when> clauses. Variables $_i$ in the action refer to the translation of $_i$. Table 2 summarises the different forms of CGTL action.

Action	*Meaning*	*Example*
$_i\ P$	Set translation of term $_i$ to have type/stereotype P	`_1 String`
$_\$\ P$	Set value of global variable $_\$$ to P	`_$ _1`
$_*\ all\ t$	Set all evaluated terms in $_*$ to have property P	
$e\ /\ x$	Replace x by e in the RHS result of rule	`0 / x`
$_i`f\ P$	Set f tagged value of $_i$ translation $f = P$	`` _2`isObjectPointer true ``

Table 2: Action forms in CGTL

An example of the use of actions is the following script:

```
Typ::
_1 |-->_1

Decln::
```

```
var _1 : Sequence ( _2 ) |-->  _2[] _1;\n<action> _1 Sequence
var _1 : _2 |-->  _2 _1;\n<action> _1 _2

Stat::
print _1 |-->  displaySequence(_1);\n<when> _1 Sequence
print _1 |-->  displayint(_1);\n<when> _1 int
print _1 |-->  displaydouble(_1);\n<when> _1 double
print _1 |-->  display(_1);\n

Prog::
_* |-->_*
```

If a declaration *var x : int* is encountered, then the type stereotype *int* is recorded for x, so that a subsequent statement *print x* is translated to *displayint*(x); .

Rule actions are a powerful facility for expressing complex processing, however they also make scripts more difficult to understand and modify. Where possible their use should be avoided.

3.5 CGTL examples

Examples of abstraction transformations are in Java2UML.cstl, JS2UML.cstl, VB2UML.cstl, python2UML.cstl, pascal2UML.cstl, cobol2UML.cstl and C2UML.cstl, together with their associated auxiliary scripts. These take as input parse trees in *output/ast.txt* produced by ANTLR parsers for Java, JavaScript, VisualBasic6, Python, pascal, Cobol85 and C, respectively. They are executed via the *File* menu options "From Java AST", "From JavaScript AST", "From Python AST", "From Pascal AST", "From VB6 AST", "From COBOL AST" and "From C AST".

The *Build* menu option "Apply CSTL to AST" loads a selected *.cstl* file from the *cg* subdirectory, and applies this to *output/ast.txt*.

To write an abstraction script for a new programming language, use a structure similar to the VB2UML.cstl, pascal2UML.cstl or python2UML.cstl abstractors, with rulesets specific to the grammar of the new language. As described above, the *antlr2cstl* script can also be used to generate an outline CGTL script for a language.

A CGTL script *script.cstl* can be applied to an ast file by the command line command:

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

4 Machine learning of $\mathcal{CSTL}$ scripts

Automated construction of code generation scripts from text examples is also supported, in prototype form. The training data consists of files *typeExamples.txt*, *expressionExamples.txt*, *statementExamples.txt* and *declarationExamples.txt* in the *output* directory, with the format

```
UML/OCL text      Target language text
```

with one line for each example, and the source and target text separated by one or more tabs. E.g, in *typeExamples.txt*:

```
Sequence(String)         ArrayList<String>
Map(String,Person)       HashMap<String,Person>
```

for translation to Java. Two or more different examples for each specific rule are required.

The target program language grammar and grammar rules should be specified in a file *configuration.txt*. Eg., for C:

```
parser = C
typeRule = typeSpecifier
expressionRule = expression
```

```
statementRule = statement
featureRule = externalDeclaration
classRule = externalDeclaration
localVariableRule = blockItem
enumerationRule = externalDeclaration
```

The ANTLR parser for the specified target language needs to be available, and is invoked to process each target text example of the appropriate kind, and the output (as parse trees) is then used for symbolic machine learning of tree-to-tree mappings from the corresponding parse trees of the UML/OCL source texts [4]. Run `CGBE` from the AgileUML *Synthesis* menu to produce a code generator script *output/tl.cstl* from the text training data. This option can also be executed by the command

```
cgbeTrain configuration.txt
```

The grammar categories of UML/OCL are provided as a metamodel *output/mmCGBE*. This is used by the above options, as is the outline mapping of grammar categories *output/forwardCGBE.tl*.

To validate a learned CSTL file, run:

```
cgbeValidate output/tl.cstl output/asts.txt
```

where *asts.txt* are a series of UML/OCL ASTs, one per line.

To learn a general language-to-language mapping, use the option LTBE from the AgileUML *Synthesis* menu. "LTBE from ASTs" learns a tree-to-tree mapping of corresponding source language ASTs (in textual format in file output/sourceasts.txt) and target language ASTs (in file output/targetasts.txt), one example per line of each file. "LTBE from text" uses a series of text example files formatted with lines

```
SourceText      TargetText
```

with the source and target separated by one or more tabs. ANTLR parsers for source and target languages are needed.

5 Further information

Publications using $\mathcal{CSTL}$ include [5, 3, 1, 6, 2].

The latest version of AgileUML is at https://github.com/eclipse/agileuml, code generator and abstraction scripts are in cg.zip and supporting libraries in libraries.zip either on GitHub or at https://agilemde.co.uk. cgbe.zip contains source files to perform code generator synthesis from examples.

References

[1] K. Lano. Implementing OCL in Swift. In *OCL 2021*, 2021.

[2] K. Lano. Program translation using Model-driven engineering. In *ICSE 2022 Companion Proceedings*, pages 362–363, 2022.

[3] K. Lano, S. Kolahdouz-Rahimi, and L. Alwakeel. Synthesis of mobile applications using AgileUML. In *ISEC 2021*, pages 1–10, 2021.

[4] K. Lano and Q. Xue. Code generation by example using symbolic machine learning. *Springer Nature Computer Science*, 2023.

[5] Kevin Lano and Qiaomu Xue. Agile specification of code generators for model-driven engineering. In *2020 15th International Conference on Software Engineering Advances (ICSEA)*, pages 9–15, 2020.

[6] Kevin Lano and Qiaomu Xue. Code generation by example. In *Proceedings of the 10th International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, pages 84–92, 2022.