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

K. Lano

April 14, 2020

1 Introduction

$\mathcal{CSTL}$ is a simple text-based language for defining code generators for the UML-RSDS subset of UML.

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 UML-RSDS 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>
```

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

The $\_i$ for positive integer i 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. Variables should be named successively $\_1$, $\_2$, etc, up to $\_9$.

Specialised rules are listed before more general rules. Given a source text element *elem*, the first rule 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 rule set and the result of transformation substituted for $\_i$ on the RHS of the rule. If no rule applies, an element is mapped to itself.

As an example, 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 rules into Java code beginning:

```
package App;

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

```
}

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

2 Writing $\mathcal{CSTL}$ specifications

The following categories of rules are available:

- 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.

- Statement rules, defining how the high-level activity statements of UML-RSDS 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.:

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

  These apply to basic expressions and statements.

Rules are applied repeatedly 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.

A built-in rule is that a list of expressions $\_1, \_2$ is mapped to $\_1, \_2$, where $\_1$ is an individual expression and $\_2$ a list of expressions.

Rules may have *conditions*, 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
```

Possible conditions for expression metavariables are:

```
numeric
String
```

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

The first 6 concern the type of the expression, the last 4 concern the syntactic category of a basic expression: *classId* is true for an identifier which is a class name, *value* for a literal value, etc.

Conditions can be negated, eg:

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

Possible conditions for type rules are:

```
enumerated
class
```

Class means a class type. 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 *primary*, meaning that it is the first identity attribute listed in its owner class.

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

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

for a mapping from UML to Swift.

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 *type*, *typename* and *elementType* for expressions, attributes and operations, *owner* and *ownername* for attributes and operations, and *name* for classes, types, operations, use cases, attributes and references. An example rule is:

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

In general, this mechanism provides access to any feature of the abstract syntax of UML-RSDS class diagrams.

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.

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.

3 Using $\mathcal{CSTL}$ specifications

The LHS of rules have a fixed format and these cannot be varied. The files *output/cgJava*8*.cstl* and *output/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 in a file *output/cg.cstl*. The syntax of this can be checked by loading it into the Agile UML tools via the File menu option *Load transformation* $\Rightarrow$ *Load CSTL*. Additional $\mathcal{CSTL}$ files in the *output* directory are also loaded by this step and can be invoked within *cg.cstl* by their names.

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

The latest version of the tools can be obtained from: https://nms.kcl.ac.uk/kevin.lano/uml2web/.