Implementing OCL in Swift

K. Lano[1]

[1]Dept. of Informatics, King's College London, UK

Abstract

Swift is the new programming language for Apple platforms such as MacOS and iOS, and it is the basis of the SwiftUI mobile app framework. The language incorporates first-class functions, open classes, and other advanced features.

In this paper we describe a tool for translating UML and OCL specifications into fully-functional code in Swift. Translation of OCL extensions (map and function types) are also described.

Keywords

Object Constraint Language (OCL), Code generation, Swift, CSTL

1. Introduction

Swift (https://swift.org) was introduced at Apple's Developer's Conference (WWDC) in 2014 as the preferred language for future program development on Apple platforms, replacing Objective-C. Swift can be characterised as a compiled language with strong typing, containing an object-oriented programming core but with extensions (eg., structs and global operations) to the object-oriented paradigm. It provides functions as first-class objects, and anonymous functions (termed *closures*) can be defined and used as function values. It has a universal type *Any* (cf., OCL's *OclAny*) and 'absence of value' value *nil* (cf., OCL *null*).

We describe the details of the AgileUML tool[1] mapping from UML+OCL to Swift in Section 2. In Section 3 we describe the implementation of OCL extensions for map and function types.

2. Translation from OCL to Swift

At a high level, the translation from UML+OCL to Swift can be defined as follows:

1. UML/OCL types are mapped to corresponding Swift types, with the exception that *OclMessage* and *UnlimitedNatural* are not mapped.
2. UML data features are mapped to Swift variables or constants of corresponding type and scope (instance or class scope).
3. OCL expressions are mapped to corresponding Swift expressions, defined directly in terms of Swift expression operators, or by using a library *Ocl.swift* which provides

OCL 2021: 20th International Workshop on OCL and Textual Modeling, June 25 2021, Bergen, Norway

kevin.lano@kcl.ac.uk (K. Lano)

CEUR Workshop Proceedings (CEUR-WS.org)

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

implementations of OCL operators[2]. 101 OCL 2.4 library operators are supported by the translation, out of 181 operators in [11].

4. The translation of UML activities to program statements is direct.
5. Operations and use cases map to Swift local and global functions.
6. UML class definitions map to Swift classes, with additional auxiliary variables and operations to support use of the *allInstances*() operator.

We describe the translation by using the $\mathcal{CSTL}$ notation for code-generator specification [8]. A $\mathcal{CSTL}$ script defines a text-to-text code generator, and consists of a sequence of rules grouped into source language syntax categories. Individual rules have the form:

```
selement |--> telement <when> Condition
```

The <*when*> clause and condition are optional. The left hand side (LHS) of a $\mathcal{CSTL}$ rule is some piece of textual concrete syntax in the source language, e.g., in Kernel Metamodel (KM3) [5] textual notation for UML class models, and the right hand side (RHS) is the corresponding concrete syntax in the target language (e.g., C, Java, Swift, etc), which the LHS should translate to. Apart from literal text concrete syntax, the LHS may contain variable terms _1, _2, etc, representing arbitrary source concrete syntax fragments, and the RHS refers to the translation of these fragments also by _1, _2, etc. Specialised rules are listed before more general rules. The default translation (if no rule applies) is textual copying.

For example, to map UML/OCL type occurrences to Swift 5, the following rules are used:

```
Type::
int |-->Int32
long |-->Int64
Boolean |-->Bool
double |-->Double
OclVoid |-->Void
OclAny |-->Any

Sequence(_1) |-->[_1]
Set(_1) |-->Set<_1>
```

No rules for occurrences of enumerations, String or entity types are needed because their syntactic form is identical in OCL and Swift.

The semantics of a $\mathcal{CSTL}$ script is based on recursive application of string replacement in the RHS of rules [8]. For a rule

```
LHS[_1,...,_n] |--> RHS[_1,...,_n] <when> Cond[_1,...,_n]
```

if the LHS matches against a piece of source text $LHS[t_1, ..., t_n]$, with each t_i instantiating $\_i$, then if $Cond[t_1, ..., t_n]$ holds, the output text $RHS[t'_1, ..., t'_n]$ is produced, where each t'_i is the result of applying the script to t_i. $\mathcal{CSTL}$ provides a simple and concise notation for defining

[2]This library of 1107 LOC can be used independently of the UML to Swift translator. This library and OCL libraries for other languages are available from [1].

code generators, but is relatively limited in expressiveness compared to template languages such as Acceleo or EGL.

The rules for translating OCL expressions are divided into five groups: (1) Basic expressions, for literal values, variables, feature applications, etc; (2) Unary expressions, for one-argument expression forms $not(e)$, $e{\rightarrow}size()$, etc; (3) Binary expressions, for infix binary and binary $\rightarrow$ operators: $x + y$, $sq{\rightarrow}union(s)$, etc; (4) collection expressions $Set\{\}$, $Sequence\{x1, x2\}$, etc; (5) Conditional and let expressions.

The translation of basic expressions is direct, and includes rules such as:

```
null |-->nil
_1.allInstances() |-->_1_allInstances
```

For each class E, a global scope sequence *var E_allInstances* : $[E]$ of current E instances is maintained in the Swift implementation.

Prefix unary expressions $not(e)$ and $-e$ also translate directly to $!(e')$ and $-e'$ in Swift, where e' is the translation of e. Postfix unary expressions either translate directly to Swift, or into *Ocl.swift* calls, eg.:

```
_1->size() |-->_1.count
_1->max() |-->Ocl.max(s: _1)
_1->sum() |-->Ocl.sum(s: _1)
```

Likewise, with binary expressions a direct translation is possible in some cases:

```
_1 mod _2 |-->_1 % _2

_1->union(_2) |-->_1.union(_2)<when> _1 Set
_1->union(_2) |-->_1 + _2<when> _1 Sequence
_1->collect(_2 | _3) |-->_1.map({_2 in _3})
```

where $\{x\ in\ e\}$ is Swift syntax for the lambda expression $\lambda\, x \cdot e$.

Some collection operators require specialised implementations:

```
_1->sortedBy(_2|_3) |-->Ocl.sortedBy(s: _1, f: { _2 in _3 })

_1->select(_2 | _3) |-->Ocl.select(s: _1, f: { _2 in _3 })
_1->reject(_2 | _3) |-->Ocl.reject(s: _1, f: { _2 in _3 })
```

Conditional expressions have a similar implementation in many 3GLs:

```
if _1 then _2 else _3 endif |-->((_1) ? _2 : _3)
```

To ensure that variables never hold *null* objects, we use the *Default instance* pattern, a variant on the Singleton design pattern: for each class E, a static method $defaultInstance() : E$ is defined, which returns an existing instance of E if there is any E instance, otherwise it creates a new E instance and returns it. Generated object variable declarations then have the form

```
var v : E = E.defaultInstance()
```

3. Translation of map and function types

Proposals for extending OCL with map types have been made by several researchers [7, 14, 10], and proposals for OCL function types are given in [10, 13]. Table 1 summarises our OCL extension map operators and their translations to Swift. These are a superset of the Eclipse OCL Map operations [2]. The *force unwrap* expression *e*! throws an exception if *e* is *nil*.

Map type/expression E	Semantics	Swift 5 implementation E
$Map(S, T)$	Finite maps from S to T	Dictionary<S,T>
$m \rightarrow at(x)$	m applied to x	m[x]!
$m[x] = y$	m updated by $x \mapsto y$	m[x] = y
$Map\{k1 \mapsto v1, ..., kn \mapsto vn\}$	Literal map	[k1:v1, ..., kn:vn]
$m \rightarrow size()$	Number of mappings in m	m.count
$m1 \rightarrow keys()$	Set of keys in $m1$	Ocl.mapKeys(m: m1)
$m1 \rightarrow values()$	Range of $m1$	Ocl.mapRange(m: m1)
$m1 \rightarrow union(m2)$	$m1$ overridden by $m2$	Ocl.unionMap(m1: m1, m2: m2)
$m1 \rightarrow intersection(m2)$	Common mappings of $m1$, $m2$	Ocl.intersectionMap(m1: m1, m2: m2)
$m1 - m2$	Mappings of $m1$ not in $m2$	Ocl.excludeAllMap(m1: m1, m2: m2)
$m \rightarrow restrict(ks)$	Mappings of m with key in ks	Ocl.restrict(m: m, ks: ks)
$m \rightarrow select(x \mid P)$	$s \mapsto t$ of m where $t \models P$	Ocl.selectMap(m: m, f: { x in P })
$m \rightarrow collect(x \mid e)$	Map composition of m and e	Ocl.collectMap(m: m, f: { x in e })

Table 1
Translation of map type and operators to Swift

Table 2 gives the implementation of function types and function operators in Swift[3]. An important feature of function abstraction in Swift, Java and Python is *scope capture*: identifiers in scope at the point of definition of the lambda expression can be used (for read-only access) in its body. Our translation supports this feature for these languages.

Function type/expression E	Semantics	Swift 5 implementation E
$Function(S, T)$	Type of functions from S to T	(S) -> T
$f(x)$	Application of f to x	f(x)
lambda $x : S$ *in* E E of type T	Function abstraction $\lambda\, x : S \cdot E$	{ (x : S) -> T in E }

Table 2
Translation of function type and operators to Swift

We extend the $\mathcal{CSTL}$ code generation specification of Section 2 with type and expression rules for maps and functions, eg.:

```
Map(_1,_2) |-->Dictionary<_1,_2>
Function(_1,_2) |-->(_1)->_2

_1->keys() |-->Ocl.mapKeys(m: _1)
_1->values() |-->Ocl.mapRange(m: _1)
```

[3]Functions are immutable values, in contrast to maps.

```
_1 |-> _2 |-->_1:_2
_1->at(_2) |-->_1[_2]!<when> _1 Map

_1->union(_2) |-->Ocl.unionMap(m1: _1, m2: _2)<when> _1 Map
_1->intersection(_2) |-->Ocl.intersectionMap(m1: _1, m2: _2)<when> _1 Map
_1 - _2 |-->Ocl.excludeAllMap(m1: _1, m2: _2)<when> _1 Map
_1->restrict(_2) |-->Ocl.restrict(m: _1, ks: _2)

_1->collect(_2 | _3) |-->Ocl.collectMap(m: _1, f: {_2 in _3})<when> _1 Map
```

The default map value is the empty map `[:]`, the default function value of $Function(S, T)$ is the function that returns the default value of T for each S element.

Lambda expressions are directly implemented by closures:

```
lambda _1 : _2 in _3 |-->{ (_1 : _2) -> _3‘type in _3 }
```

_i‘type denotes the translation of the *type* metafeature of the source expression bound to *_i*.

In the directories *oclmapexamples.zip* and *oclfunctionexamples.zip* on [1] we give examples of specifications using maps and functions, together with the corresponding generated code in Java, C, Swift and Python. The code generators and support libraries for these languages, and examples of iOS apps synthesised using the Swift code generator [9] are also provided on [1].

4. Evaluation

In this section we evaluate the efficiency of map and function code implementations generated by AgileUML. Two evaluation cases are used: (i) a word-count algorithm using maps; (ii) a numerical optimisation procedure using functions. All Java, C and Python tests were carried out on a Windows 10 i5-6300 dual core laptop with 2.4GHz clock frequency, 8GB RAM + 3MB cache. For Swift we also tested execution on a similar iMac running MacOS 10 (dual core i5, 2.3GHz/8GB RAM/4MB cache). The REM IDE was used for Swift execution on Windows[4].

4.1. Maps example: word counts

This example stores word counts of the words in an input sequence *sq* in a result map using assignments `result[x] := sq->count(x)`.

Table 3 shows the execution times of *wordCount* for inputs of 1000, 10000 and 100000 words, for the Swift, Java, Python and C implementations generated from the example using AgileUML. All times are in seconds, computed as an average of 3 independent executions, using the same datasets. The MacOS implementation of Swift is intermediate in performance between C and Python.

[4] https://www.remobjects.com

	$\#sq = 1000$	$\#sq = 10000$	$\#sq = 100000$
Java	0.024	0.236	21.212
Python	0.012	1.245	134.4
Swift (MacOS)	0.012	0.97	97.3
Swift (Windows)	0.087	3.24	335.5
C	0.007	0.604	53.7

Table 3
Performance of Maps example

4.2. Functions example: numerical optimisation

The test example for functions uses function types and function evaluation as part of a numerical optimisation procedure:

```
class SomeFunctions {
  static operation secant(rn : double , rminus : double , fminus : double ,
    tol : double , f : Function(double,double) ) : double
  pre: true
  post: fn = f(rn) and
    (if fn->abs() < tol then result = rn
     else (
       result = SomeFunctions.secant(rn -
         fn * ( ( rn - rminus ) / ( fn - fminus ) ), rn,fn,tol,f)
     )
     endif);
}
```

The *secant* routine is invoked with f instantiated by lambda expressions *lambda* x : *double in* $x * x + x - 1$, *lambda* x : *double in* $x \rightarrow pow(x) - 0.7$. Table 4 shows the execution time of this example in Java, Python, Swift and C, for 1000 function calls, 10,000 and 100,000. This case has an approximately linear time complexity. Swift on MacOS is faster than both Python and Java, and comparable to C.

	1000 calls	10000 calls	100000 calls
Java	0.044	0.0463	0.051
Python	0	0.0104	0.0587
Swift (MacOS)	0.0003	0.0016	0.0143
Swift (Windows)	0.938	0.964	1.1
C	0	0.0007	0.006

Table 4
Performance of Functions example

5. Related work

Three general approaches for defining mappings from UML and OCL to programming languages are Model-to-model (M2M), Model-to-text (M2T) or Text-to-text (T2T). Table 5 compares exam-

ples of the three approaches with regard to their size and scope. T2T solutions are substantially more concise than M2M or M2T solutions, relative to the supported functionality (scope) of the code generator.

Approach	*Case*	*Implementation*	*Size (LOC)*	*Scope*
M2M	UML2Java [4]	QVT-R	4308	Class diagrams
	UML2C [6]	UML-RSDS	4492	Class diagrams, OCL, use cases, activities
M2T	UML2Java [3]	Acceleo	3957	Class diagrams, OCL
	UML2Java [12]	EGL	1425	Class diagrams, statemachines
	UML2Python [1]	UML-RSDS	1715	Class diagrams, OCL, use cases, activities
T2T	UML2Swift	$\mathcal{CSTL}$	445	Class diagrams, OCL, use cases, activities
	UML2Java8 [8]	$\mathcal{CSTL}$	426	Class diagrams, OCL, use cases, activities

Table 5
Comparison of code generation approaches

Conclusions

We have shown that $\mathcal{CSTL}$ can be used to define a practical translation from OCL to Swift. This translation includes support for OCL map and function extensions and regular expressions. The translation is used to generate the business tier code of SwiftUI apps [9].

References

[1] AgileUML repository, https://github.com/eclipse/agileuml/, 2021.
[2] Eclipse OCL, https://projects.eclipse.org/projects/modeling.mdt.ocl, 2021.
[3] Eclipse UML2Java code generator, https://git.eclipse.org/c/umlgen/, accessed 18.8.2020.
[4] S. Greiner, T. Buchmann, B. Westfechtel, *Bidirectional transformations with QVT-R: a case study in round-trip engineering UML class models and Java source code*, Modelsward 2016.
[5] F. Jouault, J. Bezivin, *KM3: a DSL for metamodel specification*, ATLAS team, INRIA, 2006.
[6] K. Lano, S. Yassipour-Tehrani, H. Alfraihi, and S. Kolahdouz-Rahimi, *Translating from UML-RSDS OCL to ANSI C*, OCL 2017, STAF 2017, pp. 317–330.
[7] K. Lano, *Map type support in OCL?*, https://www.eclipse.org/forums/index.php/t/1096077/, November 2018.
[8] K. Lano, Q. Xue, S. Kolahdouz-Rahimi, *Agile specification of code generators for model-driven engineering*, ICSEA 2020.
[9] K. Lano et al., *Synthesis of mobile applications using AgileUML*, ISEC 2021.
[10] K. Lano, S. Kolahdouz-Rahimi, *Extending OCL with map and function types*, FSEN 2021.
[11] OMG, *Object Constraint Language 2.4 Specification*, 2014.
[12] TU/e, SLCOtoJava1.0 code generator, https://gitlab.tue.nl/SLCO, 2020.
[13] E. Willink, *Reflections on OCL 2*, Journal of Object Technology, Vol. 19, No. 3, 2020.
[14] E. Willink *An OCL Map Type*, OCL '19, 2019.