Agile Specification of Code Generators for Model-Driven Engineering

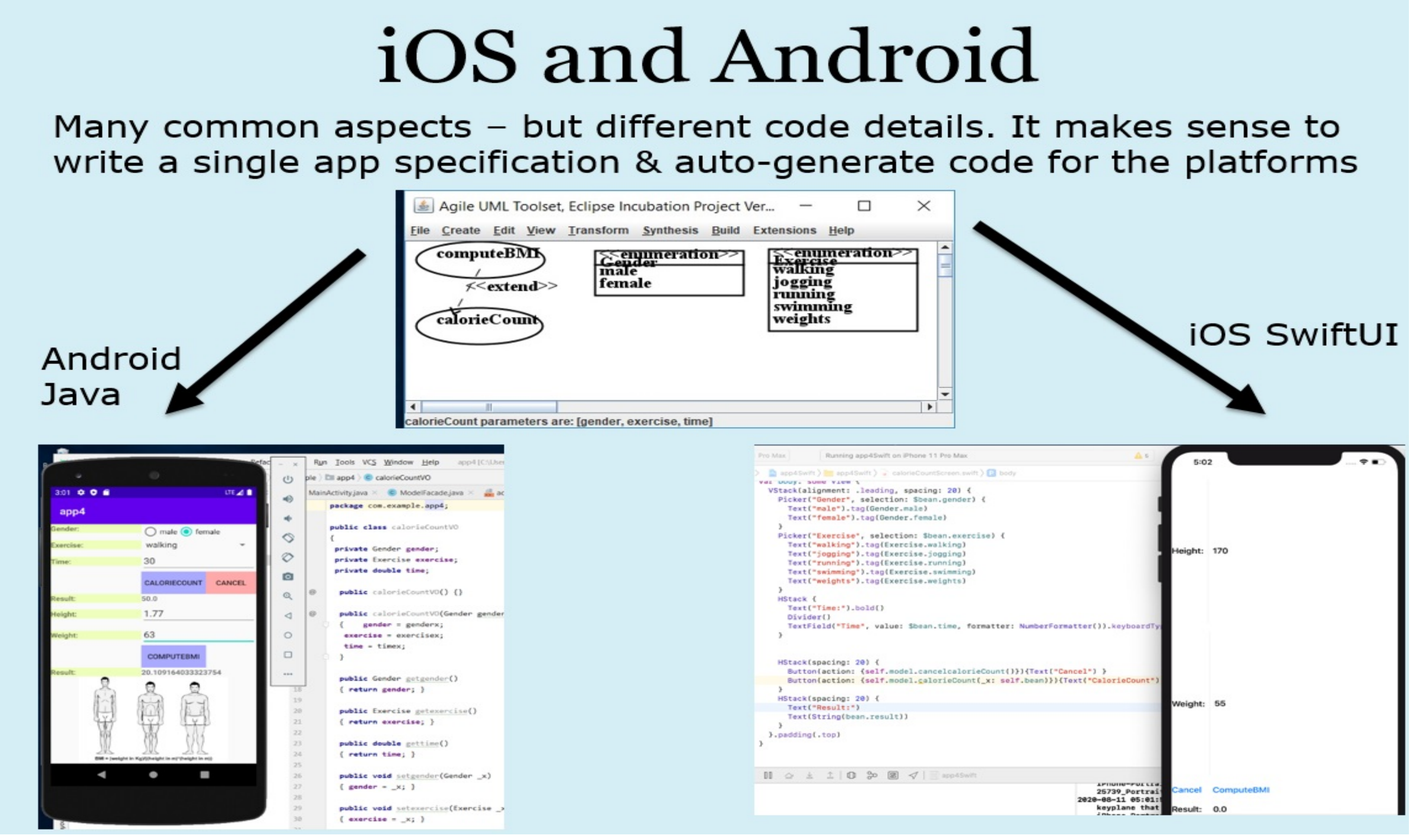
Kevin Lano, Qiaomu Xue, Shekoufeh Kolahdouz-Rahimi

August 31, 2020

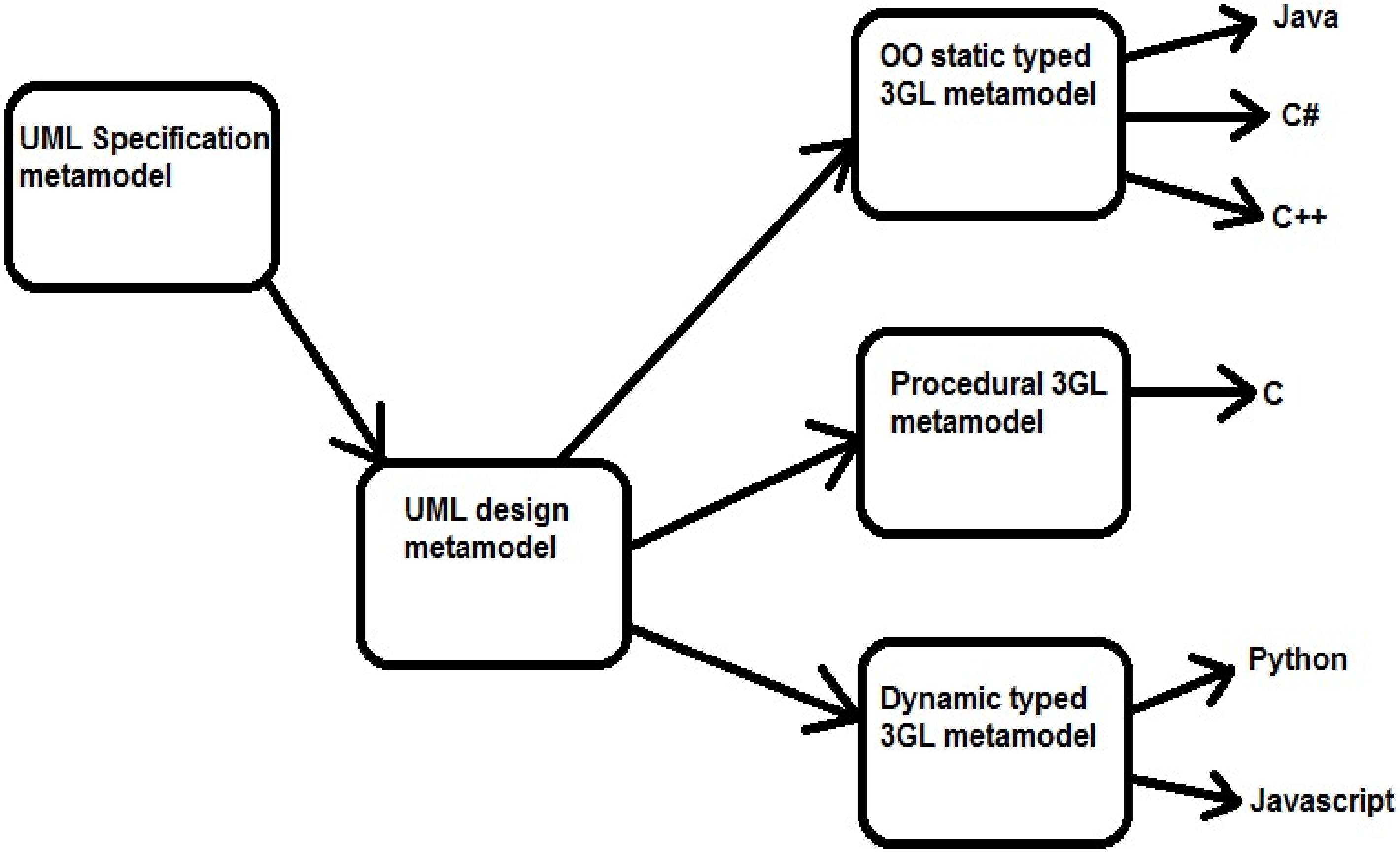
Dept. of Informatics, King's College London
Email: kevin.lano@kcl.ac.uk

Code generation in Model-driven engineering

- Model-Driven Engineering (MDE) approaches specify systems via models
- Executable code is generated (automatically) from models
- E.g., Android and iOS implementations of an app generated from its specification
- But writing code generators is an expensive task – several person years for typical UML to Java generator
- How can this task be simplified and costs reduced?



Code generation scenario



Code generation stages

Production of code generators

1. Define valid representation of source language in target language;
2. Define code generation strategy to create target representation of each source model;
3. Write and test code generation rules in a 3GL or code generation language.

Difficulty increases with distance between source and target (e.g., UML to Java is easier than UML to C).

Code generation approaches

1. *Abstract syntax to abstract syntax*: code generation rules written in 3GL or model transformation (MT) language. Map elements of abstract syntax of source language, to elements of abstract syntax of target language.
2. *Abstract syntax to concrete syntax*: templates using target concrete syntax are defined, with variable parts or slots written in source language abstract syntax.

   Specialised languages: EGL, Acceleo
3. *Concrete syntax to concrete syntax*: defined using source and target language concrete syntax, rules specify how source concrete syntax fragments map to target concrete syntax.

   Our $\mathcal{CSTL}$ language.

UML to Java using Java

```
public String queryForm(java.util.Map env, boolean local)
{ String res;
  boolean bNeeded = needsBracket;
  String cont = "Controller.inst()";

  if (operator.equals("#") || "->existsLC".equals(operator) ||
      operator.equals("#LC") || operator.equals("->exists"))
  { return existsQueryForm(env,local); }

  if (operator.equals("#1") || operator.equals("->exists1"))
  { return exists1QueryForm(env,local); }

  if (operator.equals("->forAll") || operator.equals("!"))
  { return forAllQueryForm(env,local); }

  if (operator.equals("=>"))
```

```
{ return "(!(" + lqf + ") || " + rqf + ")"; }

if (operator.equals("xor"))
{ return "((" + lqf + " && !(" + rqf + ")) || (!(" +
    lqf + ") && " + rqf + "))"; }

if (operator.equals("->oclAsType"))
{ // Type typ = Type.getTypeFor("" + right,types,entities);
  if (type != null)
  { String jtyp = type.getJava();
    return "((" + jtyp + ") " + lqf + ")";
  }
  return "((" + right + ") " + lqf + ")";
}
```

UML to Java using EGL

```
public class BooleanExpression_[%=transition.name%]
  extends BooleanExpression{
public [%= stateMachine.name%] sm;
public BooleanExpression_[%=transition.name%](Class slcoClass,
  SharedVariableList sharedVariables, [%= stateMachine.name%] sm) {
super(slcoClass, sharedVariables);
this.sm = sm;
// TODO Auto-generated constructor stub
}
@Override
public boolean evaluate() {
// TODO Auto-generated method stub
        return [%= "("+(generateExpression(statement.operand1,
  class, stateMachine) + generateOperator(statement) +
  generateExpression(statement.operand2, class, stateMachine))+")"%];
}
```

```
}
[* Generate the right of "Assignment": BinaryOperatorExpression*]
[%operation generateExpression(expression : slco::Expression,
  class : slco::Class, stateMachine : slco::StateMachine) : String{%]
[%var returnValue : String;%]
[%if(expression.isTypeOf(IntegerConstantExpression) or
  expression.isTypeOf(BooleanConstantExpression)){%]
  [% returnValue = (expression.value).toString();%][%}%]
[%else if(expression.isTypeOf(StringConstantExpression)){%]
  [%returnValue = ('"'+(expression.value).toString()+'"');%][%}%]
[%else if(expression.isTypeOf(VariableExpression)){%]
  [%returnValue = generateVariableExpression(expression.variable,
  class, stateMachine);%][%}%]
[%else if(expression.isTypeOf(BinaryOperatorExpression))
  {returnValue = "("+generateExpression(expression.operand1,
  class, stateMachine) + generateOperator(expression) + generateExpression(ex
[% return returnValue;%]
[%}%]
```

UML to Java using $\mathcal{CSTL}$

```
BinaryExpression::
_1 & _2 |-->_1 && _2
_1 or _2 |-->_1 || _2
_1 xor _2 |-->((_1 || _2) && !(_1 && _2))

_1 = _2 |-->_1.equals(_2)<when> _1 String
_1 = _2 |-->_1.equals(_2)<when> _1 object
_1 = _2 |-->_1.equals(_2)<when> _1 Sequence, _2 collection
_1 = _2 |-->Ocl.equalsSet(_1,_2) <when> _1 Set, _2 collection

_1 < _2 |-->_1 < _2<when> _1 numeric, _2 numeric
_1 < _2 |-->(_1.compareTo(_2) < 0)<when> _1 String, _2 String

_1 > _2 |-->_1 > _2<when> _1 numeric, _2 numeric
_1 > _2 |-->(_1.compareTo(_2) > 0)<when> _1 String, _2 String
```

```
_1 <= _2 |-->_1 <= _2<when> _1 numeric, _2 numeric
_1 <= _2 |-->(_1.compareTo(_2) <= 0)<when> _1 String, _2 String

_1 >= _2 |-->_1 >= _2<when> _1 numeric, _2 numeric
_1 >= _2 |-->(_1.compareTo(_2) >= 0)<when> _1 String, _2 String
```

Comparison of approaches

1. *Abstract syntax to abstract syntax*: requires deep understanding of metamodels of both source + target languages, & understanding of OCL-style navigation expressions, & knowledge of target concrete syntax.

   Can be verbose, low-level & costly to write/maintain.

2. *Abstract to concrete syntax*: templates mix abstract syntax source expressions + concrete target text. Knowledge of source metamodel & target concrete syntax is needed.

   Complex navigation expressions needed. Delimiters between source/target text cause confusion.

3. *Concrete syntax to concrete syntax*: advantage that no abstract syntax knowledge needed. Navigation over source/target models is implicit, based on concrete syntax structures.

Concrete syntax to concrete syntax specification using $\mathcal{CSTL}$

- Our experience of building large code generators in Java and OCL convinced us that a more usable, agile and lightweight approach was needed.
- Defined concrete syntax transformation notation $\mathcal{CSTL}$. Can define code-generators by concrete syntax to concrete syntax mappings.
- A small language, which does not require high degree of MDE expertise.
- Scripting language – interpreted.

$\mathcal{CSTL}$ Specifications

Individual rules in $\mathcal{CSTL}$ notation have form:

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

The <*when*> clause and condition are optional.

LHS is concrete syntax in source language, RHS corresponding concrete syntax in target language.

LHS, RHS may contain variables _1, _2, ..., representing arbitrary concrete syntax fragments & their translations.

Rules grouped into source language syntactic categories, e.g., *BinaryExpression*, *Statement*.

Specialised rules are listed before more general rules.

Example: UML to C

```
Type::
Integer |-->int
Real |-->double
Boolean |-->unsigned char
String |-->char*
Set(_1) |-->_1*
Sequence(_1) |-->_1*
_1 |-->struct _1*<when> _1 Class

Attribute::
_1 : _2; |-->  _2 _1;\n
_1 : _2; _3 |-->  _2 _1;\n  _3
```

Example: UML to C

```
Class::

class _1 { _2 } |-->struct _1\n{_2};

class _1 extends _2 { _3 } |-->
struct _1\n{ struct _2* super;\n_3};
```

Example: UML to C

These rules translate class declaration

```
class Customer extends Person
{ name : String;
  age : Real;
}
```

into:

```
struct Customer
{ struct Person* super;
  char* name;
  double age;
};
```

$\mathcal{CSTL}$ Specifications

Stereotypes of LHS model elements can be used as conditions, e.g.:

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

for UML to Swift.

Specification in file $f.cstl$ can be invoked on element $\_i$ from a rule by

```
_i`f.cstl
```

Eg., could have separate specifications to produce header and code files for C.

$\mathcal{CSTL}$ Applications

- $\mathcal{CSTL}$ used for App synthesis (Android and iOS) from UML:
  - Translation of UML to Java (UML2Java8)
  - Translation of UML to Swift 4/5 (UML2Swift)
- $\mathcal{CSTL}$ is also provided as part of Eclipse Agile UML toolset.

$\mathcal{CSTL}$ *Applications*

Rules from UML2Java8:

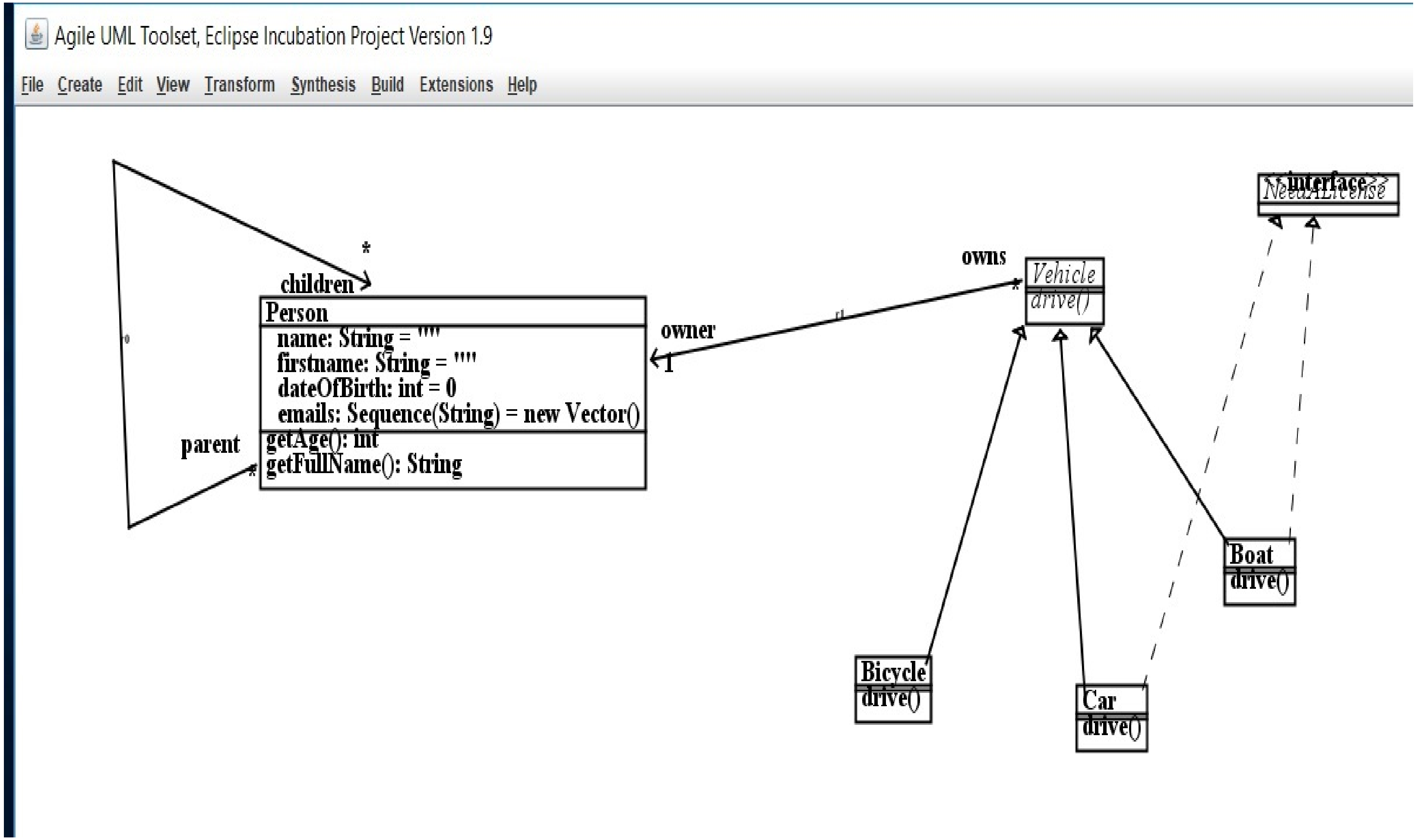
```
BinaryExpression::
_1 & _2 |-->_1 && _2
_1->count(_2) |-->Collections.frequency(_1,_2)
_1->select(_2 | _3) |-->Ocl.selectSet(_1,(_2)->{return _3;})
    <when> _1 Set

_1->includes(_2) |-->_1.contains(_2)
_1->includesAll(_2) |-->_1.containsAll(_2)
```

A Java 8 library of OCL functions, Ocl.java, defines implementation of some operators, such as $\rightarrow$*select*.

Evaluation

UML to 3GL code generators specified using different approaches:

Generator	*Implemented*	*Size*	*MaxES*	*Scope*
UML2C++	Java	18,100	–	Behaviour from OCL
UML2Java	Acceleo/ Java	3,957	27	Outline behaviour
UML2Java	EGL	1,425	35	Statemachine behaviour
UML2Java8	$\mathcal{CSTL}$	426	11	Behaviour from OCL
UML2Swift	$\mathcal{CSTL}$	398	5	Behaviour from OCL



Acceleo example model

Performance Comparison

Model	*Size*	*Acceleo*	$\mathcal{CSTL}$ *(UML2Java8)*	*Java (UML2Java4)*
1	25	480ms	45ms	28ms
2	50	750ms	70ms	47ms
4	100	800ms	36ms	37ms
10	250	1.12s	80ms	79ms
15	375	1.52s	104ms	194ms

Comparison of development effort

Approach/Generator	*Effort*
Acceleo/Java: UML2Java	3000+ hours
Java: UML2C++	2170 hours
OCL: UML2C	1375 hours
$\mathcal{CSTL}$: UML2Java8	36 hours
$\mathcal{CSTL}$: UML2Swift	50 hours

Related and future work

- Machine learning can be used to learn text-to-text mappings. But many examples needed, & result is a 'black box' with implicit rules.
- 'Sketch to code' tools convert UI sketches into UI code. Could also convert hand-drawn UML models into formal versions.
- Low-code approaches, template-based or data-based app builders, e.g., Microsoft PowerApps or Google AppSheet.

Specifically for $\mathcal{CSTL}$, will define additional tool support, & investigate how quality of generated code can be ensured.

Conclusion

- Introduced $\mathcal{CSTL}$, a declarative scripting language for writing code generators
- Enables simpler and more concise code-generator specifications, compared to existing approaches.
- Showed that this can produce smaller and more efficient code generators for UML to Java transformation.
- Development & maintenance costs also reduced.