Introduction to Developing Domain Specific Languages with XMF

Tony Clark

Ceteva Ltd (www.ceteva.com)

TOOLS Europe 2008

Outline

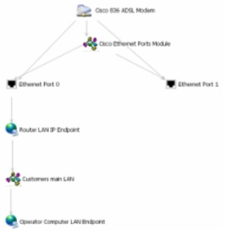
1. Domain Specific Languages: technologies; approaches.
2. XMF
3. Case Study
4. Specification
5. Implementation: syntax and semantics.
6. Variations:
    1. Static Checking
    2. Dynamic Checking
    3. Arbitrary Expressions (guards)
    4. A Control Language
    5. Parameters
    6. XML Generation
    7. Code Generation
7. Conclusion

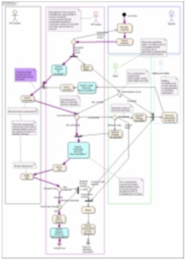
Domain Specific Languages

- Languages are the essential technology of Software Engineering.
- One size fits all - problems:
  - Large and complex (Ada, UML)
  - Lowest common denominator (UML)
  - Not accessible to non-experts.
- Domain Specific Languages (DSLs):
  - Direct representation of domain concepts.
  - Reduces the 'representation chasm'.
  - Increased quality through domain specific checking.
  - Increased utility through domain specific processing.
  - (Potentially) accessible to non-experts.

Types of DSL

	Graphical	Textual

Declarative		`network Local`
		`  modem Cisco 836`
		`    port = P0`
		`  end`
		`end`
		`network Customer`
		`  ethernet`
		`    port = P1`
		`  end`
		`end`
		`connection`
		`  P0 = P1`
		`end`

Executable		`process Publishing`
		`  action DevelopProposal`
		`  action SubmitProposal`
		`  if IsComplete`
		`  then`
		`    action …`
		`  else`
		`    action …`
		`  …`
		`end`

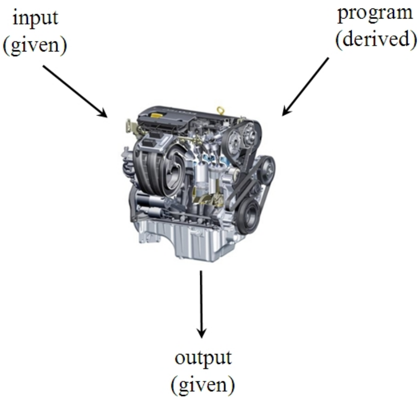
DSL Technologies

- UML and Model Driven Architecture (MDA).
- Eclipse, EMF and GMF
- Visual Studio (Software Factories).
- MetaEdit+
- OpenArchitectureWare
- XMF, XMF-Mosaic
- Ruby, Java
- Roll-your-own.

Executable Textual DSLs: A Method

Process

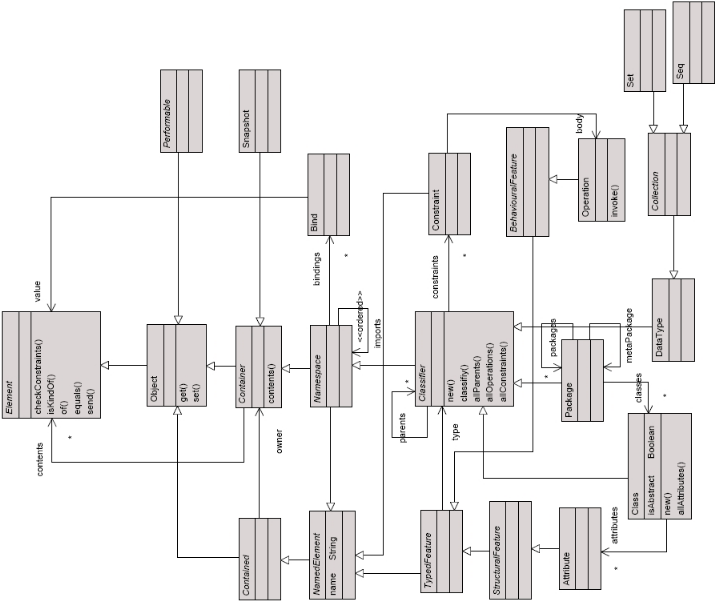
1. Model input and output.
2. Derive abstract program model via engine-as-relation.
3. Flatten relationship to produce step-by-step engine.
4. Introduce any implementation and efficiency features.
5. Design concrete representation for program.
6. Proceed with:
    1. Direct implementation.
    2. Export to external engine.
    3. Code generation.



XMF

- An Engine for Model/Language Driven Development.
- Consists of:
  - VM in Java.
  - XCore (equivalent of MOF).
  - OCL (90% complete).
  - XOCL (compiles to VM in terms of XCore).
  - Grammars for LOP.
  - Features: XML generators; XML parsers; Model Walkers; Daemons; Java interface; Ecore interface; Code Generation.

XCore


Element
checkConstraints()
isKindOf()
of()
equals()
send()
Object
get()
set()
Container
contents()
Namespace
Classifier
new()
classify()
allParents()
allOperations()
allConstraints()
Package
DataType
Class
isAbstract Boolean
new()
allAttributes()
Attribute
StructuralFeature
TypedFeature
NamedElement
name String
Contained
Performable
Snapshot
Bind
Constraint
BehaviouralFeature
Operation
invoke()
Collection
Set
Seq
value
contents
owner
bindings
<<ordered>>
imports
constraints
parents
type
packages
metaPackage
classes
attributes
body

Case Study

- Example taken from Martin Fowler.
- Presentation at JAOO 06:
  `www.infoq.com/presentations/domain-specific-languages`
- Simple problem: process raw customer, product and transaction data.
- Data format changes regularly.
- Many different formats in raw input.
- Fowler argues raw code, libraries and frameworks are not optimal.
- Use a DSL approach.

Example

Input Data

```
CSTM001Alan Brown
PROD001100.99
CSTM002Alan Smith
PROD002001.99
TRAN001001999
TRAN001001000
```

Result Data

```
Customer[
  id=1,
  name=Alan Brown]
Product[
  id=1,
  price=100.99]
Customer[
  id=2,
  name=Alan Smith]
Product[
  id=2,
  price=1.99]
LegalTransaction[
  custId=1,
  prodId=1,
  amount=999]
NullTransaction[
  custId=1,
  prodId=1,
  amount=0]
```

Specification: Approach

- Need to define a DSL that specifies how to turn raw data into records.
- An executable DSL can be defined as a 3-way relationship.
- Set up a representation for the 2 knowns.
- Use relationship definitions to force a representation for the unknown.
- Analyse relationship for a suitable machine definition.

Specification: Representation

- Sequences:
  - empty: []
  - elements: $[e_1, \ldots]$
  - concatenation: $s_1 + s_2$
- Sets:
  - empty: $\emptyset$
  - elements: $\{e_1, \ldots\}$
  - union: $s_1 \cup s_2$
- Input lines:
  - $t(s)$, example: PROD(001100.99).
  - input is sequence of lines.
- Strings: $s[i, j]$, example: 001100.99[0,2] is 001.
- Tables:
  - sets of maplets $k \mapsto v$.
  - look-up in table t by $t(k)$.
- Records: $r(F)$ type r and maplets F.

Specification: Inducing the Program

- Specification is a relationship: $input, program \vdash result$
- Using representations for input and result, induce the program.
- Typical requirement:

$$PROD001100.99, ??? \vdash Product(\{id \mapsto 001, cost \mapsto 100.99\})$$

- What are the variable parts?
- Need program to map:
  - input tag to record type
  - specific fields of input string
  - name each field.
- Therefore, we might have:

$$??? = Map(PROD \mapsto Product, id \mapsto [0,2], cost \mapsto [3,8])$$

Specification: Definition

Specify execution for a singleton mapping:

$$[t(s)] + i, \Sigma, [n] \vdash [r(\bigcup_{f \mapsto [i,j] \in F} f \mapsto s[i,j])], i$$

$$\textbf{when} \quad \Sigma(n) = Map(t \mapsto r, F)$$

Specify execution for the degenerate case:

$$i, \Sigma, [] \vdash [], i$$

Specify execution for a sequence of mappings:

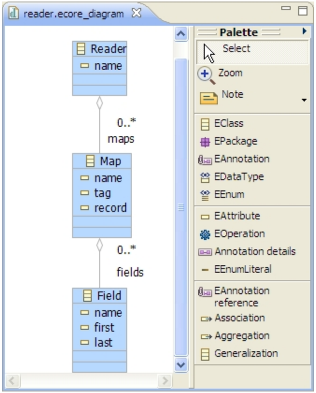
$$\frac{\begin{array}{c} i_1, \Sigma, p_1 \vdash r_1, i_2 \\ i_2, \Sigma, p_2 \vdash r_2, i_3 \end{array}}{i_1, \Sigma, p_1 + p_2 \vdash i_3}$$

Flattening

$$
\begin{aligned}
perform([t(s)] + i, [n] + p) &= [r(\bigcup_{f \mapsto [i,j] \in F} f \mapsto s[i,j])] + perform(i, p) \\
&\quad \textbf{when} \quad \Sigma(n) = Map(t \mapsto r, F) \\
perform([], []) &= []
\end{aligned}
$$

Abstract Syntax and semantic domain

- EMF can be used with XMF.
- Generate the EMF model.
- Input model into XMF.
- Export model as XMF code.
- XMF export to EMF too.



Working with EMF

```
[1] XMF> importer := Import::Ecore::EcoreImporter();
<Root>
[1] XMF> importer.importFile("@reader1/model",
                             "reader.ecore");
<Package readers>
[1] XMF> importer.initAll();
true
[1] XMF> importer.deployAll("c:/tmp");
true
[1] XMF>
```

Deploy EMF to XMF

```
parserImport XOCL;
import XOCL;
import readers;
context ! readers
  @Class Reader extends Syntax, Container, NamedElement
    @Attribute maps : Seq(Map) (?,!,+,-) end
    // A named-element uses symbols as names. The
    // easy way to force a symbol is to use setName...
    @Constructor(name) !
      self.setName(name)
    end
    // A container must implement add and contents.
    // We will just be adding Maps to a reader...
    @Operation add(e:Element)
      if e.isKindOf(Map)
      then self.addToMaps(e)
      else super(e)
      end
    end
    // Return the contents of the container...
    @Operation contents()
      maps
    end
  end
```

XMF Models

- An XMF program is a collection of definitions in files.
- Definitions are named elements added to a name-space.
- Definitions may be packages, classes, operations etc.
- Typical example:

```
context Root
  @Class C
    @Attribute a : String end
  end
```

Manifests

- Programs are built and loaded via manifests.
- A Manifest lists the files and sub-program units.
- A manifest has the following structure:

```
@Manifest ManifestName
  @File file      end
  @Ref  subModule end
do
  someOptionalAction
end
```

Records Manifest

```
parserImport XOCL;
parserImport Manifests;
@Manifest readers
  p = @File readers end
  @File Field end
  @File Map end
  @File Reader end
do p
end;
```

Concrete Syntax

- XMF provides a grammar language XBNF.
- Classes can define a grammar to become a syntax-class.
- Reference the class via @ in programs.
- parserImport is used to import all syntax-classes in a name-space.
- DEMO: simple expression language

Expressions Grammar

```
context Root
  @Class MyLang
    @Grammar
      MyLang ::=
        a = Atom (Tail^(a) | { a }).
      Tail(left) ::=
        '+' right = MyLang { left + right }
      | '*' right = MyLang { left * right }.
      Atom ::= Int | '(' a = MyLang ')' { a }.
    end
  end
```

Expressions Parsing

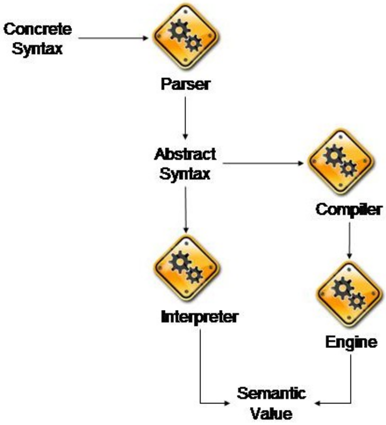
```
MyLang.grammar.parseString("1");
MyLang.grammar.parseString("1 + 2");
MyLang.grammar.parseString("1 + 2 * 5");
MyLang.grammar.parseString("(1 + 2) * 5");
```

Performable Syntax

- XMF execution process.
- Parser is hi-jacked using @.
- DEMO: modify MyLang to produce syntax.

- Pretend to use concrete syntax using quasi-quotes.
- DEMO: modify MyLang to use quasi-quotes.

- Protect abstract syntax using XOCL::Syntax
- DEMO: implement own AST and perform.



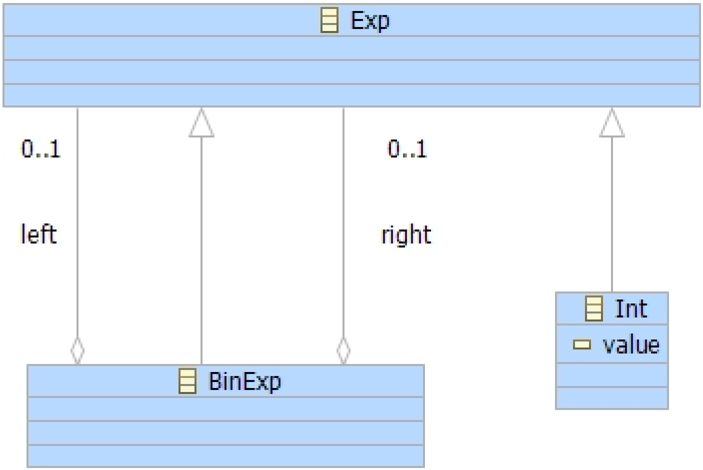
Expressions Parser Modification (1)

```
import OCL;
context Root
  @Class MyLang
    @Grammar
      MyLang ::= e = Exp 'end' { e }.
      Exp ::=
        a = Atom (Tail^(a) | { a }).
      Tail(left) ::=
        '+' right = Exp { BinExp(left,"+",right) }
      | '*' right = Exp { BinExp(left,"*",right) }.
      Atom ::=
        i = Int { IntExp(i) }
      | v = Name { Var(v) }
      | '(' a = Exp ')' { a }.
    end
  end
```

Expressions Parser Modification (2)

```
context Root
  @Class MyLang
    @Grammar
      MyLang ::= e = Exp 'end' { e }.
      Exp ::=
        a = Atom (Tail^(a) | { a }).
      Tail(left) ::=
        '+' right = Exp { [| <left> + <right> |] }
      | '*' right = Exp { [| <left> * <right> |] }.
      Atom ::=
        i = Int { i.lift() }
      | v = Name { Var(v) }
      | '(' a = Exp ')' { a }.
    end
  end
```

Expressions Abstract Syntax



Expressions Parser Modification (3)

```
import exps;
context Root
  @Class MyLang
    @Grammar
      MyLang ::= e = Exp 'end' { e }.
      Exp ::=
        a = Atom (Tail^(a) | { a }).
      Tail(left) ::=
        '+' right = Exp { BinExp(left,"+",right) }
      | '*' right = Exp { BinExp(left,"*",right) }.
      Atom ::=
        i = Int { exps::IntExp(i) }
      | '(' a = Exp ')' { a }.
    end
  end
```

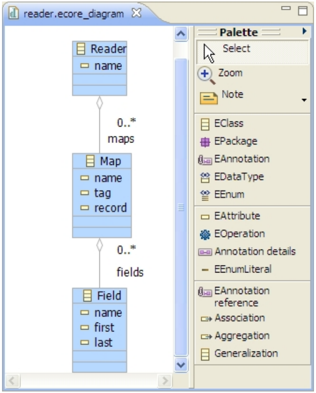
Writing an Interpreter

```
context Exp
  @AbstractOp perform():Integer end
context BinExp
  @Operation perform():Integer
    @Case op of
      "+" do
        left.perform() + right.perform()
      end
      "*" do
        left.perform() * right.perform()
      end
    end
  end
context IntExp
  @Operation perform()
    value
  end
```

Case Study: Concrete Syntax

- Look at the structure of the abstract syntax.
- What are the containership relationships?
- What needs to be named?
- Single or multiple language features?
- What 'noise' will make it easy to read?
- Will it be owned by its containing structure?
- How does the enclosing binding context fit in?
- What is being *defined* and what is being *used*?

Reader Concrete Syntax




```
@Reader X
  @Map Customer(CSTM -> Customer)
    id = [4,6]
    name = [7,20]
  end
  @Map Product(PROD -> Product)
    id = [4,6]
    price = [7,12]
  end
  @Map Transaction(TRAN -> Transaction)
    custId = [4,6]
    prodId = [7,9]
    amount = [10,12]
  end
end
```

Reader Grammar

```
parserImport XOCL;
parserImport Parser::BNF;
context Reader

  @Grammar extends OCL::OCL.grammar

    // The reader grammar inherits from the OCL grammar to
    // reference Exp. This allows the @Map ... end language
    // construct to be nested inside a @Reader ... end
    // construct.

    // A reader language construct just parses the contents
    //and add them to a newly created reader object...

    Reader ::= n = Name maps = Exp* 'end' {
      maps->iterate(m r = Reader(n) |
        r.add(m))
    }.

  end
```

Map Grammar

```
parserImport XOCL;
parserImport Parser::BNF;
context Map

  @Grammar

    Map ::= n = Name '(' t = Name '->' r = Name ')'
      fields = Field* 'end' {
        fields->iterate(f m = Map(n,t,r) |
          m.add(f))
      }.

    Field ::= n = Name '=' '[' s = Int ',' e = Int ']' {
      Field(n,s,e)
    }.

  end
```

Execution Engine

Execution engine must implement the specification.

- Input text is provided by a Data class.
- Singleton rule is implemented by Map::process and Field::process.
- Sequence of maps is implemented by Reader::processMaps.

Visit Reader aspect

- Derive input from a file using Reader::processFile.

Variation 1: Static Checking

- Programs that are syntactically correct may have semantic errors.
- Two types of semantic error:
  - static
  - dynamic
- Example static errors: $Map(t \mapsto r, f \mapsto [3,1], f \mapsto [4,10])$
- Specifications: $\forall Map(t \mapsto r, F) \in \Sigma$:
  - $\forall f \mapsto [i,j] \in F \bullet i < j$
  - $\forall f_1 \mapsto [i,j], f_2 \mapsto [m,n] \in F \bullet f_1 = f_2 \implies i = m \wedge j = n$

XMF Constraints

- Static checks are implemented as XMF constraints.
- What can you do with XMF constraints?
  - Check an object satisfies some conditions.
  - Produce a test report.
  - Produce an HTML report.
- Basic structure:

```
context SomeClass
  @Constraint Name
    OCLExpression
  fail StringExpression
  end
```

OCL in XMF

- OCL is an OMG standard. XMF implements most of OCL.
- OCL consists of:
    - basic expressions.
    - iteration: collect; select; reject; iterate.
    - quantification: forAll; exists

Case Study Constraints

- Define constraints as an aspect on model.
- Use `X.checkConstraints()` to produce a constraint report.
- Check satisfaction using `ConstraintReport::satisfied`.
- Get text report using `ConstraintReport::reportString()`
- Get HTML report using `ConstraintReport::writeHTML(path:String)`

Variation 2: Dynamic Checking

- Extend the language with typed fields:

$$[t(s)] + i, \Sigma, [n] \vdash [r(\bigcup_{f \mapsto^{\tau} [i,j] \in F} f \mapsto \tau(s[i,j]))], i$$

$$\textbf{when} \quad \Sigma(n) = Map(t \mapsto r, F)$$

- Rest of specification/flattening is same.
- Type errors occur when $\tau(s)$is not defined.
- Changes to DSL:
  - Modify Field to include type.
  - Modify Field grammar to parse type.
  - Modify engine to perform conversion.
  - Throw exception when error occurs.

Exceptions

- try-catch in XMF
- Exceptions package with some examples.
- Defining your own.
- What happens when you throw them.

Checking Types

- Modify:
  - Data to perform field conversion.
  - Field to have a new component - type.
  - Map language construct.
- DEMO.

Variation 3: Adding Guards

- Specification:

$$[t(s)] + i, \Sigma, [n] \vdash [r(e)], i$$

$$\textbf{when} \quad \begin{array}{l} \Sigma(n) = Map(t \mapsto r, F, g) \\ e = \bigcup_{f \mapsto^{\tau} [i,j] \in F} f \mapsto \tau(s[i,j]) \\ g(e) \end{array}$$

- Modifications to engine:
  - Need a language for boolean expressions.
  - Can now have multiple mappings with the same name.
  - Distinguish between mappings based on guard satisfaction.

Expressions and Evaluation

- XCore::Performable interface:

```
compile(env:Compiler::Env,frame:Integer,isLast:Boolean,saveSource:Boolean):Seq(Instr)
eval(target,env:Seq(Binding),imports:Seq(NameSpace))
FV():Set(String)
```

- Environments represented as sequences of pairs.
- Use Exp from OCL grammar to embed expressions in languages.

Case Study Modification

- Add guards to Map language construct.
- Implement a Map::satisfied operation.
- Use Map::satisfied to select the correct mapping in Reader.

Variation 4: A Control Feature

- Programs often consist of definitions and some control.
- We have mapping definitions.
- Our control is currently a sequence of mapping names.
- More useful to allow:
  - Optional formats in input.
  - Arbitrary repetition.
  - Sequences of control.

Specification: Optional Input Formats

- Want to achieve the following for the same program p:

$$[t(s)] + i, \Sigma, p \vdash [r(e)], i$$

$$[t'(s')] + i, \Sigma, p \vdash [r'(e')], i$$

- Generalize to:

$$i_1 + i_2, \Sigma, p \vdash r, i_2$$

$$i_1' + i_2, \Sigma, p \vdash r', i_2$$

- Design a construct that contains two alternative controls.

Specification: Repetition

- Want to achieve a sequence of arbitrary length:

$$[t(s)] + i, \Sigma, p \vdash [r(e)], i$$

$$[t(s), t(s')] + i, \Sigma, p \vdash [r(e), r(e')], i$$

$$[t(s), t(s'), \ldots] + i, \Sigma, p \vdash [r(e), r(e'), \ldots], i$$

- Degenerate case:

$$[] + i, \Sigma, p \vdash [], i$$

- Design a construct that consumes sequences of input.

Specification

Specify execution for a singleton mapping:

$$[t(s)] + i, \Sigma, n \vdash [r(e)], i \qquad \textbf{when} \quad \ldots as \quad before$$

Specify execution for the degenerate case:

$$i, \Sigma, \epsilon \vdash [], i$$

Specify execution for a sequence of mappings:

$$\frac{\begin{array}{c} i_1, \Sigma, p_1 \vdash r_1, i_2 \\ i_2, \Sigma, p_2 \vdash r_2, i_3 \end{array}}{i_1, \Sigma, p_1 p_2 \vdash i_3}$$

Alternatives:

$$\frac{i_1,\Sigma,p_1 \vdash r, i_2}{i_1,\Sigma,p_1 | p_2 \vdash r, i_2} \qquad \frac{i_1,\Sigma,p_2 \vdash r, i_2}{i_1,\Sigma,p_1 | p_2 \vdash r, i_2}$$

Abstract Syntax

- Need a new language structure.
- Define an EMF model for the commands:
  - And
  - Star
  - Or
  - Call
  - OK

Command Engine

$$
\begin{array}{lcl}
process(ok, i, s, f) & = & s([], i, f) \\
\\
process(n, [t(s)] + i, s, f) & = & s([r(e)], i, f) \quad \textbf{when} \quad \ldots \\
 & = & f() \quad \textbf{otherwise} \\
\\
process(c_1 c_2, i, s, f) & = & process(c_1, i, s', f) \\
 & & \quad \textbf{where } s'(v_1, i, f) = \\
 & & \quad\quad process(c_2, i, s'', f) \\
 & & \quad\quad\quad \textbf{where } s''(v_2, i, f) = \\
 & & \quad\quad\quad\quad s(v_1 + v_2, i, f) \\
\\
process(c_1, \mid c_2, i, s, f) & = & process(c_1, i, s, f') \\
 & & \quad \textbf{where } f'() = process(c_2, i, s, f) \\
\\
process(c^*, i, s, f) & = & process((cc*) \mid ok, i, s, f)
\end{array}
$$

Backtracking

- Alternatives require a new execution mechanism:
  - Try one alternative.
  - If it fails, try the other *from the same point in the input*.
- Modify Data to support backtracking.

Execution Engine

- Implement engine in XMF

Concrete Syntax

- A new language construct that can be embedded in XMF programs:

```
context Root
  @Operation test1()
    @Read(X <- "@reader/tests/test1.txt")
      (Customer | Product | Transaction)*
    end
  end
```

Variation 5: Parameters and Locals

- Add Parameters to Mapping Definitions.
- Exercise left to reader.

Variation 6: Generating XML

- Use the PrintXML language feature:

```
parserImport XML::PrintXML;
...
  @XML(out)
    <TAG ATTS>
      ...
    </TAG>
  end
...
```

XML Generation for Reader

```
parserImport XOCL;
parserImport XML::PrintXML;
import readers;
import IO;
context Reader
 @Operation toXML(out:OutputChannel)
   @XML(out)
     <Reader name=name>
       @For map in maps do
         map.toXML(out)
       end
     </Reader>
   end
 end
```

Variation 7: Code Generation

- Generate Framework Code from Reader Definition.
- Exercise left to reader.

Conclusion

- XMF is an engine for modelling/prototyping systems.
- XMF supports a Language Oriented Approach.
- Domain Specific Languages offer advantages.
- XMF is open-source (www.ceteva.com).
- Many examples described in Superlanguages: Developing Languages and Applications with XMF.