Tutorial of the ATL transformation language
http://github.com/jesusc/atl-tutorial

Part III

THE ATL LANGUAGE (CONT'D)

jesus.sanchez.cuadrado@gmail.com
@sanchezcuadrado
http://sanchezcuadrado.es

Outline

- Less known things about ATL, like
  - Map & Tuple data types
  - The execution algorithm
  - Details about rule execution and resolution
  - Imperative code
  - Some patterns and anti-patterns

  - An several other tricky things...

The ATL language

More about data types

Map

- Associative table
  - Keys – Any object
  - Values – Any object
  - Similar to Java maps
- Syntax
  - Type declaration:

```
Map(String, Sequence(CD!Feature))
```

Key type: `String` — Value type: `Sequence(CD!Feature)`

  - Initialization:

```
Map { }

Map { ('String', Sequence {}),
      ('Integer', Sequence {}) }
```

Map

- Operations
  - `get(key : oclAny)`
    - Returns the value associated with the key
    - OclUndefined if there is no key
  - including(key : oclAny, val : oclAny)
    - Inserts value associated with key
    - Returns a copy of self
  - union(m : Map)
    - Returns a map containing all self elements to which are added those elements of m whose key does not appear in self;
  - getKeys()
    - Returns a set containing all the keys of self;
  - getValues()
    - Returns a bag containing all the values of self.

Map

- Simple example
  - (More complex examples later, because Map typically goes with “iterate”)

```
helper def : eDayMap : Map(String, RSS!DayKind) =
  let m : Map(String, RSS!DayKind) =
     Map{('Mon',#Monday),('Tue',#Tuesday),('Wed',#Wednesday),
         ('Thu',#Thursday),
         ('Fri',#Friday),('Sat',#Saturday),('Sun',#Sunday)}
  in m ;
```

Tuple

- Tuples
  - A tuple type is not named.
  - A declared tuple type has to be identified by its full declaration each time it is required.
    - What can you do in ATL to avoid this burden…?

```
let var : TupleType(idx : Integer, value : String) =
         Tuple { idx = 1, value = 'something' }
  in var.idx
```

Feature access like a regular object

`aTuple.toMap` returns a map version of the tuple

Types of multi-valued features

- Which is the type of a navigation expression over a multivalued feature, like in:

```
aClass.features
```

- There are two cases:
  - Feature with isUnique = false ➡ Sequence(T)
  - Feature with isUnique = true ➡ Sequence(T) !
    - Be careful with this one, it should be Set(T)

Type of allInstances()

- Which is the type of allInstances()?
  - OrderedSet (that's fine)

- Testing the types:

```
helper def: testHelper1 : OclAny =
   CD!Class.allInstances().debug('collectionType');

helper def: testHelper2 : OclAny =
   CD!Class.allInstances()->first().features.debug('featureType'));
```

Enumerations

- The concept does not exist in ATL
  - You can only name literals

```
rule model2gui {
  from m : CD!Model
    to w : GUI!Window (
       title <- m.name,
       layout <- vflow,
       widgets <- m.classifiers
     ), g : GUI!GUI (
        windows <- w
     ), vflow : GUI!FlowLayout (
        direction <- #vertical
     )
}
```

FlowDirection.vertical

The ATL language

Rules revisited

Rules

- Matched rule
- Lazy rule
- Unique lazy rule
- Called rule
- Entry point rule
- End point rule

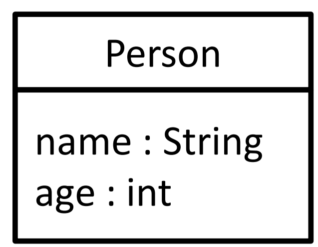
Matched rules
Resolving multiple input elements

Java-like meta-model


```java
class Person {
  public void setName(String name) {...}
  public String getName() { ... }

  public void setAge(int v) { ... }
  public int getAge() { ... }
}
```

Each pair of “get” / “set” methods is transformed into a property

UML-like meta-model



Motivation
Resolving multiple input elements

```
rule JavaClass2Classifier {
   from j : JAVA!JavaClass
     to c : CD!Class (
        name <- j.name,
        features <- j.operations ¿?
     )
}

rule get_set2attribute {
   from get : JAVA!Operation, set : JAVA!Operation (
     get.name.startsWith('get') and set.name.startsWith('set') and
     get.name.substring(3, get.name.size()) =
       set.name.substring(3, get.name.size())
   )
   to feature : CD!Property (
        name <- get.name,
        type <- get.type
   )
}
```

How can we resolve the properties created with get_set2attribute?

Matched rules
Resolving multiple input elements

Construct a tuple.
Tuple variables must match input pattern names

```
rule JavaClass2Classifier {
   from j : JAVA!JavaClass
     to c : CD!Class (
        name <- j.name,
        features <- j.operations->collect(o1 | j.operations->collect(o2 |
              thisModule.resolveTemp(Tuple {get=o1, set=o2}, 'feature'))
     )
}

rule get_set2attribute {
   from get : JAVA!Operation, set : JAVA!Operation (
     get.name.startsWith('get') and set.name.startsWith('set') and
     get.name.substring(3, get.name.size()) =
       set.name.substring(3, get.name.size())
   )
   to feature : CD!Property (
        name <- get.name,
        type <- get.type
   )
}
```

cd2gui_no_default.atl

Matched rules
Disabling creation of trace links

- **nodefault** rules
  - Avoid rule conflicts with rules compatible types
  - Cannot be resolved by bindings

```
nodefault rule model2gui {
       from m : CD!Model
       to   w : GUI!Window (
                title <- 'top-model',
                widgets <- m
       )
}

rule model2gui_frame{
       from m : CD!Model
       to   w : GUI!Frame ( name <- 'no-top-model’ )
}
```

Matched rules Question...

- **What would happen...?**

```
nodefault rule model2gui {
       from m : CD!Model
       to   w : GUI!Window (
                 title <- 'top-model’
       )
}

rule model2gui_frame{
       from m : CD!Model
       to   w : GUI!Frame (
            name <- 'no-top-model’,
            widgets <- m
        )
}
```

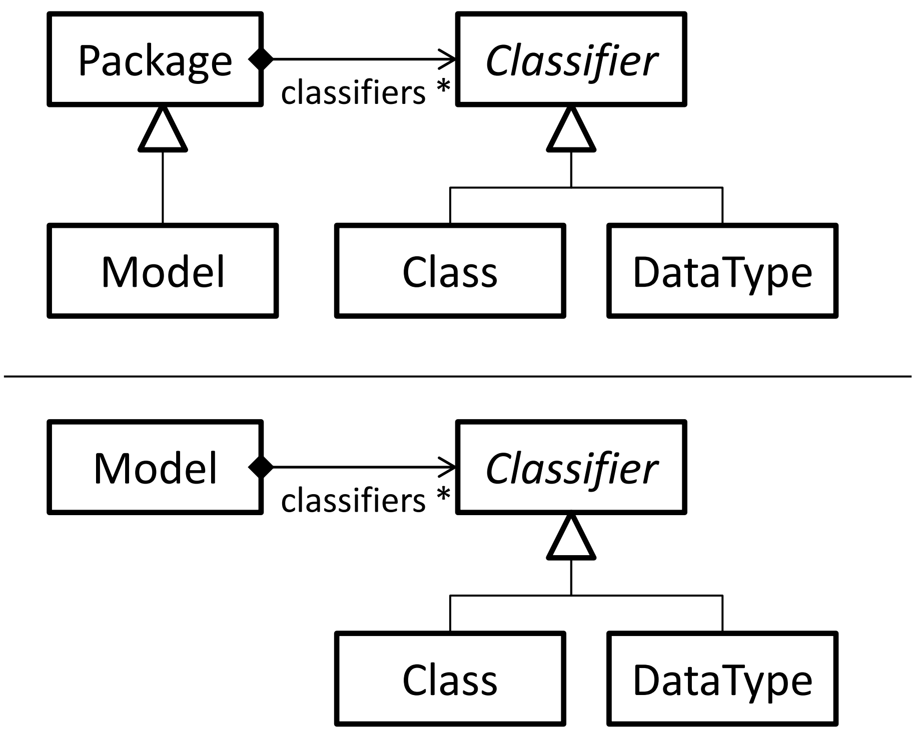
Rule inheritance

- Support for single inheritance of rules
  - Abstract rules to factorize common code
  - For matched rules
    - A means to enable dynamic dispatch of rules
  - For lazy rules
    - A means to enable rule-based pattern matching

* Wimmer, Manuel, et al. "Surveying Rule Inheritance in Model-to-Model Transformation Languages." Journal of Object Technology 11.2 (2012): 3-1.

Rule inheritance

- Solving a rule conflict
  - The super-rule is first matched, but then then subrules are tried (dynamic dispatch)




```
rule model2model
 extends package2model {
  from m1: UML!Model
  to   m2: CD!Model
}

rule package2model {
  from p: UML!Package
  to   m: CD!Model
}
```

Abstract rules

- Abstract rules
  - All abstract classes in the output pattern must be overridden in the concrete subclasses
  - To override, use the same variable name for the input/output pattern
    - Bindings for the same feature declared in the super rule are not executed.

Abstract rules

- Example. Before abstract rules, similar rules replicate code.

```
rule attribute2text {
  from a: CD!Attribute (
       a.isText() )
  to   t: GUI!Text,
       l: GUI!Label,
       g1: GUI!GridInfo (
        column <- 1,
        widget <- t
       ),
       g2: GUI!GridInfo (
        column <- 2,
        widget <- l
       )
}
```

```
rule attribute2int {
  from a: CD!Attribute (
       a.isText() )
  to   t: GUI!Text,
       l: GUI!Label,
       g1: GUI!GridInfo (
        column <- 1,
        widget <- t
       ),
       g2: GUI!GridInfo (
        column <- 2,
        widget <- l
       )
}
```

cd2gui_09.atl

Abstract rules

- With abstract rule, common patterns are factorized.

```
abstract rule attribute2widget {
  from a: CD!Attribute
    to w: GUI!Widget,
       l : GUI!Label (
          value <- a.name
       ), g1: GUI!GridInfo (
          column <- 1,
          widget <- w
       ),
       g2: GUI!GridInfo (
          column <- 2,
          widget <- l
       )
}
```

```
rule attribute2text
  extends attribute2widget {
  from a: CD!Attribute (a.isText())
   to   w: GUI!Text (
     name <- 'txt' + a.name
  )
}

rule attribute2int
  extends attribute2widget {
  from a: CD!Attribute ( a.isInt())
    to w: GUI!Text (
       name <- 'int' + a.name
     )
}
```

Abstract rules and lazy rules

- Before abstract rules, we had to pattern match explicitly to select the correct rule to call:

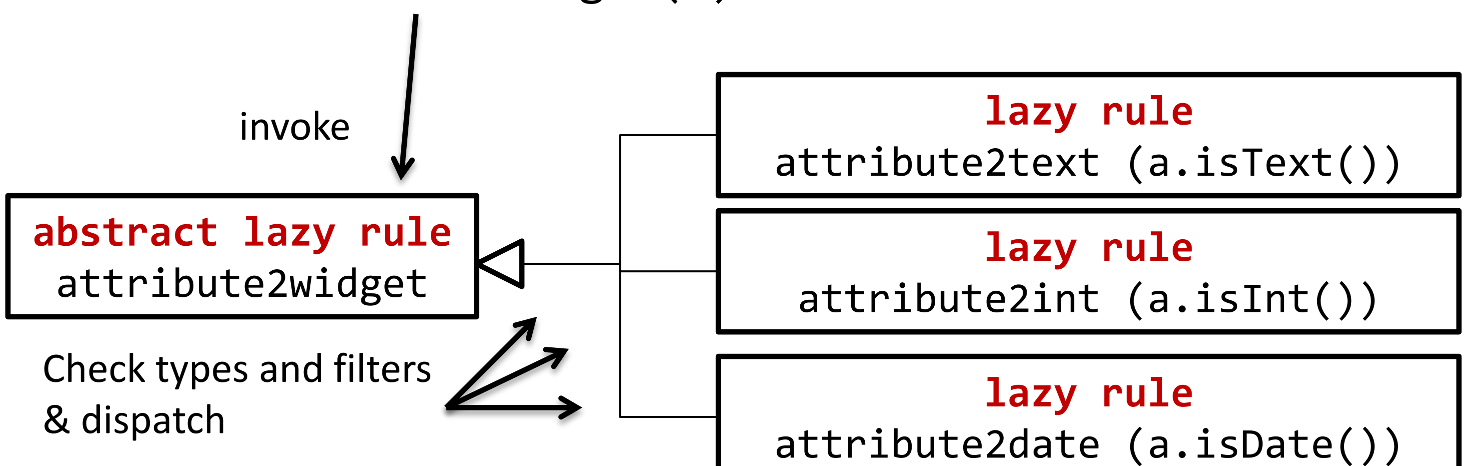
```
rule class2frame {
  from c : CD!Class ( not c.isAbstract )
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.allAttributes->collect(f |
        if f.isText()       then thisModule.attribute2text(f)
        else if f.isInt()   then thisModule.attribute2int(f)
        else if f.isDate()  then thisModule.attribute2date(f)
        else                     OclUndefined endif endif endif
     )
  )
}
```

cd2gui_10.atl

Abstract rules and lazy rules

- Now, the sub-rule satisfying the filter is dispatched:

```
rule class2frame {
  from c : CD!Class ( not c.isAbstract )
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.allAttributes->collect(f |
       thisModule.attribute2widget(f)
     )
  )
}
```



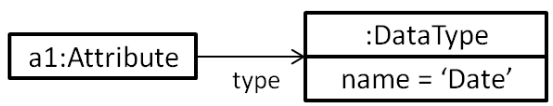
Abstract rules and lazy rules

```
lazy abstract rule attribute2widget {
  from a: CD!Attribute
  to   t: GUI!Widget
}

lazy rule attribute2text extends attribute2widget {
  from a: CD!Attribute ( a.isText() )
  to   t: GUI!Text ( name <- 'txt' + a.name  )
}

lazy rule attribute2int extends attribute2widget {
  from a: CD!Attribute ( a.isInt() )
  to   t: GUI!Text ( name <- 'int' + a.name )
}

lazy rule attribute2date extends attribute2widget {
  from a: CD!Attribute ( a.isDate() )
  to   t: GUI!DatePicker ( name <- 'date' + a.name )
}
```

Abstract rules and lazy rules

- What happen if there is no rule satisfying the filter?

```
widgets <- c.allAttributes->collect(f |thisModule.attribute2widget(f))
```




```
lazy abstract rule attribute2widget {
  from a: CD!Attribute
  to   t: GUI!Widget
}

lazy rule attribute2text extends attribute2widget {
  from a: CD!Attribute ( a.isText() )
  to   t: GUI!Text ( name <- 'txt' + a.name  )
}
```

Returns a TransientLink!

Abstract rules and lazy rules

- What happen if there is no rule satisfying the filter?
  - The result is a TransientLink object
  - The error is difficult to interpret but it is probably worth allowing ATL to notify the missing match
  - Nevertheless, if you want to ignore:

```
widgets <- c.allAttributes->
     collect(f |thisModule.attribute2widget(f))->
     reject(o | o.oclType().name = 'TransientLink'),
```

Abstract rules and lazy rules

- What happen if several rules may match?
  - The first one in the program order wins.

Binding assignment Cross-references

- By default cross references between an models are not allowed.
  - Input/output or different output models
  - ATL try to enforce the model-to-model semantics.
- When this behaviour is required, it needs to be explicitly indicated in the launch configuration.
- Example:
  - UML to CD
  - The library of primitive types for CD is a parameter of the transformation

uml2cd_cross_refs.atl

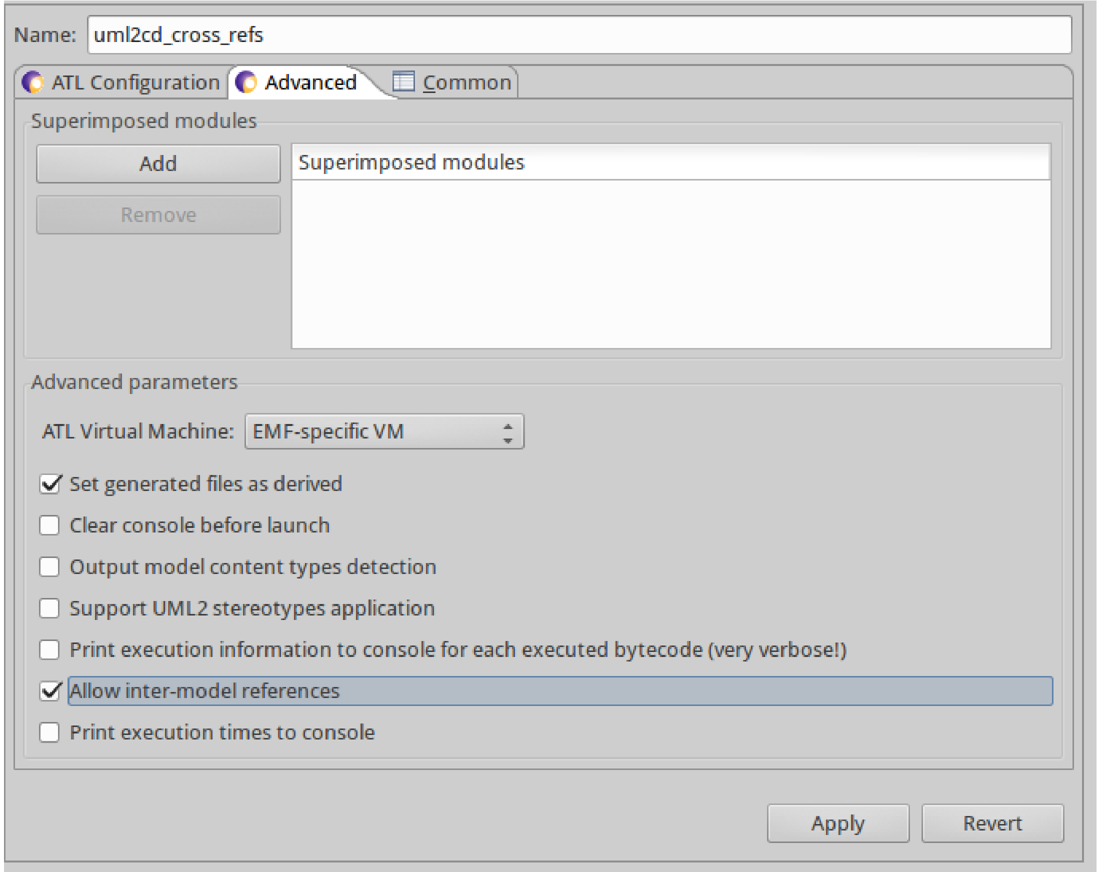
Binding assignment
Cross-references

```
module "uml2cd showing cross references";
create OUT: CD from IN: UML, CDTYPES: CD;

helper def : findDataType(name : String) : CD!DataType =
        CD!DataType.allInstancesFrom('CDTYPES')->any(d | d.name = name);

rule Class2Class {
   from m : UML!Class
     to w : CD!Class (
        name <- m.name,
        features <- m.ownedAttribute
   )
}

rule Property2Feature {
   from p : UML!Property ( p.type.oclIsKindOf(UML!DataType) )
     to f : CD!Attribute (
        name <- p.name,
        type <- thisModule.findDataType(p.type.name)
   )
}
```

This is a cross-reference

Binding assignment

- Cross references
  - Must be enabled explicitly



Imperative code

```
rule r {
  from ...
  to   ...
  do {
    ...
  }
}
```

- Written in the do section of any kind of rule.
  - It is optional.
- Executed when,
  - All target elements of the rule has been created, and
  - Bindings have been resolved.
- Sequence of statements
  - Executed one after the other

Imperative code

- Example. Generate unique widget names

```
helper def: counter : Integer = 0;

rule property2text {
  from p : UML!Property ( p.isText() )
  to t : GUI!Text {
    name <- p.name
  }
  do {
     t.name <- t.name + thisModule.counter.toString();
     thisModule.counter <- thisModule.counter + 1;
  }
}
```

Imperative code

- Generate unique widget names (alternative)

```
helper context UML!Property def: toIdx(): Integer =
  UML!Property.allInstances()->indexOf(self).toString();

rule property2text {
  from p : UML!Property ( p.isText() )
  to t : GUI!Text {
    name <- p.name + p.toIdx()
  }
}
```

Imperative code

- If statement

```
if (condition) {
  stm1; stm2; ...; stm;
}
[else {
  stm1; stm2; ...; stm;
}]?
```

- For statement

```
for(iterator in collection) {
  stm1; stm2; ...; stm;
}
```

- Assignment

```
left-expression <- right-
expression
```

Imperative code

- Acceptable uses:
  - Data preparation in entry point rules
    - E.g., initialize a module attribute
  - Local modifications of the target elements
  - Add data in a collection stored in a module attribute
    - This attribute should not be used in matched rules
  - Delayed actions applied end point rules

Entry point / End point rules

- Similar to called rules but:
  - Implicitly invoked
  - No parameters. No return value.
- Entry point rules:
  - Keyword **entrypoint**
  - Cannot invoke any "trace-related" operation like `resolveTemp`
- End point rules
  - Keyword **endpoint**

Entry point rule and imperative code

- Example. Create data types.

```
helper def : dtString  : CD!DataType = OclUndefined;
helper def : dtInteger : CD!DataType = OclUndefined;

entrypoint rule createDataTypes() {
   to str : OO!DataType ( name <- 'String' ),
      int : OO!DataType ( name <- 'Integer' ),
   do {
        thisModule.dtString <- str;
       thisModule.dtInteger <- str;
   }
}
```

Entry point rule and imperative code

- Example. Assigning grid information, was not properly solved in the previous examples. Requires:
  - Using some imperative code
  - Passing an additional element to the lazy rule

```
helper def : delayedGridInfo :
  Sequence(TupleType(src : CD!Class, tgt : GUI!GridInfo)) = Sequence { };

rule class2frame {
 from c: CD!Class ( not c.isAbstract )
   to frm: GUI!Frame (
     widgets <- c.allAttributes->collect(f |thisModule.attribute2widget(c, f))
   )}

lazy rule attribute2text extends attribute2widget {
  from  c : CD!Class, a: CD!Attribute  ( a.isText() )
  to    t: GUI!Text (
     name <- 'txt' + a.name
  )
  do {
   thisModule.delayedGridInfo <-
    thisModule.delayedGridInfo->including(Tuple { src = c, tgt = info });
  }
}

endpoint rule createGridLayouts() {
  do {
    for ( tuple in thisModule.delayedGridInfo ) {
      thisModule.resolveTemp(tuple.src, 'grid').info <- tuple.tgt;
    }
  }
}
```

The same for all inherited rules

Simpler solution...

- Simpler solution

```
rule class2frame {
 from c: CD!Class ( not c.isAbstract )
   to frm: GUI!Frame (
     widgets <- c.allAttributes->collect(f |thisModule.attribute2widget(c, f))
   )
}

lazy rule attribute2text extends attribute2widget {
  from  c : CD!Class, a: CD!Attribute  ( a.isText() )
  to    t: GUI!Text (
     name <- 'txt' + a.name
  )
  do {
   thisModule.resolveTemp(c, 'grid').info <- info
  }
}
```

The ATL language

The execution algorithm

The ATL algorithm

- Execute entry point rules
- Match phase
- Resolve phase
- Execute end point rules

F. Jouault, I. Kurtev, Transforming models with ATL,
in MoDELS Satellite Events, 2005, pp. 128–138. (long version)

The ATL algorithm – Match phase

```
ForEach standard rule R {
  ForEach candidate pattern C of R {
    -- a candidate pattern is a set of elements matching the
    -- types of the source pattern of a rule
    evaluate the guard of R on C
    If guard is true Then
       create target elements in target pattern of R
       create TraceLink for R, C, and target elements
    Else
       discard C
    EndIf
  }
}
```

F. Jouault, I. Kurtev, Transforming models with ATL,
in MoDELS Satellite Events, 2005, pp. 128–138. (long version)

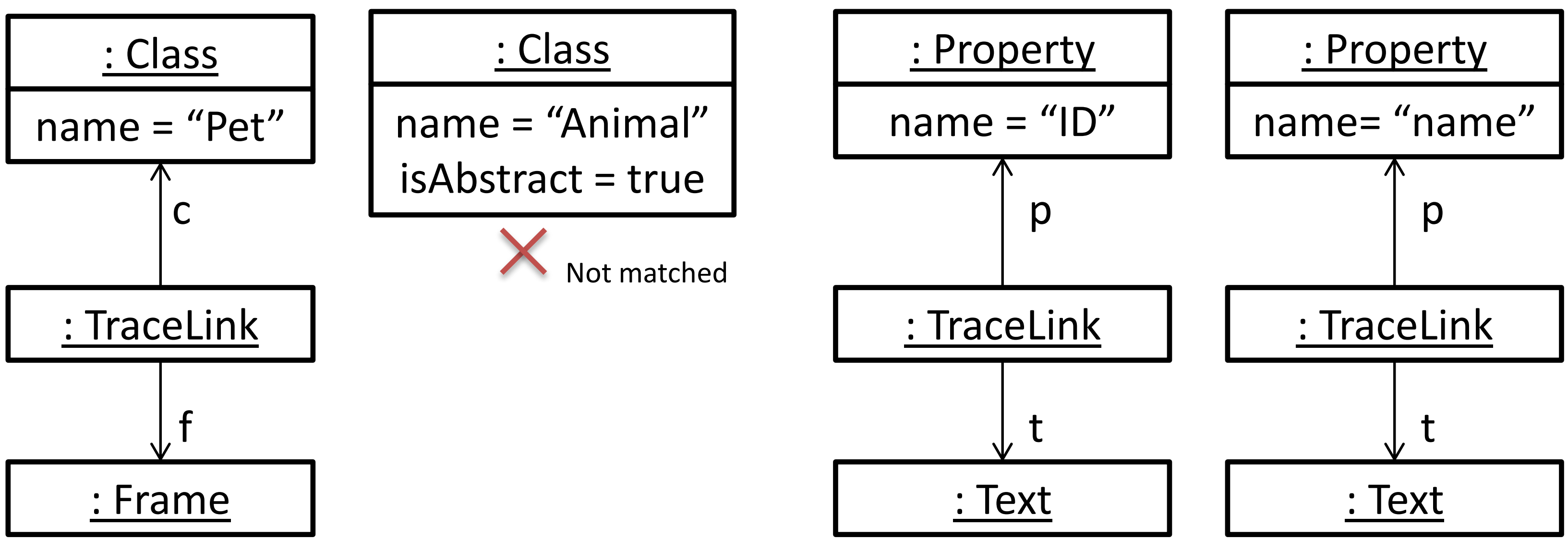
The ATL algorithm – Apply phase

```
ForEach TraceLink T {
  R = the rule associated to T
  C = the matched source pattern of T
  P = the created target pattern of T
  -- Initialize elements in the target pattern:
  ForEach target element E of P {
    -- Initialize each feature of E:
    ForEach binding B declared for E {
      expression = initialization expression of B
      value = evaluate expression in the context of C
      featureValue = resolve value
      set featureValue to corresponding feature of B
    }
  }
  execute action block of R in the context of C and T
  -- Imperative blocks can perform any navigation in C or T and
  -- any action on T. It is the programmer's responsibility
  -- to perform only valid operations.
}
```

The ATL algorithm – Match phase

```
rule class2frame {
  from c : CD!Class (not c.isAbstract)
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.ownedAttribute
  )
}
```

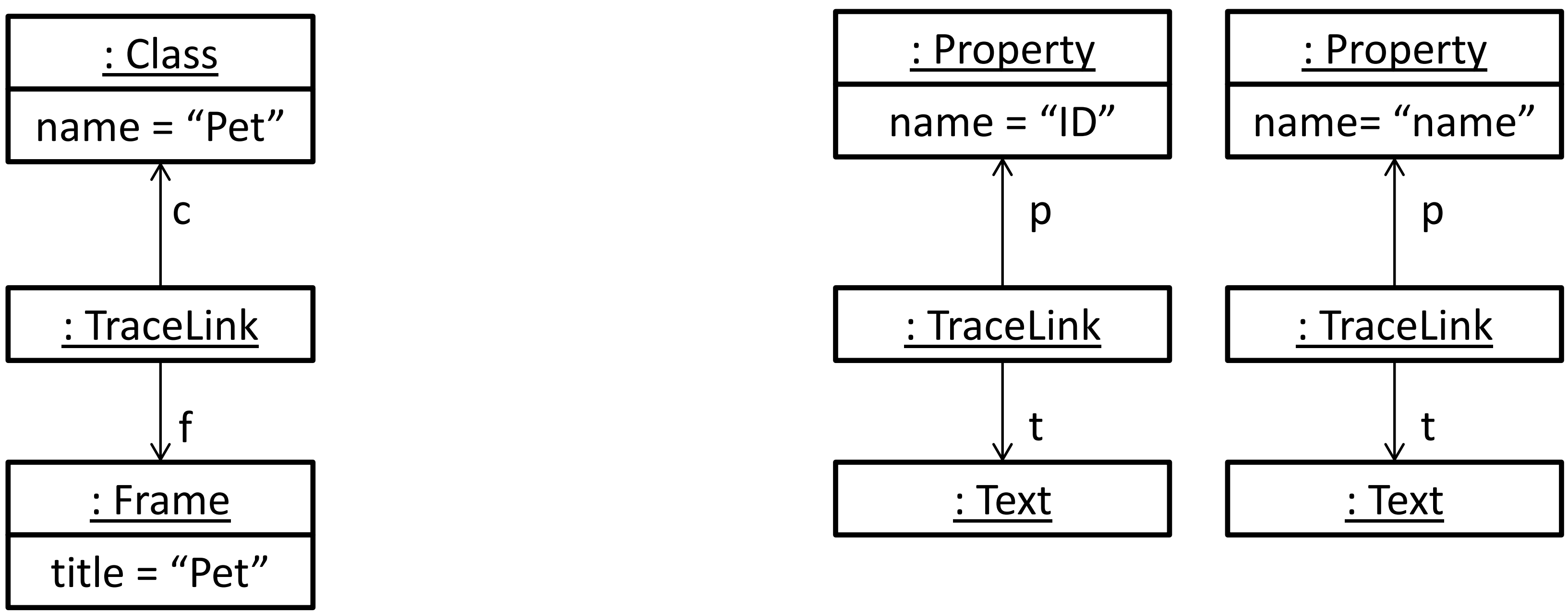
```
rule property2text {
  from p : CD!Property (p.isText())
  to   t : GUI!Text
}
```



The ATL algorithm – Apply phase

```
rule class2frame {
  from c : CD!Class (not c.isAbstract)
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.ownedAttribute
  )
}
```

```
rule property2text {
  from p : CD!Property (p.isText())
  to   t : GUI!Text
}
```

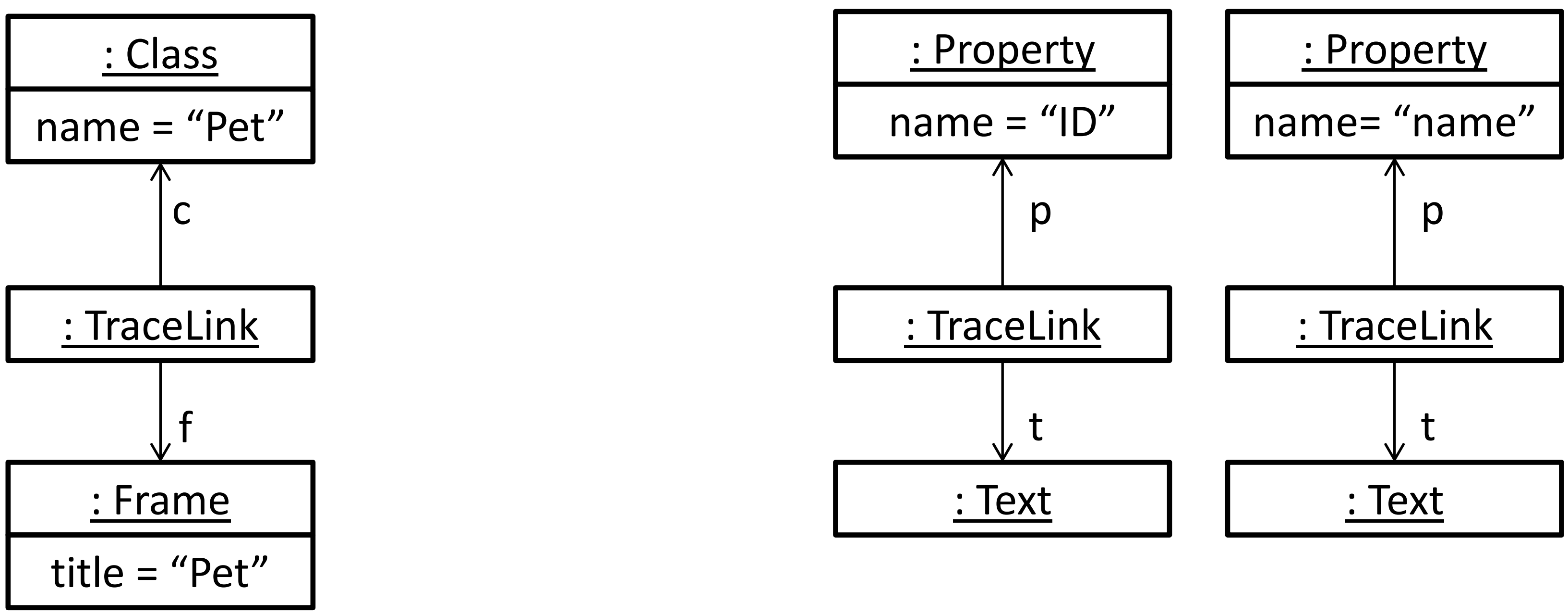


The ATL algorithm – Apply phase

```
rule class2frame {
  from c : CD!Class (not c.isAbstract)
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.ownedAttribute
  )
}
```

(1) Evaluate expression

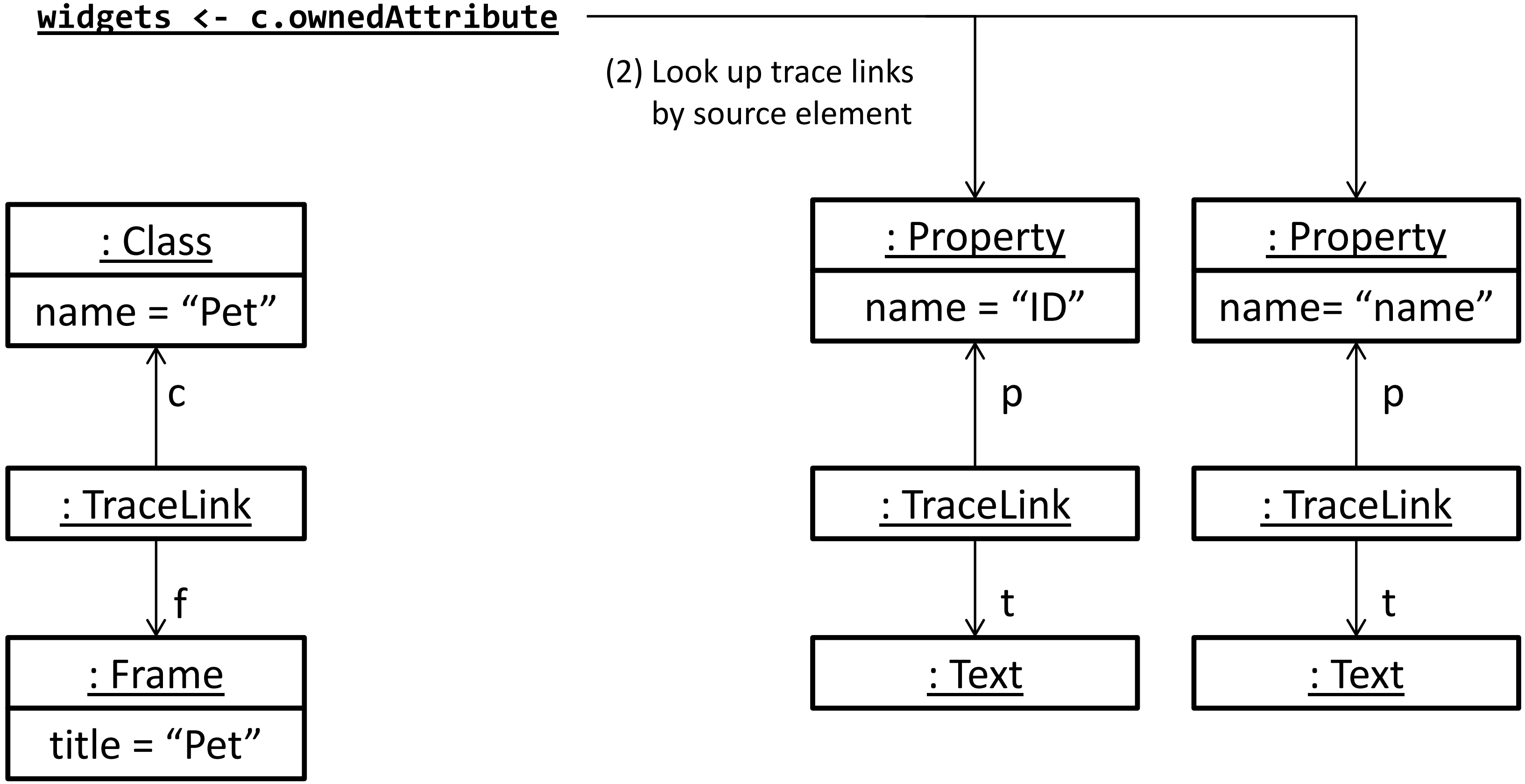
```
rule property2text {
  from p : CD!Property (p.isText())
  to   t : GUI!Text
}
```



The ATL algorithm – Apply phase

```
rule class2frame {
  from c : CD!Class (not c.isAbstract)
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.ownedAttribute
  )
}
```

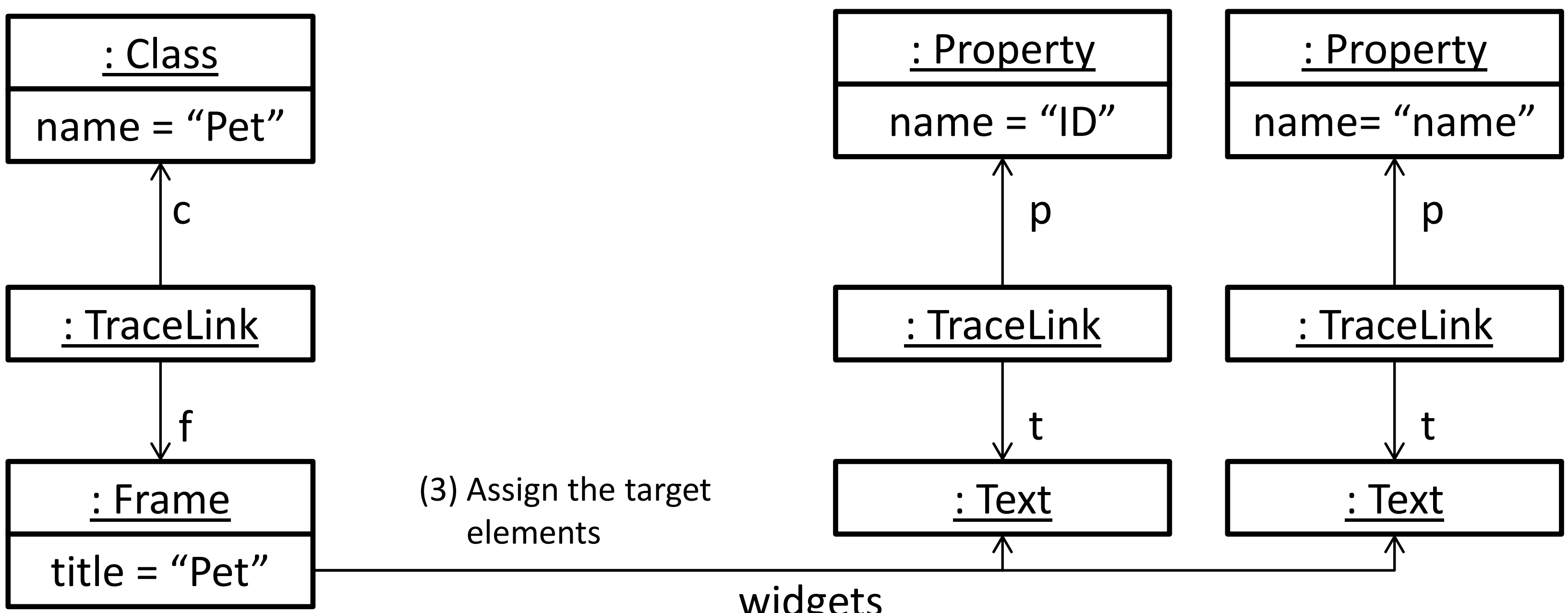
```
rule property2text {
  from p : CD!Property (p.isText())
  to   t : GUI!Text
}
```



The ATL algorithm – Apply phase

```
rule class2frame {
  from c : CD!Class (not c.isAbstract)
  to   f : GUI!Frame (
     title <- c.name,
     widgets <- c.ownedAttribute
  )
}
```

```
rule property2text {
  from p : CD!Property (p.isText())
  to   t : GUI!Text
}
```



The ATL algorithm – Apply phase

- What if look up fails?
  - i.e., there is no rule to resolve a source element
  - In the example is the `isText()` predicate is not satisfied
- By default:
  - Nothing happens in the target model
    - Message in the console for debugging purposes
    - What does this means? It depends…
    - (One needs to understand these kind of details…)

The ATL language

More on OCL

Let expressions

- There are no variables in OCL
- A **let** expression is a syntactic facility to bind an expression to a value

```
let classes : Set(CD!Class) =
    CD!Class.allInstances()->select(c | not c.isAbstract)
 in classes->size()
```

This is the result

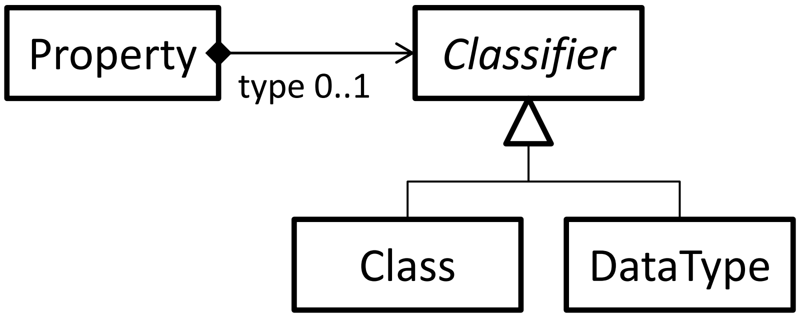
Let expressions

- Multiple let expressions can be nested

```
let classes : Set(CD!Class) = CD!Class.allInstances() in
let nonAbstract : Set(CD!Class) =
    classes->select(c | not c.isAbstract) in
let size : Integer = classes->size()
 in size
```

If expressions

- Q: Helper to check if a UML property is of type text?



- Straightforward attempt:

```
helper context UML!Property def: isText() : Boolean =
   self.type.oclIsKindOf(UML!DataType) and
   self.type.name = 'String‘;
```

If expressions

- Q: Helper to check if a UML property is of type text?
  - The problem is that there is no short-circuit in OCL.
  - Both sub-expressions of and are evaluated
- Correct implementation:

```
helper context UML!Property def: isText() : Boolean =
   if not self.type.oclIsUndefined() and
          self.type.oclIsKindOf(UML!DataType) then
      self.type.name = ‘String’;
   else
      false
   endif;
```

If expressions

- Probably a better balance:

```
helper context UML!Property def: isText() : Boolean =
        if self.type.oclIsKindOf(UML!DataType) then
                self.type.name = 'String'
        else
                false
        endif;
```

- Why this works?

Iterate

- It is a generic collection iterator
  - Equivalent to fold-left in functional languages
  - Syntax:

```
aCollection->iterate(it; acc = <init-expression> |
   <body>
)
```

Iterate

- Behaviour:
  - Iterates over the elements of the collection assigning them to the **it** variable in each iteration step.
  - At the beginning, the value of **acc** is **<init-expr>**
  - Each time, **<body>** is evaluated and the **acc** variable is updated with the result of the evaluation, so that:
    - It is passed to the next iteration, or
    - It is the final result of the operation.

Iterate

Iteration variable

Initial value for acc

```
aCollection->iterate(it; acc = <init-expression> |
   <body>
)
```

Accumulator variable

Evaluated in each iteration. Its result becomes the next value for **acc**.

Iterate

- Any built-in iterator can be imitated with iterate
  - Advice: try built-in operators before using iterate
- aCollection->collect(it | <body>)

```
aCollection->iterate(it, acc = Sequence {} |
   acc->including(<body>)
)
```

Iterate

- aCollection->select(it | <body>)

```
aCollection->iterate(it, acc = Sequence {} |
   let body_result : Boolean = <body>
   if body_result then
       acc->including(it)
   else
       acc
   endif)
```

Iterate

- aCollection->exists(it | <body>)

```
aCollection->iterate(it, acc = Boolean |
   let body_result : Boolean = <body>
   if body_result then
       true
   else
       false
   endif)
```

Iterate

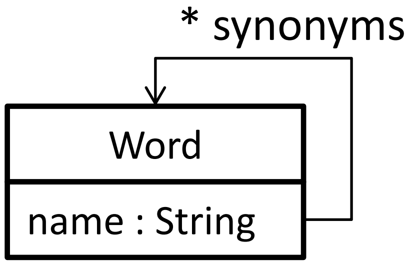
- aCollection->exists(it | <body>)

```
aCollection->iterate(it, acc = Boolean |
   let body_result : Boolean = <body>
   if body_result or acc then
       true
   else
       false
   endif)
```

Correct version

Iterate + Map

- Example
  - Parameterize CD2GUI with a dictionary of synonyms.
  - Simple meta-model:



  - “Slow” lookup:

```
helper def : isSynonym(word : String, syn : String) : Boolean =
   DICT!Word.allInstances()->exists(w |
     if w.name = word then w.synonyms->exists(w2|w2.name = syn)
     else false endif );
```

Iterate + Map

- Example
  - We can pre-compute for fast look up *

```
helper def : syns : Map(String, Set(String)) =
  DICT!Word.allInstances()->iterate(w, acc = Map {} |
    acc->including(w.name,
       w.synonyms->collect(w2 | w2.name)->asSet()) );

helper def : isSynonym(word : String, syn : String) : Boolean =
   let syns : Set(String) = thisModule.syns->get(word)
    in if syns <> OclUndefined then syns->includes(syn)
       else false endif;
```

* Jesús Sánchez Cuadrado, et al. "Optimization patterns for OCL-based model transformations.“ Satellite Events of MoDELS’08

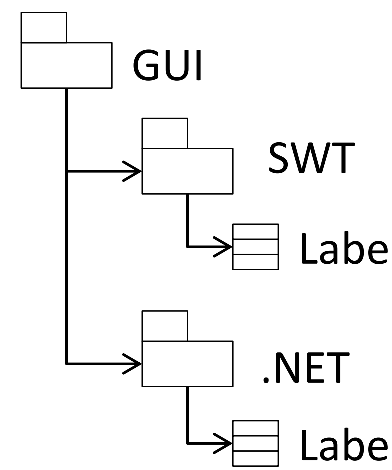
Iterate + Map

- Example
  - Compute a collection of class features we may want to organise features by its type

```
helper def: featuresByType : Map(CD!Classifier, Set(CD!Feature)) =
  CD!Features.allInstances()->iterate(f, acc = Map {} |
    if acc->get(f.type).oclIsUndefined() then
       acc->including(f.type, Set { f })
    else
       acc->including(f.type, acc->get(f.type)->including(f))
    endif
  )
```

Model References

- By default ATL assumes a flattened meta-model (e.g., no sub-packages)
  - This is not a problem if all classes in the meta-model have distinct names
  - Sometimes it is a problem
    - A warning will be reported at runtime
- How to disambiguate?
  - GUI!“GUI::SWT::Label"



This is the logical meta-model name

This is the actual meta-model path

Multiple models with the same meta-model

- Problems if the transformation has two input/output models with the same input/output meta-model
- E.g., two UML models as input model
- Keyword "in"
  - `from c : UML!Class in IN1`
  - `to c : UML!Class in OUT1`
- Also, allInstancesFrom('IN1')

Helpers

- Helpers can be attached to primitive types

```
helper context String def: firstToUpper() : String =
    self.substring(1, 1).toUpper() +
    self.substring(2, self.size());
```

- Helpers can be attached to OclAny
  - Not collections (i.e., `context Sequence(OclAny)`)

```
helper context OclAny def: myDebug() : String =
    self.debug(‘Debugging... ‘);
```

Libraries

- ATL support libraries of helpers
  - Only context helpers are allowed
  - Only operation helpers are allowed

```
library UMLfacilities;

helper context UML!Class def: allSuperClasses() : Set(UML!Class) = …
```

- Merged at runtime with the transformation

```
-- @nsURI UML=http://www.eclipse.org/uml2/5.0.0/UML
-- @path GUI=/guigen.trafo.uml2gui/metamodels/gui.ecore
-- @lib UMLfacilities=/guigen.trafo.uml2gui/lib/UMLfac.atl
module “uml2gui”;
create OUT : GUI from IN : UML;
uses UMLfacilities
```

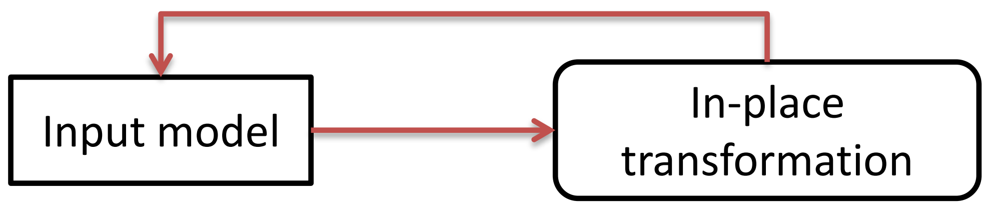
-- @lib
AnATLyzer annotation

The ATL language

Refining mode

Refining mode

- ATL support for in-place transformations *
  - The input model is changed by the transformation rules



* Refining Models with Rule-based Model Transformations.
Massimo Tisi , Salvador Martínez , Frédéric Jouault , Jordi Cabot.
Technical report.
* http://modeling-languages.com/refiningrefactoring-transformations-atl/

Refining mode

- Limitations
  - Simplistic support (poor’s man in-place transf.)
    - NOT recursive application of graph-rewriting rules
  - Adequate for “refinements” of a model
    - E.g., a refactoring
  - Not adequate for...
    - Things like the pac-man...
    - Complex program transformation manipulations
      - E.g., a compiler optimization
- Older versions implemented a copy strategy

Refining mode

- Design rationale
  - Fit in the model-to-model schema ATL developers are used to
  - Refining mode conceptually acts as a kind of copy transformation
    - Everything is implicitly copied
    - Write rules for what you want to change
      - If a binding is not overridden, the default value is used

Module

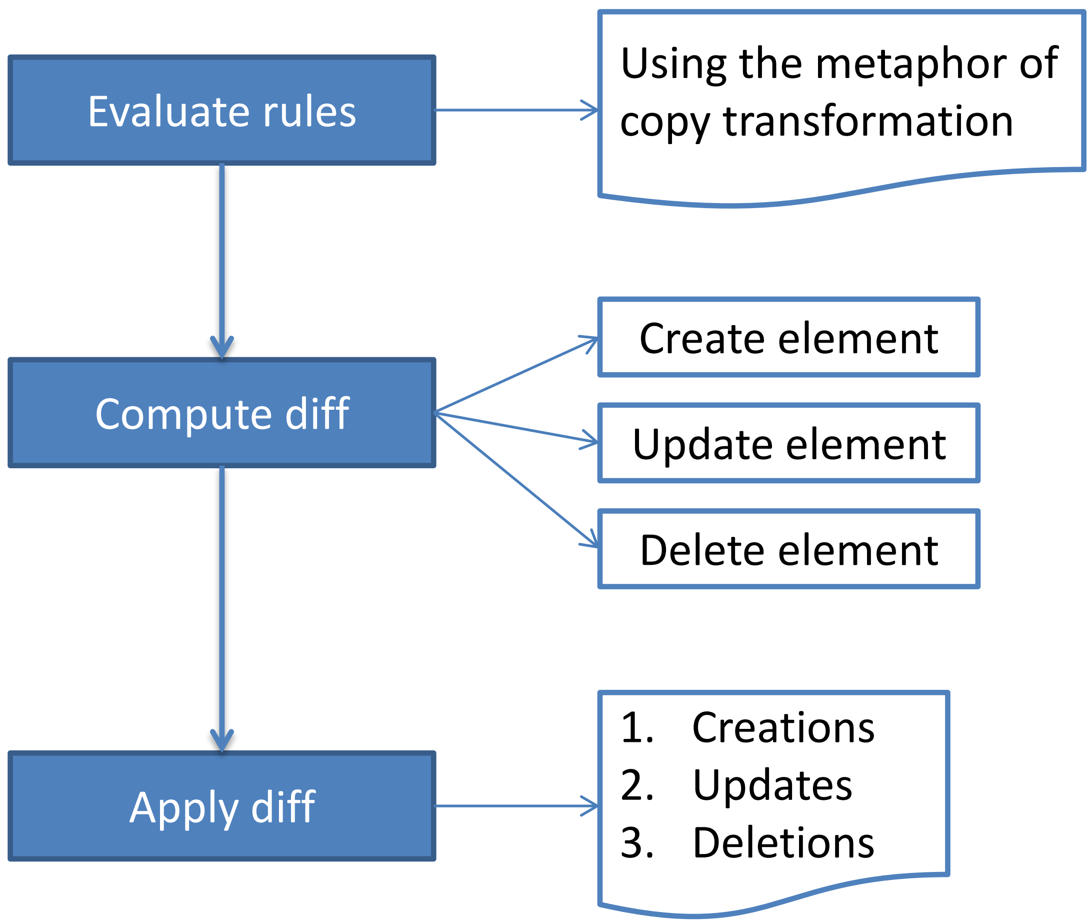
- Module declaration is a bit different
  – Requires atl2010 or emftvm versions for new in-place semantics
  – Keyword **refining** instead of **from**

```
-- @atlcompiler emftvm
-- @path CD=/guigen.trafo.uml2gui/metamodels/cd.ecore

module "copy inheritance";
create OUT : CD refining IN : CD;
```

Execution

- Two steps:
  - Rule modifications are stored in a change model.
  - Then, applies all the modifications at once
- This implies that rules always “see” the original model
  - Avoid rule recursion problems

Execution


Evaluate rules
Using the metaphor of copy transformation
Compute diff
Create element
Update element
Delete element
Apply diff
1. Creations
2. Updates
3. Deletions

cd2cd_01.atl

Example

- The CD2GUI transformation became complex due to the handling of inherited attributes
  - An alternative is to flatten inheritance by copying all inherited features

Example

```
-- @atlcompiler emftvm
-- @path CD=/guigen.trafo.uml2gui/metamodels/cd.ecore

module “cd2cd_01";
create OUT: CD refining IN: CD;

helper context CD!Class def: allFeatures : Sequence(CD!Feature) =
  self.superclasses->collect(c | c.allFeatures)->
                   flatten()->union(self.features);

rule class2class {
 from c1 : CD!Class
   to  c2 : CD!Class (
     features <- c1.allFeatures->collect(f |
       if f.oclIsKindOf(CD!Attribute) then thisModule.createAttribute(f)
       else if f.oclIsKindOf(CD!Reference) then thisModule.createReference(f)
       else OclUndefined.fail_('pattern match error') endif endif)
   )
}
```

Example

```
lazy rule createAttribute {
 from f : CD!Attribute
   to a : CD!Attribute (
     name <- f.name,
     lowerBound <- f.lowerBound,
     upperBound <- f.upperBound,
     isId <- f.isId,
     type <- type
  )
}

lazy rule createReference {
 from f : CD!Reference
   to a : CD!Reference (
     name <- f.name,
     lowerBound <- f.lowerBound,
     upperBound <- f.upperBound,
     isId <- f.isId,
     type <- type
  )
}
```

Example

- Remember that there is an implicit copy semantics

```
rule featureRemoval {
   from f : CD!Feature
--    to drop
}
```

  - In EMFTVM **drop** keyword is not used, but the target pattern is left empty

The ATL language

Modularity – Superimposition

Modularity

- ATL does not support any kind of module importation facility
- Superimposition *
  - Allows merging two ATL modules at load time

* Wagelaar, Dennis, Ragnhild Van Der Straeten, and Dirk Deridder. "Module superimposition: a composition technique for rule-based model transformation languages." Software & Systems Modeling 9.3 (2010): 285-309.

Modularity

- Example
  - We don’t want to decouple the mapping for widgets from the decision about which layout to use.
  - With superimposition:
    - Base module with mappings for widgets
    - “Extension” modules with rules to create the layout

cd2gui_base.atl

cd2gui_mobile.atl

Superimposition

- Basic idea:
  - Replace each rule in the original transformation with the rule with the same name in the superimposed module

```
rule class2frame {
    ...
}

rule attribute2text {
    ...
}

rule attribute2int {
    ...
}
```

superimpose

```
rule class2frame {
    ...
}

lazy rule createVLayout {
    ...
}
```

Superimposition

```
rule class2frame {
  <create a gridlayout>
}

rule class2frame {
    <create a vlayout>
}

lazy rule createVLayout {
    ...
}

rule attribute2text {
    ...
}

rule attribute2int {
    ...
}
```

Limitations

- Interface between transformations is just the rule names
- It is a “copy-paste” mechanism
  - To override a rule the original rule must copied and adapted
- No compile-time checks

- Some of these issues addressed in EMFTVM

Limitations

- Superimposition and rule inheritance
  - In standard ATL rule inheritance is handled at compile time.
  - But superimposition is handled at load time

```
rule A {
    ...
}

rule B extends A {
    ...
}
```

superimpose

Not valid

```
rule B extends A {
    ...
}
```

The ATL language

UML Support

uml2cd_with_profile.atl

UML Profiles

- Natively supported by the language via eOperations

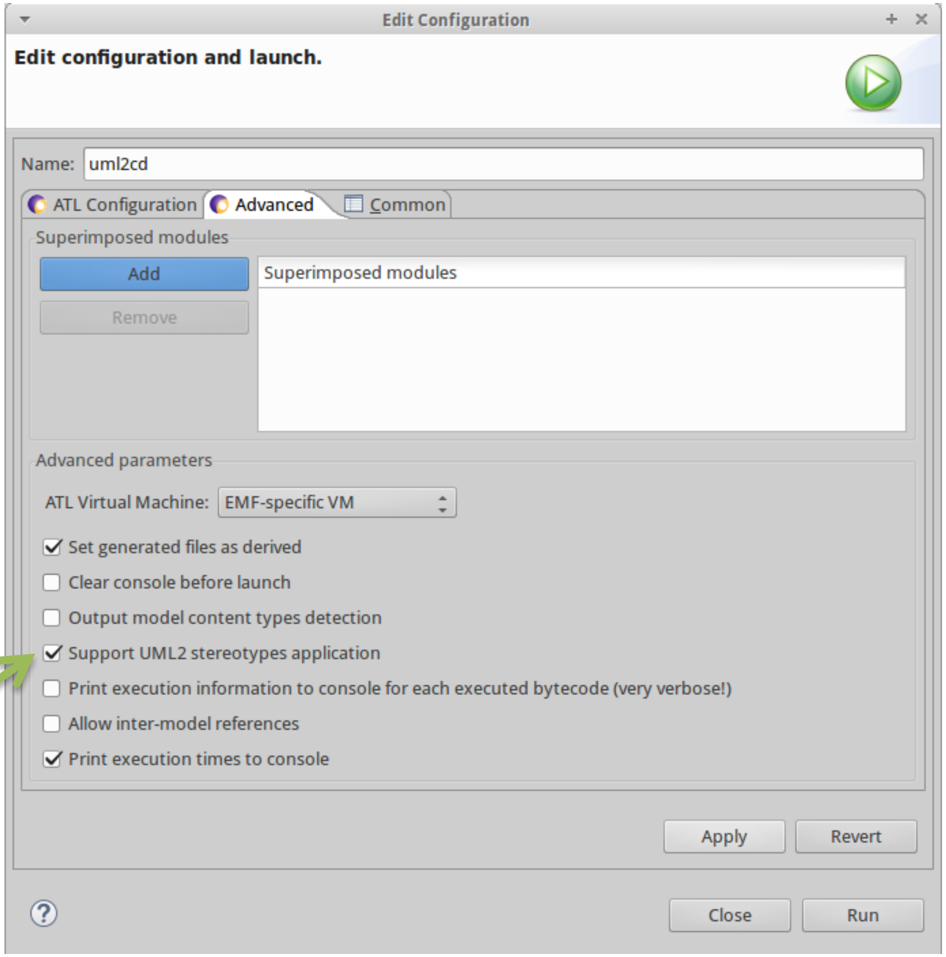
```
rule property2attribute {
   from p : UML!Property ( p.type.oclIsKindOf(UML!DataType) )
     to f : CD!Attribute (
        name <- p.name,
        type <- p.type,
        isId <- p.getAppliedStereotype('ExtendedCD::isID')
                    <> OclUndefined
     )
}
```

Profile operations

- Stereotype queries
  - getApplicableStereotype(String) : Stereotype
  - getApplicableStereotypes() : Stereotype
  - getAppliedStereotype(String) : Stereotype
  - getAppliedStereotypes() : Stereotype
  - getAppliedSubstereotype(Stereotype, String) : Stereotype
  - getAppliedSubstereotypes(Stereotype) : Stereotype
  - getValue(name, taggedValue) : OclAny
- Profile application
  - applyStereotype(Stereotype)
    - You must gather the stereotype first

```
helper def: getStereotype(name: String): UML!Stereotype =
   UML!Stereotype.allInstancesFrom('INPROFILE')->
      any(p | p.name = name);
```

UML profiles



Need if the UML model is in the target

Loading UML models

- If your UML model use primitive types from the UML library, the model needs to be loaded explicitly

```
module "uml2cd";
create OUT: CD from IN: UML, PT: UML;

rule DataType2DataType {
   from d1 : UML!DataType
     to d2 : CD!DataType (
        name <- d1.name
     )
}
```

The ATL language

Some patterns

Disclaimer

- Based on my own experience
  - By no means a complete list
  - Possibly wrong...

Matched rules

- **Problem**: When to use matched or lazy rules?
  - It is not always clear
- **Solution**: Try to favour matched rules.
  - Start your design with matched rules
  - Rationale: ATL has better support for lazy rules
  - Use lazy rules when:
    - The same element needs to be transformed more than once (i.e., be aware of child stealing)
    - It is easier to control the generation of objects from the calling rule

Fail as soon as possible

- **Problem**: Pattern match using nested ifs. No sensible default value can be used.
  - There are no runtime exceptions in ATL
  - Returning “OclUndefined” is not an option
- **Solution**: Fail as soon as possible.
  - Use OclUndefined.fail_(“Reason”)
    - Imitation of MatchError in other languages
  - Convention supported by anATLyzer

Rule conflicts

- **Problem**:
  - There are conflicting rules in the transformation
- **Solution**
  - Use rule inheritance if possible
  - Add filter to rules

Imperative code anti-pattern

- **Problem:** Attribute initialization in do in matched rule.
  - There is no guarantee that the rule is executed before the others.
- Example:

```
helper def : dtString : OO!DataType = OclUndefined;

rule model2model {
  from m1 : UML!Model
  to m2 : CD!Model(
        ...
  ), str : OO!DataType ( name <- ‘String’ )
  do {
      thisModule.dtString <-  str
  }
}
```

Imperative code anti-pattern

- **Solution:**
  - Use an entry point rule to initialize only once.

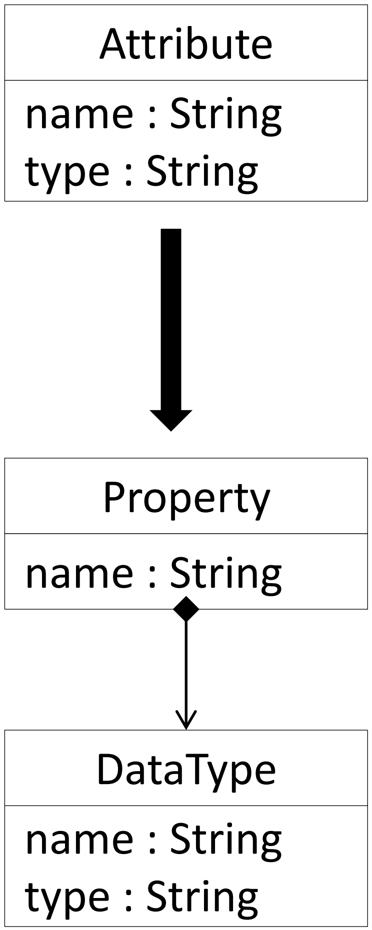
Dynamic map

- **Problem:**
  - Example. Need a correspondence between primitive types and objects
- **Solution:**
  - Use string (or any other datatype) as input pattern of a lazy rule
  - Act as dynamically filled map

Dynamic map

- Map data types encoded as strings to explicit data types

```
rule Attribute2Attribute {
  from a : CD!Attribute
    to p : UML!Property {
      name <- a.name,
      type <- thisModule.createDataType(a.type)
  }
}

unique rule createDataType {
  from s : String
    to t : UML!DataType {
    name <- s
  }
}
```



Proper typing

- **Problem:** Should I care about types in ATL?
  - Sometimes it is difficult to find the proper types
- **Solution:**
  - Use proper typing as far as possible.
  - AnATLyzer provides quick fixes to help you in most of the cases

Proper typing

- **Solution:**
  - Sometimes to need a fall back
    - Use OclAny

```
helper def myHelper(p : Boolean): OclAny =
   if p then 1
   else '1' endif;
```

Which is the type?