ETL: Epsilon Transformation Language

MISO - Uniandes
References

- http://www.eclipse.org/epsilon/examples/
- Examples available at the wiki
“Models are first class entities”

“Transformations are assets”
What is ETL?

- ETL is a model transformation language developed at University of York to answer the QVT Request For Proposal.
- It is a hybrid of declarative and imperative.
What is ETL?

- In declarative style:
  - simple mappings can be expressed simply.

- Imperative constructs:
  - are provided to express more complex mappings.
What is ETL?

- An ETL transformation program is composed of rules that define:
  - how source model elements are matched and navigated
  - how to create and initialize the elements of the target models.
Motivating example: Class 2 Relational

Class metamodel
Motivating example: Class 2 Relational

Relational metamodel
Mappings between Class and Relational metamodel
ETL and Ecore
ETL Engine

ETL Transformation rules

Input MM
Input M
Output MM

Output M'

XMI
Structure of an ETL transformation

- An ETL module corresponds to a model to model transformation.
Structure of an ETL transformation (cont.)
ETL modules

They are the files that contain the transformations

.etl extension

Consist of:

- An optional import section:
  - enables to import some existing EOL libraries or ETL modules
- A set of operations:
  - can be viewed as an ETL equivalent to Java methods
- Pre/post blocks
- Rules
Import

- This section is optional
- Defines the EOL libraries or ETL modules to be imported
- Example:

  ```import "strings.eol";```
EOL Operations

- They are **functions** used to implement code that can be reused
- Possible to have a no context-defining operation

```ruby
operation getIntegerDataType() : Class!DataType {
    Class!DataType.allInstances() ->
    select(e | e.name = 'Integer') -> first()
}
```

- Calling the function
  ```ruby
  objectIdType();
  ```
EOL Operations

- Context-defining operation:

```java
operation Class!Class getMultivaluedAttributes() : OrderedSet {
...
;
```

- Calling the function:

```
c.getMultivaluedAttributes()
```
Pre block

- `pre {`
- "Running ETL".println();
- `var integerDatatype : Class!Datatype = Class!DataType.allInstances()->`
- `select(e | e.name = 'Integer')->first();`
- `}`
Post block

```java
post {
var mgs : String := "Number of tables " + Relational!Table.all().size();
mgs.println();
}
```
Transformation Rules

- Two different kinds of rules:
  - matched rules (declarative programming)
  - Lazy rules (imperative programming)
Matched Rules

**rule** DataType2Type

**transform**

dt : Class!DataType
to

out : Relational!Type

{
    name = dt.name
}
Matched Rules

rule DataType2Type

transform

dt : Class!DataType

to

out : Relational!Type {

    name = dt.name

}
Matched Rules

- We need:
  - To access any element of the input model
  - To create elements of the output model from information of the elements of the input model

- The access to the elements is done using
  The OCL: Object Constraint language
_matched_rules: input

- The input pattern consists of:
  - the keyword `transform`,
  - the declaration of input variables

```
transform
dt : Class!DataType
```
Matched Rules: Output

- The output pattern declares:
  - on which output elements of the output model, the input elements matching the input pattern will be transformed
  - an OCL expression that returns the input element to be transformed

- The implementation of the output pattern declares the details of the transformation

- It is possible to have more than one element in the output pattern
A filter is an OCL expression that restraints the elements of the input model to those that satisfy a set of constraints

Example:
- Refer to the instances of the `Attribute` element in the Class Metamodel, where its `multivalued` is equal to `true`:

```oclmml
transform e : Class!Attribute
to ...
guard : (e.multivalued)
```
Declares a variable and a sequence of assignment statements.

```plaintext
to
    out : Relational!Type { 
        name = dt.name 
    }
```

Building and instance of the element `Type` of the output model
The `name` attribute of the new element will have as value the value of the `name` attribute of the element referenced by the variable `dt`
ETL Module execution semantics

- Three phases:
  - a module initialization phase:
  - a matching phase of the source model elements
  - a target model elements initialization phase
ETL execution semantics

- Four phases:
  - a module initialization phase:
    - the *pre* blocks of the module are executed first in the order
  - a matching phase of the source model elements:
    - matching condition are tested with the model elements of the source.
    - if it is matched, ETL engine allocates the set of target model elements
  - a target model elements initialization phase:
    - each allocated target model element is initialized by executing the code of the bindings that are associated
  - a finalization phase:
    - the post blocks of the module are executed in the order in which they have been declared
Source element resolution

- To automate this task and reduce coupling between rules, ETL contributes the `equivalent()` operation.
- It resolves source elements to their transformed counterparts in the target models.
- The resolution is possible only if there exist a containment relationship between the target/source elements.
Source element resolution (cont.)
Source element resolution (cont.)

rule Class2Table
transform	s:Class!Class
to
t:Relational!Table {
guard : (not s.isAbstract)
name = s.name
col = s.att.equivalent()
}

rule Attribute2Column
transform	s:Class!Attribute
to
t:Relational!Column{
name = s.name
}
Imperative solution

pre {
    Class!Class.all()->select(e|not e.isAbstract)->collect(e | e.o1);
}

Operation Class!Class o1 : Relational!Table {
    var t : new Relational!Table;
    t.name = self.name;
    t.col = self.att.collect(e | e.e2);
    return t;
}

Operation Class!Attribute o2 : Relational!Column {
    var t = new Relational!Column;
    t.name = self.name;
    return t;
}
Hybrid solution

rule Class2Table
transform
    s:Class!Class
to
t:Relational!Table {
    guard : (not s.isAbstract)
    name = s.name
    col = s.att.collect(e | e.o2);
}

Operation Class!Attribute o2 : Relational!Column {
    var t = new Relational!Column;
    t.name = self.name;
    return t;
}
Transformation Engine

- Implemented in Java
- Can use EMF as model repository
- Input/output models and metamodels are managed by the underlying model repository
ETL Eclipse Plug-in

There are:

- an ETL editor
- Launch configuration facilities
- Debugging option
- May be executed from MTCFlow