Lombardi Tiziano

l.tizzy@libero.it

Metamodel Adaptor

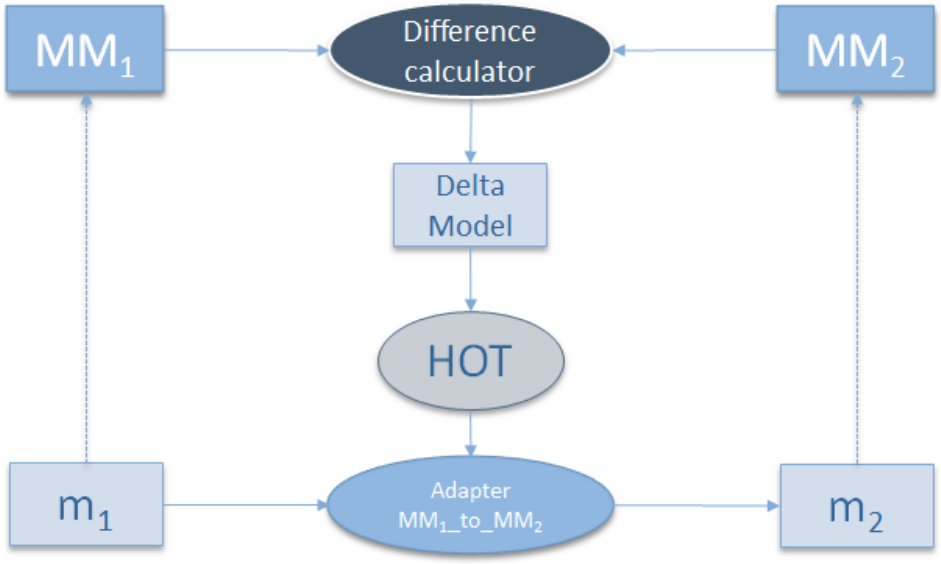
Automatic creation of an Adapter as part of the chain of transformations process

Report index

Project scope and proposal

The project proposes the development of an **ADAPTER** helping in the automation of the process of transformation chaining, that is a service provided by the MDE Forge framework.

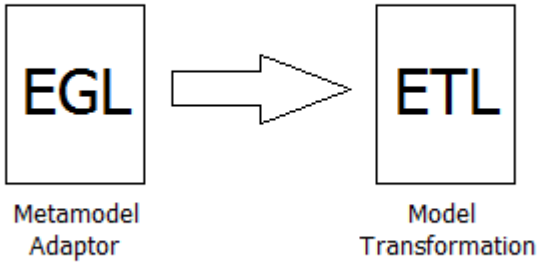
What you want to accomplish is synthesized by this scheme:



For our purposes the **Difference calculator** is *EMFCompare* and the **Delta Model** is its output. The HOT (**High Order Transformation**) is a Model-To-Text transformation that produce an *Adapter* (that is a real transformation) between the two metamodels we are considering (without user intervention).

So the result of this process is a transformation written in **ETL language**.

Project high level sketch



Based on the adapted Technologies, the first come out idea consists in doing an **EGL core** engine (our HOT) which analyze the Diff model generated by EMFCompare to produce an **ETL transformation** which has to migrate all the models conforming to the source metamodel and create new models conforming to the new target metamodel.

To analyze the *Diff model*, the EGL core has to **investigate** recursively all the **changes** written in it, going from the Packages down to the Classes' Elements. It's useful that this Diff model is tree-structured, in order to simplify the analysis.

This analysis produces guidelines to correctly write down ETL rules which migrate models. It's required **user interaction** during this process in order to solve unpredictable decisions. In fact, engine encounters three types of changes:

- **Non-Breaking Changes:**
  which do not break the conformance of models to the corresponding metamodel;

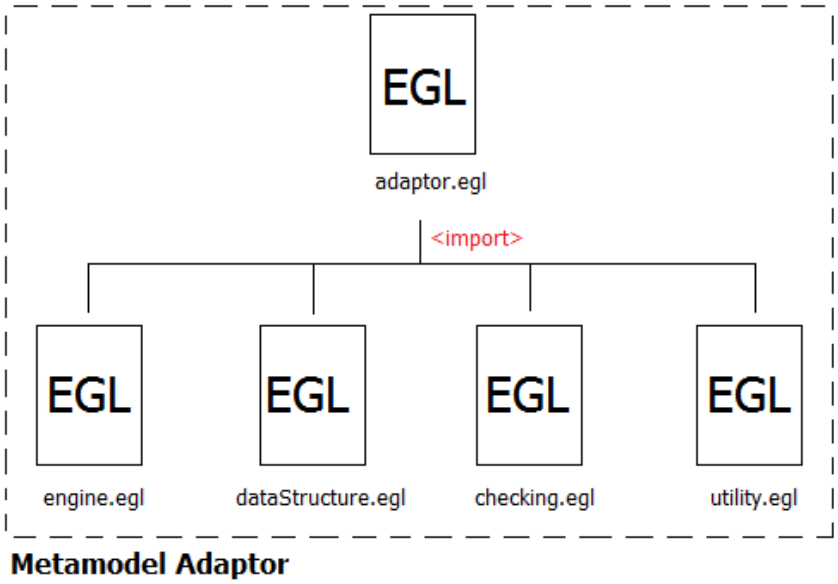
- **Breaking and Resolvable:**
  which do break the conformance of models, even though they can be automatically resolved;

- **Breaking and Not Resolvable:**
  which do break the conformance of models which cannot automatically resolved, and user intervention is required;

While for the first two cases it's possible to hard code rules to manage changes, in the last case user is necessary prompted what to do.

Moreover, the engine should have the capability to recognize **Evolution Patterns** (such as Extract Superclass, Split Attribute, etc.) in order to preserve the full and correct evolution of the metamodels and derived models.

Adaptor architecture



To implement the sketch discussed in the previous paragraph, it has been chosen to modularize the **Metamodel Adaptor** in different modules; this to simplify future expansion and bug fix.

Adaptor.egl

This is the main module of the system. It's the one is going to be launched by the configuration (*launch.xml*) and contains in the order:

- The static **PRE** and **POST** part of the ETL transformation
- The **main loop** that explore the *Diff model* and create the dynamic part of the ETL transformation
- The **static ETL operations** used frequently by the dynamic ETL code.

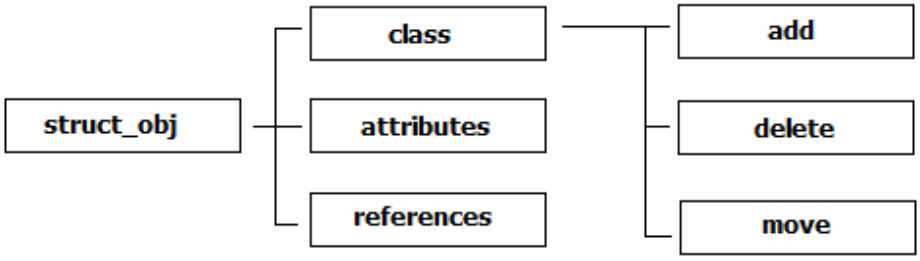
Engine.egl

This is the effective core of the Adaptor. It contains:

- The **explore()** operation, which is the one that really *examine* the Diff model
- The **operateXDiff()** operations, which *analyze the changes* in the X element
- The **writeDownX()** operations, which *creates the ETL code* to manage the specific type of change

DataStructure.egl

All the changes found by the engine are stored in a data structure managed by this module. The data structure is described as it follows.



The **Data Structure** is made of 4 levels:

- **Level 0** – this is the top level and it's the main *Map* objects which contains 3 keys: 'class', 'attributes', 'references'.
- **Level 1** – every key of the previous level is a *Map* object which contains 3 keys: : 'add', 'delete', 'move'.
- **Level 2** – every key of the previous level is a *Sequence* of objects, which contains the list of the metamodel elements found by the engine.
- **Level 3** – every entry of the Sequence is *a real element* of the metamodel, which could be a Class, an Attribute or a Reference.

Obviously, in an Object-oriented point of view, this module contains all the **operations** that allow to access the data structure, in order to **read/write elements** or **check** some kind of **status**.

Checking.egl

This module contains a set of operations which are intended to check some properties of the involved object.

They are structured mainly in two ways:

- **Object.isProperty : Boolean**
  In this case can be easy overridden to have different effects on different kind of objects.
- **isProperty(o: Object) : Boolean**
  In this case is usually defined only one checker, which is less readable if has to been applied to different kind of objects.

In any case, both the types of checking return only a *Boolean* value.

Utility.egl

This module contains a set of generic operations used by other modules, which can solve particular kind of problems in a no-matter way.

There's **no specification on** how to write **these operations** neither what kind of operations we expect to find in it.

However, to keep the code structured and clean, it's recommended *not to put here operations that can belong to the other modules.*

Code explanation

After discussing the Architecture of the Metamodel Adaptor in the previous section, it's now time to explore the code of the real implementation of the system, following the same schema of the Architecture.

Adaptor.egl

Import section of the module

```
[%
	// Imports
	import "engine.egl";
	import "dataStructure.egl";

	// Pre and Post ETL statements
%]
```

Fixed PRE and POST statements of the ETL code

```
pre {
"$> Start migration...\n".println();
}

post {
"$> Migration completed.\n".println();
}
```

Main loop of the Adaptor

```
[%
// MAIN PROCEDURE
// Creating dynamic Adaptor's rules ( starting from matches in the Comparator )

for (comp in Comparison.allInstances()) { // if there are more than 1 Comparator (???)

    "--- LOG: List of changes ---".println();
    comp.matchedResources.leftURI.println('Left: ');
    comp.matchedResources.rightURI.println('Right: ');
    "Right -> Left".println('Evolution: ');

    if ( System.user.confirm("Are you ready to proceed?") ) {

        var elementsChanging = DSCreate(); // changes data structure

        // exploring changes in deep mode
        for (el in comp.matches)
            explore(el, comp, elementsChanging);
    }
    else "Interrupted!".println();
}

// fixed ETL Code
%]
```

For every *Comparator* object (default there is one in each Diff model) it's reported source and destination metamodel, then is asked the user to proceed or not with the creation of the ETL code.
In positive case, the data structure is created and the first level matching elements are investigated (for metamodels they are the packages for the classes).

Static ETL operations

```
// list of all Input Classes
operation cListCl(): Sequence {
    … … …
}

// search for the first match
operation Any cFilterType(ft): Boolean {
    … … …
}

// filter class list by Type Name
operation cFilterListCl(tot: Sequence, filterTypes:
Sequence): Sequence {
    … … …
}
```

NB. The code of the operations it's not shown because it's not so relevant to be discussed.

Engine.egl

Import section of the module

```
[%
    // Imports
    import "utility.egl";
    import "dataStructure.egl";
    import "checking.egl";
```

The explore procedure

```
// EXPLORING CHANGES

operation explore(el: Any, comp: Any, elChanging: Map) {

    // setting source and destination models
    var source = el.right;
    var dest = el.left;

    // inspecting MATCHES

    // PACKAGES
        if ( isPackageMatch(el) ) {
            … … …
        }

    // CLASSES
        else if ( isClassMatch(el) ) {
            … … …
        }

    // ATTRIBUTES
        else if ( isAttributeMatch(el) ) {
            … … …
        }

    // REFERENCES
        else if ( isReferenceMatch(el) ) {
            … … …
        }

    // ERROR!
        else {
        "I can't recognise what is changed!".println("!!!!!!");
        }
}
```

The explore procedure is intended to be a *recursive algorithm* to analyze the tree-structured Diff model. In the main module, it's applied to the packages of the metamodels. Recursively, it goes through classes and class elements.
In its body, the operation check what type of element is under observation, do specific actions for it and then goes deep into its "submatches".

Packages actions

```
    // PACKAGES
    if ( isPackageMatch(el) ) {

        // Identifying Packages
        "Exploring package changes:".println("\n\t");
        source.println("Source: ");
        dest.println("Destination: ");

        // Explore the package changes
        operatePackDiff(el, elChanging);

        // Explore changes in the package elements
        if (not el.submatches.isEmpty())
            for (subEl in el.submatches)
                explore(subEl, comp, elChanging);
    }
```

First of all, it notifies which packages it refers to, from Source and Destination metamodels. Then it operates specific actions on the packages.
Finally, it goes deep into packages submatches, which are Class elements.

Class actions

```
    // CLASSES
    else if ( isClassMatch(el) ) {

        // Identifying Class
        ("Class [ " + classIdentification(source, dest) +
            " ]").println("\n\t");

        // Ignore DELETED classes
        if ( not elChanging.DSisDeletedClass(source) ) {

            // Explore the class changes
            operateClassDiff(source, el, elChanging);

            // write class rule (head)
            writeDownClassHeadRule( source, dest );

            // Explore changes in the class elements
            if (not el.submatches.isEmpty())
                for (subEl in el.submatches)
                    explore(subEl, comp, elChanging);

            // write class rule (foot)
            writeDownClassFootRule( source, dest );
        }
        else "Ignored DELETED class!".println();

        "\t--\t--\t--".println();
    }
```

First of all, it displays information about the class it's analyzing. Then, ignoring deleted classes (this is a policy hard coded, but changeable) it creates the rule for the specific class, or better the header and the footer; finally it goes deep into submatches to create eventually the instructions to migrate the class attributes and references.

Attribute actions

```
// ATTRIBUTES
    else if ( isAttributeMatch(el) ) {

        // Identifying Attribute

        // explore the atribute changes
        operateAttributeDiff(source, dest, el, elChanging);

}
```

```
// ... attributes
operation operateAttributeDiff(source, dest, el: Any, elChanging: Map) {

    // operate Attribute adaptation
    if ( elChanging.DSisDeletedAttribute(source) ) {
        (source.name + ' > ' + source.getEType().name + ' (so ignored)').println("DELETED ATTRIBUTE: ");
    }
    else if ( elChanging.DSisMovedAttribute(dest) ) {
        if ( dest.isRequired() or dest.isMany() or
            System.user.confirm('Requesting the optional ' + dest.type().name + ' [ ' + dest.name + ' ] @ '
+ dest.eContainer().name + '?') ) writeDownAttributeMove( source, dest );
        else "(skipped initial value not required): ".print();

        (dest.name + ' > ' + dest.getEType().name).println("MOVED ATTRIBUTE: ");
    }
    else if ( elChanging.DSisAddedAttribute(dest) ) {
        if ( dest.isRequired() or dest.isMany() or
            System.user.confirm('Requesting the optional ' + dest.type().name + ' [ ' + dest.name + ' ] @ '
+ dest.eContainer().name + '?') ) writeDownAttributeAdd( source, dest );
        else "(skipped initial value not required): ".print();

        (dest.name + ' > ' + dest.getEType().name).println("ADDED ATTRIBUTE: ");
    }
    else {
        if ( dest.isRequired() or dest.isMany() or
            System.user.confirm('Requesting the optional ' + dest.type().name + ' [ ' + dest.name + ' ] @ '
+ dest.eContainer().name + '?') ) writeDownAttributeCopy( source, dest );
        else "(skipped initial value not required): ".print();

        (dest.name + ' > ' + dest.getEType().name).println("COPIED ATTRIBUTE: ");
    }
```

Simply creates the correct code to migrate attributes from the source to the destination metamodel. As shown in the **operateAttributeDiff()** operation, the ETL code is created differently and specific for the 4 types of changing. It's also notified in console everything done. Even for the attributes, the policy (changeable) consist in ignoring deleted attributes.

References are threated in the same way of attributes, so it's not relevant to show the code.

DataStructure.egl

Creation of the empty data structure.

```
// DATA STRUCTURE
/*
    Data structure to sign elements changing

    elementsChanging [Map] -> [{'class', 'attribute', 'reference'} -> list]

    list [Map] -> [{'add', 'delete', 'move'} -> Sequence]

*/
// DS constructor
// NB. The creation for datastructure is Lazy, so only what is used is created!
operation DSCreate(): Map {
    var ris = new Map;

    // Return new empty unstructured data structure
    return ris;
}
```

To optimize the usage of the memory, the structure is lazy created, to be allocated only if it is necessary.

In the top comment we can see the structure as it was described in the previous chapter. Then there's the constructor, which initialize the empty first level.

Getting a first level element.

```
// GETTING

// Get Level 1
operation Map DSgetClasses(): Map {
if ( self.get('class') == null ) self.put('class', new Map);
return self.get('class');
}
```

This is an example of a getter for the first level of the structure; as we can see, the getter creates an empty sublevel if it does not exist.

Getting second level element.

```
// Get Level 2
operation Map DSgetDeleted(): Sequence {
    if ( self.get('delete') == null ) self.put('delete', new Sequence);
    return self.get('delete');
}
```

This is an example of getter for the second level of the structure; as we can see, even in this case the getter creates an empty sublevel if it does not exist.

Adding elements to the structure.

```
// ADDING

// Add Level 2 - Deleted
operation Map DSaddDeletedClass(cl: Any) {
    self.DSgetClasses().DSgetDeleted().add(cl);
}

// Add Level 2 - Moved
operation Map DSaddMovedAttribute(attr: Any) {
    self.DSgetAttributes().DSgetMoved().add(attr);
}

// Add Level 2 - Added
operation Map DSaddAddedAttribute(attr: Any) {
    self.DSgetAttributes().DSgetAdded().add(attr);
            }
```

This code provides three examples of adding elements to the structure, respecting the type of change about the element (Deleted, Moved, and Added).

Checking the presence of an element.

```
// CHECKING

// Check Level 2 - Deletion
operation Map DSisDeletedClass(cl: Any): Boolean {
    return ( self.DSgetClasses().DSgetDeleted().selectOne(x: Any | x == cl) <>
null );
}

// Check Level 2 - Moving
operation Map DSisMovedAttribute(attr: Any): Boolean {
    return ( self.DSgetAttributes().DSgetMoved().selectOne(x: Any | x == attr) <>
null );
}

// Check Level 2 - Adding
operation Map DSisAddedAttribute(attr: Any): Boolean {
    return ( self.DSgetAttributes().DSgetAdded().selectOne(x: Any | x == attr) <>
null );
            }
```

This code provides three examples of checking the presence of a specific element in a specific type of change. It's the main use of the structure to check easy the way an element has changed.

Checking.egl

```
// DIFFERENCE KIND
operation Any isAddedElement(): Boolean {
    return ( self.kind.asString() == 'ADD' );
}

// (META)MODEL ELEMENT TYPE
operation Any isClassElement(): Boolean {
    return ( self.value.type().name.asString() == 'EClass');
}

// DATATYPE IDENTIFY
operation isIntegerAttribute(attr: Any): Boolean {
    return attr.getEType().name == 'EInt';
}

// TYPE OF MATCHING ELEMENT
operation isPackageMatch(el: Any): Boolean {
    if ( el.left <> null ) return ( el.left.type().name == 'EPackage' );
    else if ( el.right <> null ) return ( el.right.type().name == 'EPackage' );
            }
```

This code report the types of checking are made by this library.

- In the first case is related to the EMFCompare metamodel, and investigate the kind property of the differences.
- In the second case is related to the ECore metamodel, and investigate the ECore type of an element.
- In the third case is related to the ECore basic types, and investigate the ECore type of an attribute
- In the fourth case if related to the ECore basic types used in the EMFCompare models, and investigate which types of matching has been detected.

Utility.egl

```
// UTILITY

// given Source ad Dest, return a string identifying the class and the owner
package
operation classIdentification(source, dest): String

// given Source and Dest, return the name of a class not deleted
operation getUndeletedClassName(source, dest): String

// given a Class returns the list of all parent classes (if exists)
operation listExtensionClassParent(orig): String

// given a class create the list of the supertypes
operation createSubTypesList(cl: Any)

operation Any searchForSuperType(cl, st: Any)
```

This library contains operation to help in some secondary issues, which is not so significant to have the code explained.
Signs of the operations are reported just to know which they are, parameters required and types reported.

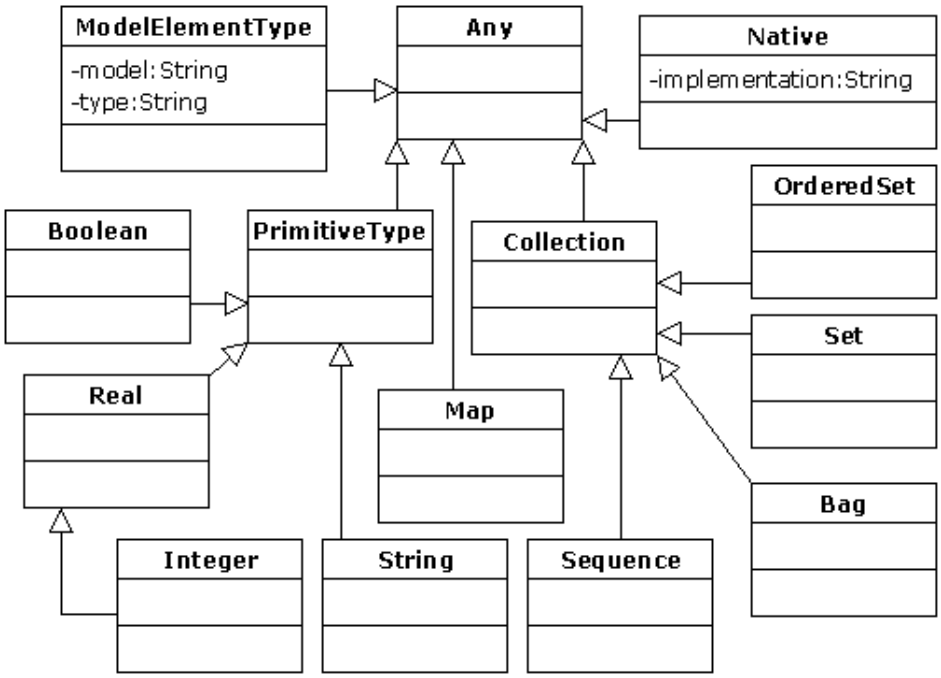
- The first identify the class, not knowing a-priori if both source and dest exist.
- The second returns the name of the class in the case of deletion.
- The third returns the list of parents of the class given.
- The fourth (helped by the fifth) returns the list of supertypes about the one given.

Appendix A – Epsilon framework languages

EOL (Epsilon Object Language)

The primary aim of EOL is to provide a reusable set of common model management facilities, atop which task-specific languages can be implemented. However, EOL can also be used as a general-purpose standalone model management language for automating tasks that do not fall into the patterns targeted by task-specific languages. This section presents the syntax and semantics of the language using a combination of abstract syntax diagrams, concrete syntax examples and informal discussion.

EOL programs are organized in modules. Each module defines a body and a number of operations. The body is a block of statements that are evaluated when the module is executed. Each operation defines the kind of objects on which it is applicable (context), a name, a set of parameters and optionally a return type. Modules can also import other modules using import statements and access their operations.



EOL enables users to create objects of the underlying programming environment by using native types. A native type specifies an implementation property that indicates the unique identifier for an underlying platform type. For instance, in a Java implementation of EOL the user can instantiate and use a Java class via its class identifier.

A common assumption in model management languages is that model management tasks are only executed in a batch-manner without human intervention. However, as demonstrated in the sequel, it is often useful for the user to provide feedback that can precisely drive the execution of a model management operation.

User-input facilities have been found to be particularly useful in all model management tasks. Such facilities are essential for performing operations on live models such as model validation and model refactoring but can also be useful in model comparison where marginal matching decisions can be delegated to the user and model transformation where the user can interactively specify the elements that will be transformed into corresponding elements in the target model.

ETL (Epsilon Transformation Language)

The aim of ETL is to contribute model-to-model transformation capabilities to Epsilon. More specifically, ETL can be used to transform an arbitrary number of input models into an arbitrary number of output models of different modeling languages and technologies at a high level of abstraction.

ETL has been designed as a hybrid language that implements a task-specific rule definition and execution scheme but also inherits the imperative features of EOL to handle complex transformations where this is deemed necessary.

ETL enables specification of transformations that can transform an arbitrary number of source models into an arbitrary number of target models. Another common assumption is that the contents of the target models are insignificant and thus a transformation can safely overwrite its contents. As discussed in the sequel, ETL - like all Epsilon languages - enables the user to specify, for each involved model, whether its contents need to be preserved or not.

```
(@abstract)?
(@lazy)?
(@primary)?
rule <name>
transform <sourceParameterName>:<sourceParameterType>
to <rightParameterName>:<rightParameterType>
(, <rightParameterName>:<rightParameterType>)*
(extends <ruleName>(, <ruleName>)*)? {

(guard (:expression)|({statementBlock}))?

statement+
}
```

```
(pre|post) <name> {
statement+
}
```

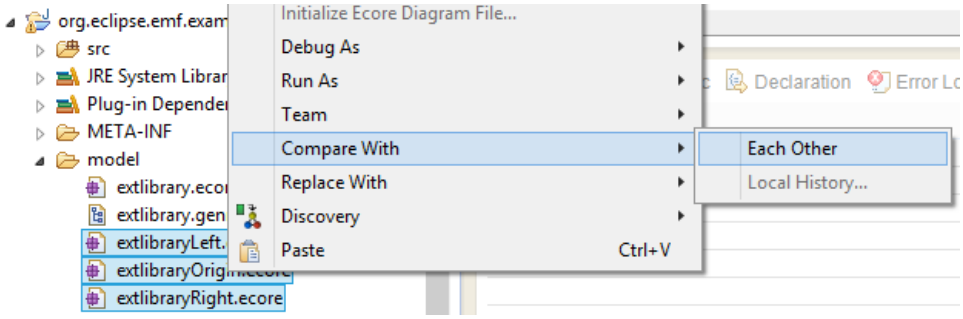
EGL (Epsilon Generation Language)

EGL provides a language tailored for model-to-text transformation (M2T). EGL can be used to transform models into various types of textual artifact, including executable code (e.g. Java), reports (e.g. in HTML), images (e.g. using DOT), formal specifications (e.g. Z notation), or even entire applications comprising code in multiple languages (e.g. HTML, JavaScript and CSS).

EGL is a template-based code generator (i.e. EGL programs resemble the text that they generate), and provides several features that simplify and support the generation of text from models, including: a sophisticated and language-independent merging engine (for preserving hand-written sections of generated text), an extensible template system (for generating text to a variety of sources, such as a file on disk, a database server, or even as a response issued by a web server), formatting algorithms (for producing generated text that is well-formatted and hence readable), and traceability mechanisms (for linking generated text with source models).

Appendix B – Known issues

EMFCompare (Eclipse plugin limitations)



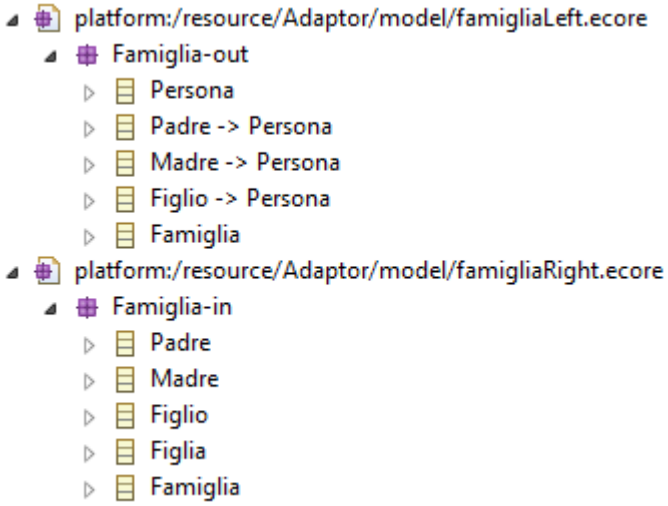
Due to the implementation of the EMFCompare Plugin, it's not possible to decide the verse of the comparison between metamodels, unless using names tricks. In fact, when selecting two metamodels EMFCompare launch the comparison assuming that the first is the new one and the second is the original.

It's possible to take control of comparison verse using specific names for the files; in the specific case, appending "Left" to the new metamodel and "Right" to the old it's possible to have a result that define all the changes Right -> Left.

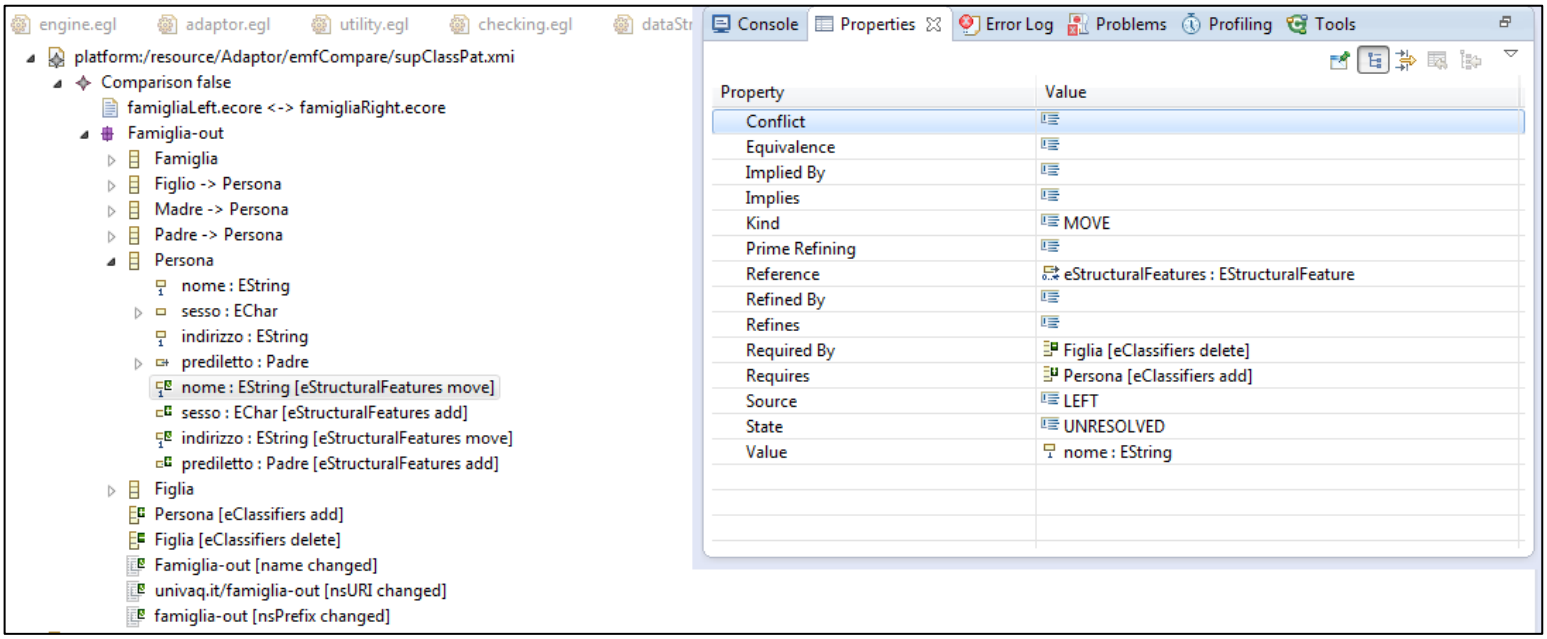
EMFCompare (Diff. model issues)

The main issue encountered regards the Evolution Pattern recognition made (or not) by EMFCompare. In fact, it seems that EMFCompare it's not so smart to recognize patterns as often as it has to.

For example, it's reported an unrecognized easy Extract SuperClass pattern.



This is the comparison between the input metamodel (below) and the output metamodel (above). As shown, it has been extracted the SuperClass *Persona* and deleted the class *Figlia*.



This image show how the pattern is not recognized. In the specific case, the tool thinks that the attribute *nome* in the class Persona was moved from Figlia, instead of a correct detection which involves that the attribute *nome* was derived from the user-selected subclass among its subclasses (Figlio, Madre, Padre).

Appendix C – Management of Extract Superclass pattern

```
operation writeDownClassHeadRule(source, dest) {
    var cName = getUndeletedClassName(source, dest);
    var cExt = listExtensionClassParent(dest);
    var cExtVirtual = listExtensionVirtualClassParent(dest);

    if (source == null) {
%]

    // Adapting non-existent class '[%=cName%]' @ package '[%=dest.eContainer().name%]'
    operation Adapt_[%=cName%](s, d: Any)
    {
[%
    }
    else {
%]

    // Adapting class '[%=cName%]' @ package '[%=dest.eContainer().name%]'
    rule Adapt [%=cName%]
    transform s: Input![%=cName%]
    to d: Output![%=cName%]
    [%=cExt%]
    {
[%
    }
%]
        var clChoiceList = cListCl();
        var sourceCh;
        var sourceList;
        var ft;

        [%=cExtVirtual%]
[%

}
```

The code shown above explain how the engine manages the **Extract Superclass Pattern**. In a few words, this pattern consist of some classes in the source model which have some common fields and/or methods; in the evolved model they are grouped in a new shared abstract superclass.

In this case we have a new class in the destination model which is not present in the source one; so, it's not possible to create a migration rule, because there's no source which refers to. Instead, it's created an operation which has in input and output the classes conformed to the respective metamodels passed by the concrete rules for the instances. In other words, a class A in the input model is migrated to the class A' in the output model. The class A' has some fields declared in the abstract superclass B. There's a concrete rule which migrates the non-common fields from A to A'; then, with an "abstract rule" implemented as operation, are migrated the common fields from A to B (which in the reality are contained in the A' object too).

To implement this idea it's just necessary to change the declaration of the rule as an operation for the abstract superclass (checked as a class which is present only in the destination model). Children of this introduced abstract superclass has at the end of the rule a call to this operation. Even if there's the possibility to have some introduces class, only those which are abstract and have children will be invoked.

This way maintain a rigorous modularity of the ETL code, without redundant code.