# Presentation of EMF Compare Utility

Antoine Toulmé

Intalio Inc.
1000 Bridge Pkwy, Suite 210
Redwood City, CA 94065, USA
email: atoulme@intalio.com

**Abstract:** In This paper we introduce a new approach to compare models based on a total comparison of their elements. We explain why we do not rely on the approaches based on identifying attributes. We present a weight system based on the information theory that computes the importance of the differences between two elements. Finally, we present an application of this approach using the Eclipse Modeling Framework.

**Keywords:** EMF, Comparison Tools, Information Theory, Model-Driven Development

## Introduction

A model is the representation of an object. It conceptualizes it by highlighting some of its aspects. Models are widely used in the programming world to build applications. Since models represent applications, they are often changed to reflect the modifications of the application, they are modified by different people, and their lifecycle is complex. That's why users need to identify the changes between models or to access to the model's history. Those operations may be performed by a comparison engine. Textual comparators already compare two documents effectively, while model comparison is just starting to exist. A first approach has been implemented to compare UML models, based on the element's identifying attribute.

This paper proposes a different way, based on the comparison of all the element's attributes. It shows how this approach is backed by the information theory.

It describes the full comparison process by detailing on the comparison of two elements and the computation of the distance between them.

## State of the art

## Models and the model-oriented approach

A model is defined as an analytic and algorithmic vision, representing units of sense and their relationships. IT systems are represented by models. Model oriented development is briefly touched by Mark Kofman in his article [KOF03]. This type of development is centered on the development and the management of models. The Object Management Group [OMG] presented the concept of the Model Driven Architecture [MDA06], which consists in developing a model abstract to the runtime platform, which enables a better stability and a better maintenance of the system. OMG is a foundation which goal is to promote the object approach in the industry. This organism defined the Meta-Object Facility [MOF00]. This approach defines a common meta-model to define meta-models. In a four level architecture, it defines itself as the meta-meta level.

On of its implementations is the Eclipse Modeling Framework [EMF], which we will use during our experimentation.

It is worth noting that the MOF is using the XMI specification (standing for XML Metadata Interchange) [XMI05] to enable sharing information and serializing models to a XML file. During the serialization process, each element for each model is serialized by being given a unique identifying attribute.

Model management specifics

Kofman [KOF03] makes an interesting point when declaring that models are different from programming languages' units. A huge quantity of information is contained in only model unit. This concentration is a problem to access and modify the data by several users at the same time. This all requires more tools: comparison tools, merging tools, revision tools.

Models are represented with graphical notations; graphical differences are not necessary model differences. Models may be non-ordered resources. Order is a possible difference between two elements in a list.

Models are constituted of elements. Elements are linked to each other by relationships that are defined by their metamodel, except for the containment link, which is defined by the MOF. They can then be browsed as trees.

Elements are composed of attributes. Attributes are objects identified by a unique key (“name”, “isAbstract” …) assimilated to an attribute and a value. The value itself is typed to a simple type, a text, a number, a boolean.

Marcus Alanen [ALA03] describes the different types of differences that may be detected between models. He distinguishes the differences affecting the element and the differences between the attributes.

- Adding an element
- Deleting an element
- Modifying an element

Modifying an element is the aggregation of at least one difference between the attributes of the two elements.

- Adding an attribute “insert”
- Deleting an attribute “remove”
- Modifying an attribute “set”

Alanen adds two differences which particularly apply to ordered elements:

- Adding an attribute “insertAt”
- Deleting an attribute “RemoveAt”

## Comparison engine use

The Eclipse’s textual comparison engine [ECL] has two uses :

Compare two models: the comparison engine compares two texts and computes their differences.

Compare two models with their common ancestor: The comparison engine compares two models, using their common ancestor to determine where the changes come from. If a change is detected on both sides, it is declared as a conflict, or eventually a pseudo-conflict in the case the two elements match. (for example, an element present in the ancestor has been deleted in the compared members).

## Discussion

### *Closed circuit*

Alanen’s [ALA03] and Letkeman’s [LET05] comparison engines are based on the matching of elements using their identifying attribute.

Several scenarios show that this approach is invalid, though the most simple to implement.

A first scenario consists in adding to two copies of the same model two elements that are semantically equal using the editors associated to those comparators. The identifying strings are generated by the editors, and the comparators declare that two differences exist between the two models, corresponding to the addition of an element on both sides.

A second scenario consists in the comparison of two models developed separately. As the two models’ identifying attributes are different, the comparator reports an error and the comparison process is simply aborted. This scenario is a particular case of the first scenario.

Those problems represent important drawbacks, for the models can only be compared amongst themselves. While this is very useful to compare a model with its own history, it is very hard to manage to export or import data.

By a way of consequence, the reusability of those models is very uneasy.

### *Far from the logical model*

Alanen’s model presents difference types who are taking in account the attributes’ order, yet the elements’ order is forgotten. This is not matching the logical model, where the attributes’ order is not changing, when elements’ order is much more fluctuant.

## Information theory and the comparison process

To distinguish an element from another, one has to find the semantic elements containing a maximum of information. Information theory has been founded by Claude Shannon [SHA48]. It proposes to calculate the quantity of information linked to a word m with the formula:

$I(m) = \log_2[p(m)]$

where p(m) is the probability for the word m to appear.

The more a word appears, the more it is redundant, and less the quantity of information it carries is.

# A total comparison engine

## Hypothesis

We have seen that previous solutions to compare models suffer of serious drawbacks. We desire to implement a new comparison method which will really show the semantic changes in the model and compare effectively any model, without being based on IDs.

We propose to use the total comparison of the elements' attributes. If the comparison of all the attributes shows that the elements are near, then we can admit that the elements are near.

## Difference types

We propose to use six difference types:

Element's differences:

- Creation: an element was added
- Deletion: an element was deleted
- Order: the order of this element has changed

Attributes' differences

- Set: the value of this attribute has changed
- Add: the attribute has been added
- Remove: the attribute has been removed

## Graphical representation of a difference

Kofman [KOF03] concludes on MetaDiff by presenting his future work on representing a difference. The textual comparator shipped with Eclipse gives us a good kick start.

It defines a color code to describe the type and the origin of the difference.

The textual comparator shows a difference tree, with the same color code, using graphical objects to represent an addition, a deletion, or a modification.

## Comparing attributes

## Computation of the distance between attributes

We define the distance between attributes the difference between the values of the two attributes. The distance is a metric which value is comprised between 0 (attributes have the same value) and 1 (attributes have totally different values).

### Comparing Boolean values

If the Booleans have an identical value, the distance is 0. Else it is 1.

### Comparing character strings

We use the Levenshtein's distance algorithm to compute the distance between the character strings. The algorithm returns the number of manipulations to perform to transform one of the character strings to the other, where a manipulation is one of those operations:

Deleting a character

Adding a character

Modifying a character

To conserve a value between 0 and 1, we divide the obtained value by the maximal distance, which is the maximum out of the two lengths of the character strings.

### Comparing numbers

Numbers have a very large sense. Dates cannot be compared as integers, or an hotel room number cannot be compared with a zip code. As numbers are entered manually, they are treated like character strings, and we have preferred to process them as it.

## *Attributes' weights*

An attribute has a weight depending of the volume of information it carries. It can be computed by two factors:

The number of times the attribute is met in the model. The more an attribute is present, the less it is important.

The number of values an attribute takes. If it has only two possible values, like a Boolean, it is very unlikely to be a good identifier.

The information relative to the attributes is collected by browsing recursively the model.

## *Comparing two elements*

This algorithm below describes how two elements are compared.

```
************************************************************
*Function match
*Input : element, comparedElt
*Output : diff
************************************************************
Body :
Diff diff <- new Diff();
// we select the two elements' attributes and compare them.


AttributeResultList is a new List
while (more attributes in element)
    if (comparedAttribute exists with the same name)
        int i <- compare_values(attribute, comparedattribute, diff)
        attributeResultList.add(attribute,i); // the attribute existed in the other element and we compared them
    else
        attributeResultList.add(attribute,0); // the attribute didn't exist in the other element, we have a matching rate of 0 on it.
        diff <- missing_element;
    end if
end while

// finally we watch the other element to see if he has attributes that our element hasn't.

while (more comparedAtt in comparedElement)
    if (!attributeResultList.contains(comparedAtt))
        AttributeResultList.add(comparedAtt,0);
        diff <- new_element;
    end if
end while

// now we just calculate the total matching rate
total <- 0;
while (more attributeResult in AtttributeResultList)
    attribute_weight <-
    getAttributeWeightForName(attributeResult.name) // get the weight for the attribute

    total <- total + attributeResult.value * attribute_weight;
end while

if (rate < 0.2)

    // the rate symbolishes the difference between the two elements, it must be under 20% to pass.
    diff.potentialmatch is set to true
else
    diff.potentialmatch is set to false
end if

return diff;
```

The difference rate in this algorithm is the value of the difference between the two elements. Note that if the distance is less than an arbitrary value (here 20 %), the elements are considered as a potential couple.

### Local weights

As to respect the distance metric, the attributes weights are constricted between 0 and 1 by dividing their value by the sum of the weights value for the attributes present on the two elements.

### *Comparing two models*

```
****************************************************************
*Function arrangeCouples
*Input : children, comparedChildren
*Output : couples
******************************************************************

Body :
couples is a new list
potentialcouples is a new list
while (more child in children)
    while (more comparedChild in comparedChildren)
        Diff diff = match(child, comparedChild);
        if (diff.potentialmatch is true)
            // we place the couple and diff in the potential
            couples list
            potentialcouples.add(child, comparedChild, diff);
        end if
    end while
end while

// we have all the potential matching couples
// we reorder them by the matching rating in increasing order

sort_on_rating_in_increasing_order(potentialcouples)

while(more potential in potentialcouples)
    if (!one_member_in(couples))
        couples.add(potential)
        differencesList.add(potential.diff) // add the
        differences found for the couple
    end if
end while

// we finally take in account the remaining singles

while (more child in children)
    if (!in couples)
        differencesList.add(new Diff(missing_element));
    end if
end while

while (more child in comparedChildren)
    if (!in couples)
        differencesList.add(new Diff(new_element));
    end if
end while
```

Models are compared recursively. Elements for every level are compared with every element for their level, using the algorithm above.

Once the elements are compared, they are ordered depending on their distance in an increasing order. Couples are then computed starting with elements with a little distance.

Potential couples which contain at least one element already locked are deleted. Single elements remaining are added to the list of couples by specifying that the difference between the two elements is an addition or a deletion, depending if the element is in the local model or the compared model.

The elements contained by the couples form the next recursive level.

### *Three-way comparison*

```
*****************************************************************
*Function threeWayMatch
*Input : left, right, ancestor
*Output : diff
*****************************************************************
Body : diff <- new diff();
// adapted algorithm from the Differencer class.
if (ancestor == null)
    if (left == null)
        if (right == null)
            // nothing
        else
            diff <- added element on right;
        endif
    else
        if (right == null)
            diff <- added element on left;
        else
            if (left == right)
                diff <- added element on left, added element on right, pseudo conflict;
            else
                diff <- added element on left, added element on right, conflict;
            endif
        endif
else
    // ancestor != null
    if (left == null)
        if (right == null)
            diff <- element deleted on left, element deleted on right, pseudo_conflict;
        else
            if (right == ancestor)
                // left is null and right == ancestor
                diff <- element deleted on left;
            else
                // left is null and change on right
                diff <- element deleted on left, match(ancestor, right);
            endif
    else
        if (right == null)
            // right is null and left is non null
            if (perfectMatch(left, ancestor))
                // right is null and left == ancestor
                diff <- element deleted on right;
            else
                // right is null and change on left
                diff <- element deleted on right, match(ancestor, left);
            endif
        else
            boolean al <- match(ancestor,left) == 0;
            boolean ar <- match(ancestor,right) == 0;
            // left and right are non null
            if (al && ar)
                // the three EObjects are equal, there is nothing to do.
            endif
            if (al && !ar)
                diff <- match(ancestor, right);
            endif
            if (!al && ar)
                diff <- match(ancestor, left);
```

```
if (!al && !ar)
// conflict or pseudo conflict
leftdiff<- match(ancestor, left);
rightdiff <- match(ancestor, right);
directdiff <- match(left, right);
diff <- mergeAndTagPseudoConflicts(leftDiff, rightDiff, directDiff);
endif
endif
endif
endif
endif
endif
```

The three-way comparison process consists of the comparison of two elements using a common ancestor. This type of comparison appears often when version control systems are used, particularly to resolve conflicts. We have reused the structure of the Differencer algorithm [DIF06] and the two-way comparison process.

## Results and future work

See the annex for a screenshot.
Differences are shown in the upper part of the editor. The common ancestor is optionally shown in the middle; the two models are present in the lower part of the screen.
Differences are highlighted by reusing the textual comparator feature which consists in tracing lines between the matched elements.

## Unit tests

The comparison engine has been tested by using a set of unit tests, Each test was composed of two models separated by only one difference, and was used to detect a certain difference type.

## Semantic issues

Compare results are satisfying and represent the real differences between the elements. The problem of this tree comparison is that is is not convenient for comparing flow charts diagrams (like BPMN diagrams). Some relationships are ignored or handled in an invalid way.
For now, those links represent a source of errors and are treated as recursive links. In the future, we should be able to browse the diagram while detecting references, and avoiding them.

### *Performance*

The engine was not built thinking in terms of performance, and the number of comparisons between elements is very high. The computation time may go up to two minutes to compare two models containing one hundred elements. To reduce this time, we propose to stop comparing elements which have found a perfect matching. We will reduce the number of comparison for one level.

## Conclusion

The model-oriented development needs tools to manage models. A model comparator is one of these tools. This paper presents a new approach to compare models. Instead of being based on the identifiers, the comparison algorithm uses the information theory to analyze the model's semantic domain. It establishes a matching between elements, and proceeds to their children recursively. We have created a model comparison engine which implements this approach, and which gives a visualization of the differences, their origin and their types. Our work will now focus on enhancing the engine's performances to be able to process more data. We will also think of an approach for diagram-based models, which tree representation does not make particular sense.

## References

[ALA03] ALANEN, Markus and PORRES, Ivan, Difference And Union Of
Models, 2003
[BPMN] OMG, Business Process Modeling Notation Specification, 2004
[DIF06] IBM, Class Differencer - Eclipse Platform API Specification, 2006,
http://help.eclipse.org/help32/index.jsp?topic=/org.eclipse.platform.doc.isv/reference/api/org/eclipse/compare/structuremergeviewer/Differencer.html
[ECL] Eclipse Foundation, Eclipse Project, 2006, http://www.eclipse.org
[EMF] Eclipse Foundation, Eclipse Tools - EMF Home,
http://www.eclipse.org/emf/
[KOF03] KOFMAN, Mark, MetaDiff - a Model Comparison Framework, 2003
[LET05] LETKEMAN, Kim, Comparing and merging UML models in IBM

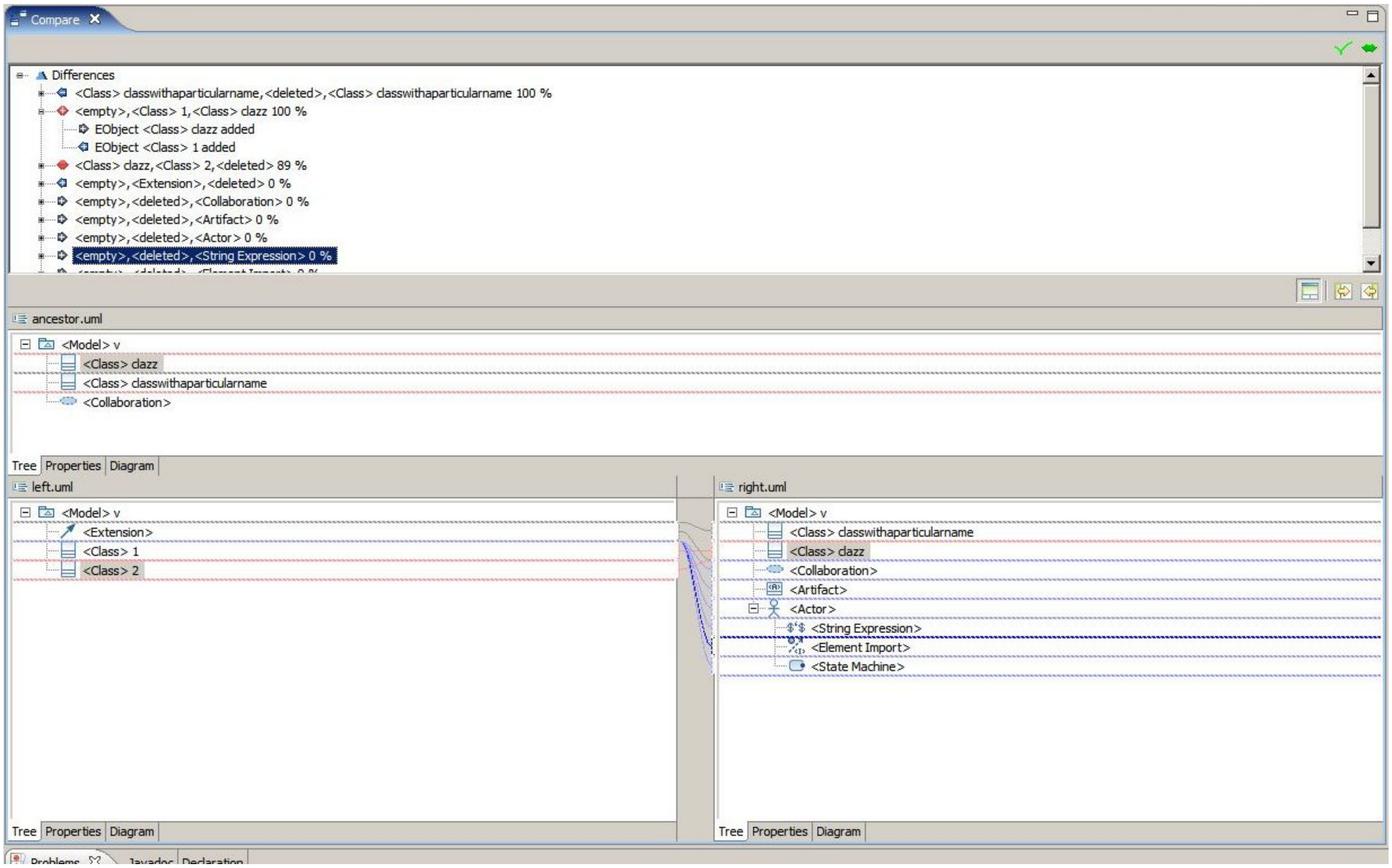
[MDA06] OMG, Model Driven Architecture Specification, 2006,
http://www.omg.org/mda/ Rational Software Architect, 2005

[OMG] Object Management Group, 2006, http://www.omg.org

[SHA48] SHANNON, Claude, A Mathematical Theory of Communication, 1948

[XMI05] OMG, XMI (XML Metadata Interchange) Specification, 2005

## Annex



```
Differences
  <Class> classwithaparticularname,<deleted>,<Class> classwithaparticularname 100 %
  <empty>,<Class> 1,<Class> clazz 100 %
    EObject <Class> clazz added
    EObject <Class> 1 added
  <Class> clazz,<Class> 2,<deleted> 89 %
  <empty>,<Extension>,<deleted> 0 %
  <empty>,<deleted>,<Collaboration> 0 %
  <empty>,<deleted>,<Artifact> 0 %
  <empty>,<deleted>,<Actor> 0 %
  <empty>,<deleted>,<String Expression> 0 %
  <empty>,<deleted>,<Element Import> 0 %
```

ancestor.uml

```
<Model> v
  <Class> clazz
  <Class> classwithaparticularname
  <Collaboration>
```

left.uml

```
<Model> v
  <Extension>
  <Class> 1
  <Class> 2
```

right.uml

```
<Model> v
  <Class> classwithaparticularname
  <Class> clazz
  <Collaboration>
  <Artifact>
  <Actor>
    <String Expression>
    <Element Import>
    <State Machine>
```