# MCTest: Towards an improvement of match algorithms for models



Vicente García Díaz; Begoña Cristina Pelayo Garcia-Bustelo; Oscar Sanjuán Martínez; Edward Rolando Núñez Valdez; Juan Manuel Cueva Lovelle. MCTest: towards an improvement of match algorithms for models. The Institution of Engineering and Technology. 6 - Issue 2, pp. 127 - 139.IET Research Journals, 2012. DOI: https://doi.org/10.1049/iet-sen.2011.0040

# MCTest: Towards an improvement of match algorithms for models

Vicente García-Díaz* – garciavicente@uniovi.es; B. Cristina Pelayo G-Bustelo* – crispelayo@uniovi.es; Oscar Sanjuán-Martínez* – osanjuan@uniovi.es;Edward Rolando Núñez Valdez* – nunezedward@uniovi.es; Juan Manuel Cueva Lovelle* – cueva@uniovi.es

***University of Oviedo, Department of Computer Science, Sciences Building, C/Calvo Sotelo s/n, 33007 Oviedo, Asturias, Spain**

**Abstract** Due to the increasing importance of Model-Driven Engineering and the changes experienced by software systems over their life cycle, the calculation, representation, and visualization of matches and differences between two different versions of the same model are becoming more necessary and useful. This work shows the need for improvement in the algorithms for calculating the relationships between models and presents a tool to test different implementations, thus reducing the effort required to measure, compare, or create new algorithms. To demonstrate the need for improvement and the framework developed, we have created different models that conform to the metamodel of a domain-specific language. Subsequently, we compared these models using the algorithms of the EMF Compare tool, part of the Eclipse Modeling Project, which is the framework of reference for Model-Driven Engineering. Thus, in the case study, our tool is used to measure the quality of the comparisons performed by EMF Compare.

**Keywords** testing tools, reuse models, domain engineering, match algorithms

# 1 Introduction

Model-Driven Engineering (MDE) [1] is an important approach in software engineering to increase the level of abstraction of the development tasks. Thus, the programming language used is closer to the concepts of the problems, that is, the knowledge in a specific domain that can be reused to translate it, directly or pseudo directly, into programming logic. For this purpose, models and metamodels are used. A model, based on the principle of "everything is a model" [2], could be any type of artifact. Metamodels allow us to increase the reusability, portability, and interoperability of systems [3]. The Model-Driven Architecture (MDA) [4] standard is evidence of the success of MDE.

Model Management aims to provide a way to represent the relationships between models using a set of operators. For example, Merge is used to create unions between models based on their relationships [5]. Match and Diff are also very interesting because they are used to calculate relationships, which are the basis for any operation between models. They serve many useful functions, such as the following: 1- correct detection of new versions in version control systems for models [6], 2- verification of the correctness of transformations between models [7], 3- investigation of the change patterns of software evolution, exploring the underlying motivations behind them, and guiding future development and maintenance activities [8], 4- detection of similarities between different variants [9], 5- supporting reuse [10], and 6- saving space, bandwidth, and communication time [11], among others.

The problem of determining the relations between models is inherently complex and involves three main steps [12]: 1- the calculation algorithm, 2- representation in a format that enables easy computer manipulation, and 3- visualization in a human-friendly format. This work focuses on the first and most crucial step, that is, the algorithm for calculating the relations, which is a difficult task that has no a single best solution because it

depends on each particular problem [13]. In particular, in this first stage, we will address the Match operator, usually applied as a precursor to Diff.

The aim of this paper is to demonstrate the need for further study of the systems and algorithms for model comparison and to present a tool to help in that study. To this end, we created a case study by using a domain-specific language (DSL). The proposed tool does not provide an algorithm for comparing models but for studying algorithms. It is used to evaluate different match algorithms quickly and effectively, using a user-friendly interface to display the results. Furthermore, there is no other tool available for comparing, measuring or helping in the creation of match algorithms.

The remainder of this paper is structured as follows: in Section 2, we present a brief overview of the relevant state of the art; in Section 3, we describe our case study; in Section 4, we present the main features of the tool designed to compare different match algorithms for models; in Section 5, we discuss an evaluation of the developed tool through its use with the case study; and finally, in Section 6, we indicate our conclusions and the future work to be done.

## 2 Approaches for comparing models

Many different algorithms could be used to compare models. The first algorithms were developed to compare only text files (e.g., Hunt and Mcllroy [14], Hirschberg [15], Myers [16]). Although these proposals serve their purposes, they use a unit of version based on the file and a unit of control based on the paragraph level, which are incorrect abstractions for working with models because they use different data structures [6].

From the point of view of models, the simplest approach is to assume that each element has a persistent universally unique identifier (UUID) assigned at the time of its creation (e.g., Alanen and Porres [11], Farail et al. [17]). This assumption is very optimistic because it cannot be applied to models that are constructed

independently of each other or in tools that do not support it. In addition, the use of UUIDs does increase the size of the models. However, in cases in which it can be applied, it allows models to be compared quickly and easily.

The next step was the use of a technique based on dynamic identifiers calculated on the basis of certain values of the model, using functions defined for that purpose [18]. This approach allows us to compare models that have been constructed independently, but it has the disadvantage that it is not easy in such a system to define functions for comparing the individual model elements for each specific problem that may exist.

There are also approaches that use trees with ordered children (e.g., Tai [19], Cobena et al. [20]) or disordered children (e.g., Zhang and Shasha [21], Wang [22]). Trees are a special type of graph that cannot contain cycles. However, the nature of models makes trees too restrictive to represent them. Therefore, the latest generation of algorithms are based on treating the models as graphs, which is not trivial [23], and making comparisons by looking for similarities between the elements of models using heuristics. To this end, there are two main groups of algorithms: independent of the metamodel and dependent on the metamodel. The latter could deliver better results because they are more specifically focused on a particular problem.

## *2.1 Comparison depending on the metamodel*

Nejati et al. [5] proposed a series of algorithms for matching using heuristics at the terminological, structural, and semantic levels. They worked only with state diagrams describing behavior issues for the telecommunications industry. Ohst et al. [24] proposed a method to identify the differences between versions of UML diagrams, placing strong emphasis on the representation of these differences.

The main disadvantage of these works is that the proposed algorithms, and the tools in which they are implemented, are specific to the type of model. The same drawback is found in other works, such as Xing and

Stroulia [8] or Selonen [25]. The fact of focusing on UML or any other specific metamodel, such as the Ontology Definition Metamodel (ODM) [26], restricts the algorithms to only the chosen metamodel. Moreover, models from an arbitrary DSL are typically more general than, for example, UML models. Thus, they often have different structure, syntax, and semantics [27], therefore requiring a different treatment.

## *2.2 Comparison independent of the metamodel*

Several works have sought to show how to compare models following a metamodel independent approach. For example, Lin et al. [27] presented an algorithm and a tool (DSMDiff) that calculates the differences and similarities between two models of an arbitrary DSL. Some features of that work include the following:

- It is based on the Generic Modeling Environment (GME) [28].
- It uses edited scripts to represent the comparison between models.
- It examines the structural similarity between models in a specific and localized region, in which the node to be compared is the center and its immediate neighbors are the edge, restricting the capacity of the algorithm.
- It uses a heuristic in which the existence of a name attribute in each element of the model is very important.

Other interesting work in this field was done by Melnik et al. [29], who presented a matching algorithm based on a fix point computation that can be used across different scenarios; Mandelin et al. [9], who presented a framework based on Bayesian methods for finding model correspondences automatically; Selonen and Kettunen [30], who presented a flexible approach for inferring correspondences between model elements; Treude et al. [31], who presented a technique for computing differences between models an order of magnitude faster than other algorithms, using a high-dimensional search tree for efficiently finding similar model elements; and

Rivera and Vallecillo [32], who presented a metamodel to represent differences between models, also using a model as a central element.

### 2.2.1 EMF Compare

Among the implementations of MDE frameworks, the one with the greatest success in both business and academia is the Eclipse Modeling Project (EMP) [33], located within the Eclipse Platform[1] and in which they are implemented many of the specifications promoted by the Object-Management Group (OMG)[2], particularly those having to do with the MDA [34] specification. In EMP, the meta-metamodel used is called Ecore, which it is so close to the industry standard called Meta-Object Facility (MOF) [35] that it is considered as a core implementation of MOF, that is, the Essential MOF (EMOF). Authors such us Gruschko et al. [36] justify the use of Ecore saying that it is the de facto standard in the industry.

The core of the Eclipse Modeling Project is the Eclipse Modeling Framework (EMF) [37]. The EMF Compare tool [38], according to its authors, "brings model comparison to the EMF framework, and provides generic support for any kind of metamodel to compare and merge models. The objectives are to provide a stable and efficient generic implementation of model comparison and to provide an extensible framework for specific needs." Moreover, its output is based on models, which facilitates further manipulations. Brun and Pierantonio [12] commented that the EMF Compare arose in 2006 based on the need for a model comparison engine within the Eclipse Platform. Because of the capabilities and possibilities of this tool (1- it is integrated into the EMP, 2- it supports any Ecore metamodel, 3- it incorporates predefined heuristics to compare models, and 4- it offers extension mechanisms), we have chosen it to demonstrate our case study and to evaluate our tool for testing

[1] The Eclipse Foundation. http://www.eclipse.org/ (January 14, 2011)

[2] Object Management Group. http://www.omg.org/ (January 13, 2011)

match algorithms because it is considered the facto standard for comparing models. Our tool is called Model Comparison Testing Engine (MCTest). Note, however, that MCTest is independent of the technology used and can therefore be used with any desired algorithm.

# 3 Case study

To illustrate the need for continued work on the issues related to model comparison, Fig. 1 shows a grammar for a DSL, created using Xtext [39]. Depending on the grammar, artifacts are automatically generated, for example, an integrated development environment (IDE) to work with the language on the Eclipse Platform, a parser to process instances of the language, or the corresponding Ecore metamodel [37]. Following the MDE principles, developers use the generated IDE for modeling, thus creating, transparently for them, instances of the inferred metamodel.

Domainmodel.xtext

```
grammar org.xtext.dsls.Domainmodel with
 org.eclipse.xtext.common.Terminals

generate domainmodel
 "http://www.xtext.org/dsls/Domainmodel"

DomainModel:
    (elements+=AbstractElement)*;

AbstractElement:
    PackageDeclaration | Type | Import;

PackageDeclaration:
    'package' name=QualifiedName '{'
        (elements+=AbstractElement)*
    '}';

Import:
    'import' importedNamespace=
     QualifiedNameWithWildCard;

QualifiedName:
    ID ('.' ID)*;

QualifiedNameWithWildCard:
    QualifiedName '.*'?;

Type:
    Entity | DataType;

DataType:
    'datatype' name=ID;

Entity:
    'entity' name=ID ('extends'
     superType=[Entity])? '{'
        (features+=Feature)*
    '}';

Feature:
    name=ID ':' type=TypeRef;

TypeRef:
    referenced=[Type] (multi?='*')?;
```

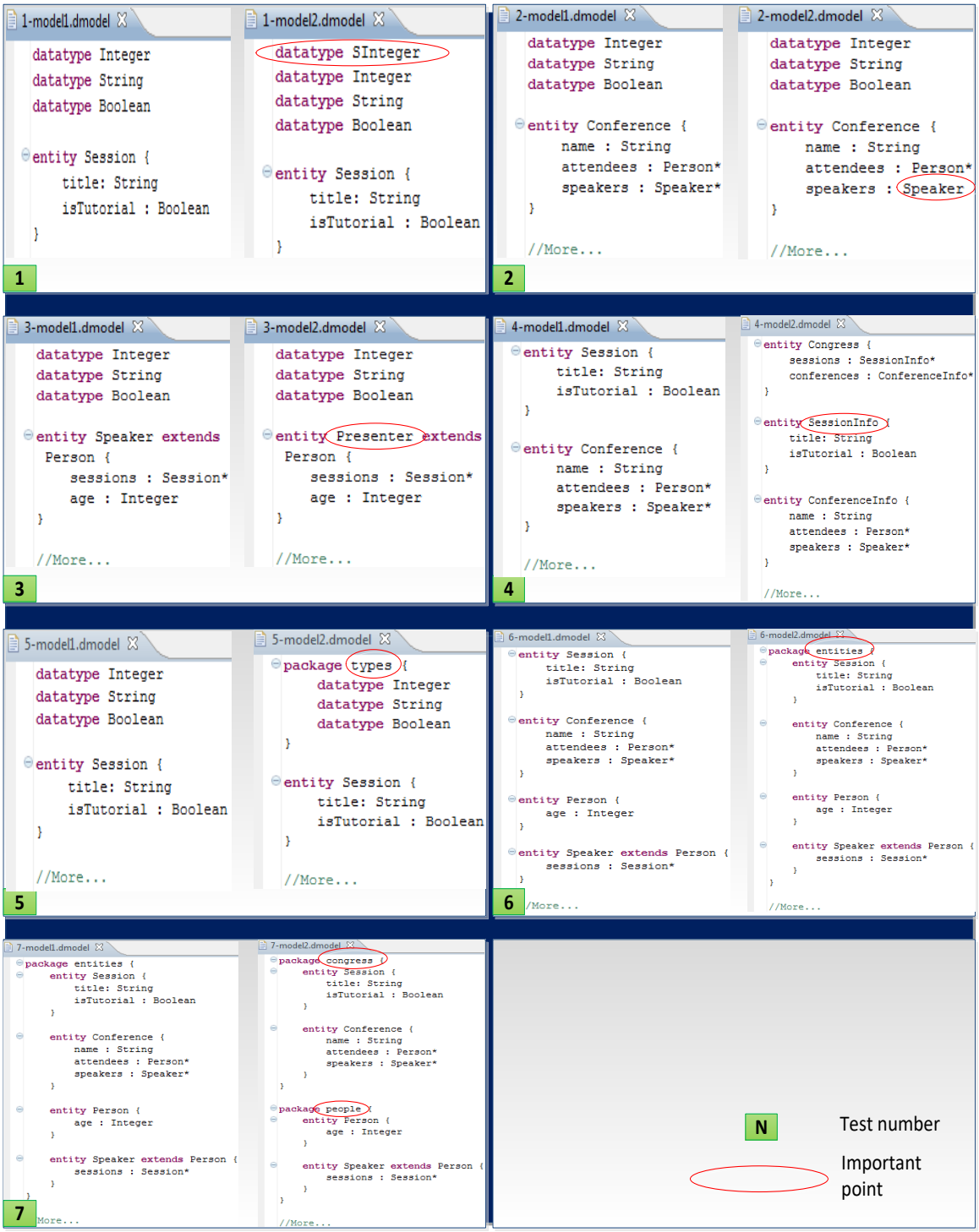
**1. Grammar of the domain-specific language**

Created models normally undergo evolution over time, which leads to new versions of those models. This is

where it is interesting to calculate the matches between versions of each model to be able to perform further actions in an optimal way.

The evaluation consisted of making changes in a model to determine whether EMF Compare works properly. In a large number of tests, the tool (using its generic algorithm) correctly detected changes. However, Fig. 2 shows examples for which we did not obtain an optimal result:

- In the first case, instead of detecting the new data type *SInteger*, the system detects that there is a change from the original *Integer* data type to the *SInteger* data type, and then it detects the insertion of a new *Integer* element.
- In the second case, instead of detecting that the multiplicity of the data type *Speaker* has changed, it detects that there are two different items.
- In the third case, instead of detecting the change of name from *Speaker* to *Presenter*, it detects that an item has been removed and another has subsequently been added.
- In the fourth case, instead of detecting the change of name from *Session* to *SessionInfo*, it detects that an item has been removed and another subsequently added. However, for *Conference* and *ConferenceInfo*, although they are very similar to the previous case, the tool detected the change correctly.
- In the fifth case, instead of detecting that three data types have been moved to a new package, it detects that these types have been removed from the original model.
- In the sixth case, instead of detecting only the new package, it detects all of its elements as new.
- In the latter case, instead of detecting the change of name from entities to congress and the incorporation of a new package with some of these elements, it detects that all of the elements are new.




2. Test performed

It seems that it should not have been difficult to have found the optimal solutions in these seven cases, based on the semantics of the language, thus:

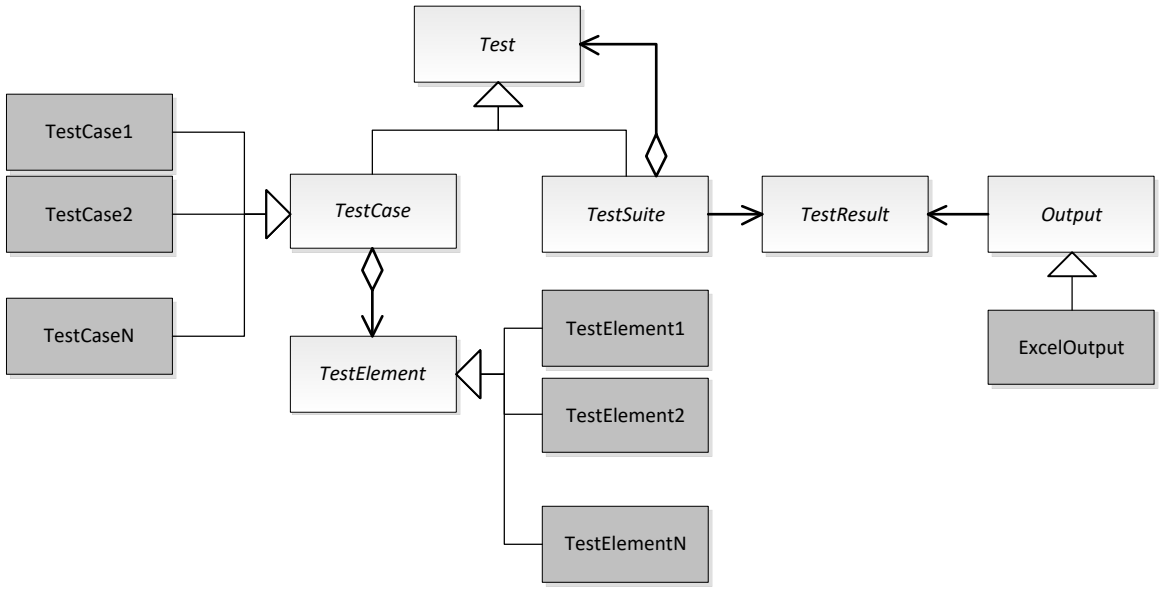
- In the first case, it seems obvious that if there is only one change in the model and this change is caused by an element that also has a different name from the others, then this element should be a new one.
- In the second case, if only one attribute (multi) of a specific *TypeRef* instance is changed, then it seems that the most realistic hypothesis is that there is only a change and not a different element.
- In the third case, simply renaming from *Speaker* to *Presenter* and retaining all of its internal structure, it makes sense to think that there has only been a change of name in this element. Moreover, with tools like RiTa.WordNet [40], a Java class library that allows us access to the famous English vocabulary ontology called WordNet [41], we could use a non-typographical linguistic approach, so instead of looking for exact matches between the names of the elements, it could consider the possible correlations that may exist between words. RiTa.WordNet is a freely distributed system that can, among other things, measure the semantic similarity or relationship between two concepts, the names of the instances in our case.
- In the fourth case, a phenomenon very similar to the third case occurs. The only difference is that in the fourth case, the comparison algorithm correctly decided that *ConferenceInfo* is not a new element but failed to detect that *SessionInfo* is new. The success is due to the greater number of letters in *ConferenceInfo* that are equal to the original name (*Conference* length > *Session* length).
- In the fifth case, three items are moved into a new package. Not having made any changes to these elements, it seems logical to presume that these elements have been moved to the new package instead of thinking that they have been removed and then three clones have been re-created.
- In the sixth case, a case similar to the previous one occurs, although it seems even more obvious in this

case because the number of elements that were moved is much larger.

- In the latter case, a mixture of some of the previous cases occurs. This time, we moved elements to a new package and then we renamed the other one.

# 4 Framework overview

Fig. 3 shows, very briefly, the overall architecture of MCTest. Each global test consists of one or several test suites, which function as classifiers of each concrete test, similar to the behavior of the xUnit tools [42]. However, the xUnit tools are intended to test traditional programming languages. The MCTest architecture is inspired by the design of JUnit[3], possibly the best known, most commonly used, and extended tool for testing software developed using the Java programming language.




**3. Framework overview**

The difference is that JUnit is designed mainly for unit testing of software, to compare the obtained result with

[3] Resources for Test Driven Development. JUnit.org. http://www.junit.org/ (November 13, 2010)

the expected result through the use of asserts according to the criteria chosen by the tester. MCTest, by contrast, is designed for testing model comparison algorithms to compare the newly obtained result with the previously established optimal result according to the criteria chosen by the person responsible for that task. For example, the expert in a particular domain, for which models are made, would be the best person to say when two models made for a given domain have equivalent results, although at first glance they may not be identical.

Just as models raise the level of abstraction compared to traditional programming languages, MCTest does testing at a higher level of abstraction compared to tests performed by previously existing tools. The idea is not to test source code but rather to test model comparison algorithms and thus compare the models obtained after the application of the algorithms. MCTest aims to fill a gap in model comparison tools and can be used to improve the results obtained by algorithms.

## *4.1 Test cases and test elements*

A test case functions as a container to enter test elements. Here, instead of using asserts, MCTest users extend the abstract class called *TestCase*. This is necessary to easily provide the optimal comparison between two models (the first one is the source, and the second one is the target). From this optimal comparison, we can try out as many algorithms as desired, using the so-called test elements. To create a test element, we just have to extend the abstract class called *TestElement* and implement the desired algorithm in such a class. The MCTest runtime will perform the relevant operations, comparing the obtained results with the expected result in each test case.

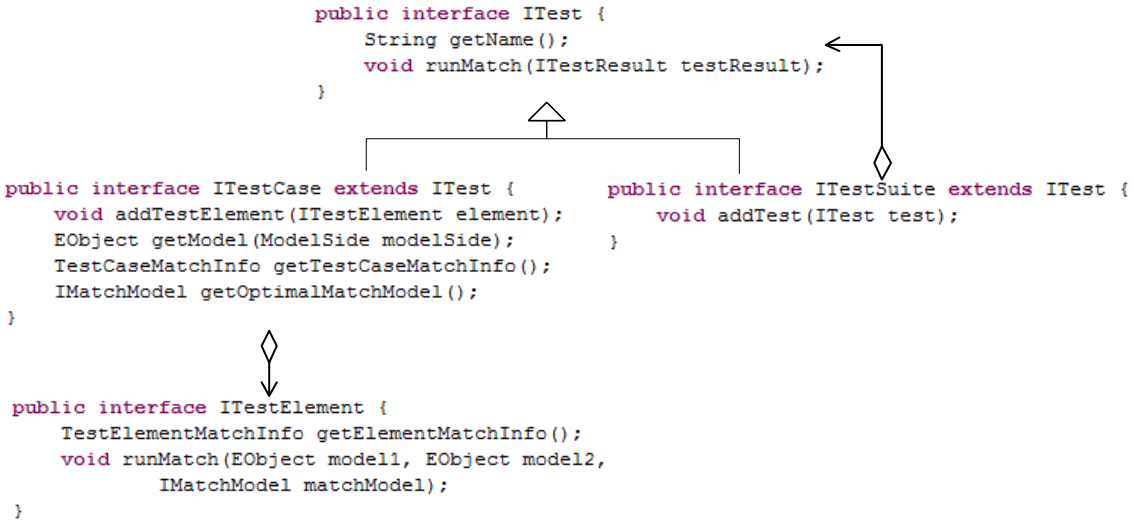
In this first version of MCTest, a test element of a test case is equivalent to a match algorithm. In other words, each test case has assigned one or more algorithms whose results are compared with the optimal result. A test element encapsulates the underlying algorithm, enabling all the elements to be exchanged and used freely with

any previously designed test case, and without any dependency on the test case. In addition, each test case uses the same interface for both the optimal result and the obtained results of each test element, using the same data structure, regardless of the true origin of the test element (e.g., EMF Compare, algorithms of other tools, algorithms that are still being developed, research works, etc.).

With this decoupled design, we can create test groups independently of the algorithms that already exist or that will exist in the future. Thus, we create test collections (test cases) that can always be expanded with the addition of new algorithms (test elements). This process is analogous to the regressive testing performed by xUnit tools, which is highly valued in software engineering [43]. Note that each test element can be run with any test case, so the number of combinations to be tested increases exponentially without any additional effort.

The key to creating test suites, test cases, and test elements is the use of the Composite design pattern [44] (Fig. 4).

Each test (*ITest*) has a name that identifies it from others and a method used to execute all possible subtests contained in it. The output is provided by the *ITestResult* interface. The *ITestSuite* interface allows to add, in a transparent manner, as many suite or test cases as desired.




**4. Basic elements for creating tests**

Fig. 5 shows a fragment of a test case. To facilitate the work, MCTest provides a basic implementation of the *ITestCase* interface, called *TestCase*. Therefore, users of the tool only need to extend the *TestCase* class by implementing two methods:

- *getTestCaseMatchInfo*. It gives feedback information of the test case (e.g., an identifying name).
- *getOptimalMatchModel*. It gives the optimal output of the comparison between two models through the *IMatchModel* interface, which provides methods for identifying if two elements are similar (or not) and to what extent they are the same. Fig. 5 shows a fragment of how to iterate through the two models. For this, the main technologies included in the EMF [37] and the Ecore meta-metamodel are used. For each pair of elements in the models, it is possible to specify whether both elements fit by using the methods `void addMatchedElement(MatchedElement rmatchElement)` and `void addUnmatchedElement(UnmatchedElement runmatchElement)`.

```
public class TestCase1 extends TestCase {

    @Override
    public IMatchModel getOptimalMatchModel() {
        IMatchModel matchModel = new MatchModel();
        DomainModel domainModel1 = (DomainModel)this.getModel(ModelSide.Model1);
        DomainModel domainModel2 = (DomainModel)this.getModel(ModelSide.Model2);
        MatchedElement matchElement = new MatchedElement();
        UnmatchedElement unmatchElement = new UnmatchedElement();
        matchElement.setModel1HashCode(domainModel1.hashCode());
        matchElement.setModel2HashCode(domainModel2.hashCode());
        matchModel.addMatchedElement(matchElement);

        for (AbstractElement abstractElement : domainModel1.getElements()){
            if (abstractElement instanceof DataType){
                DataType dataType1 = (DataType)abstractElement;
                for (Object o :
                    EcoreUtil.getObjectsByType(domainModel2.getElements(),
                    DomainmodelFactory.eINSTANCE.createDataType().eClass())){
                    DataType dataType2 = (DataType)o;

        ...
```

- **5. Fragment of a test case**

Fig. 6 shows a fragment of a test element. MCTest users only have to implement two methods of the *ITestElement* interface to incorporate new algorithms:

- *getElementMatchInfo*. It gives feedback information of a concrete test element (e.g., an identifying name or the time it takes to run its algorithm).
- *runMatch*. It gives the output of the comparison between two models through the *IMatchModel* interface and the use of a specific algorithm. As the tool was developed using the Java programming language, the only current requirement for the algorithm is that it needs to be programmed using Java, but it is possible to use any available technology. Fig. 6 provides an excerpt of how to execute the default comparison algorithm incorporated in the EMF Compare tool [38] through its *GenericMatchEngine*.

```
public class TestElementEMFCompare implements ITestElement {
    TestElementMatchInfo matchInfo = new TestElementMatchInfo();

    @Override
    public void runMatch(EObject model1, EObject model2, IMatchModel matchModel) {
        try {
            IMatchEngine matchEngine = new GenericMatchEngine();
            MatchModel match = matchEngine.modelMatch(model2, model1, null);

            List<UnmatchElement> unmatches =
                new ArrayList<UnmatchElement>(match.getUnmatchedElements());
            for (UnmatchElement u : unmatches){
                getUnmatches(u.getElement(), u.getSide(), matchModel);
            }

            List<MatchElement> matches =
                new ArrayList<MatchElement>(match.getMatchedElements());
            for (MatchElement m : matches){
                getMatches(m, matchModel);
            }

    ...
```

- **6**. Fragment of a test element

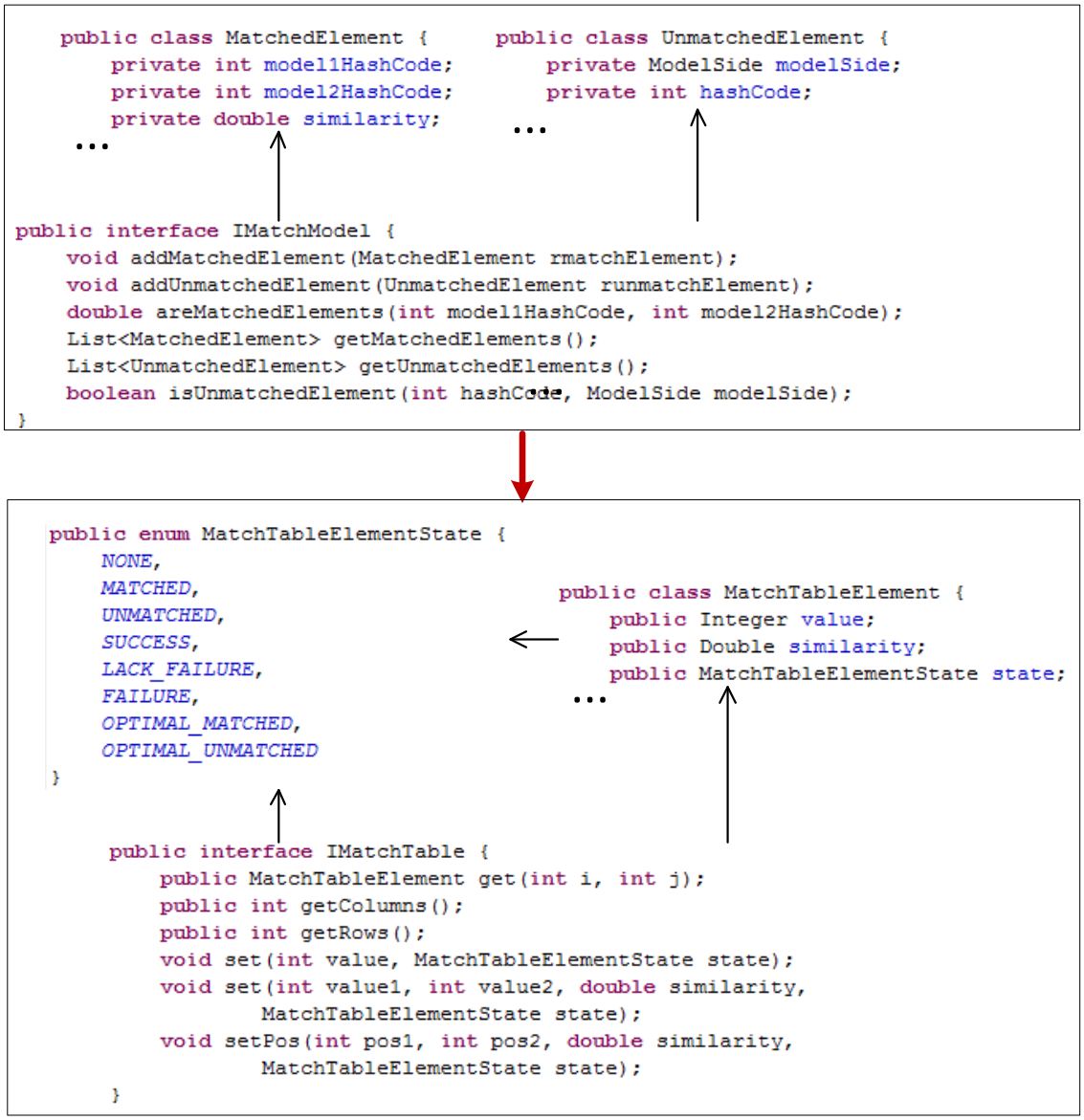
## 4.2 Match infrastructure

To study different algorithms for model comparison it is very important to have an infrastructure to store all necessary information. Fig. 7 shows the basic elements. The *MatchModel* class, default implementation of the *IMatchModel* interface, provides a structure for storing *MatchedElement* and *UnmatchedElement* objects. A

*MatchedElement* refers to two elements of two models (source and target) that have a correspondence between them. The range of values can be from 0 to 1, with 1 representing an exact match. Besides, an *UnmatchedElement* refers to an element of a model (source or target) that does not have a correspondence with any other model element. All the elements are uniquely identified by the hash code obtained by processing them with the Java programming language.

To provide personalized feedback, all the information finally becomes a structure that corresponds to the *IMatchTable* interface. Such a structure stores the basic information of all the elements, that is, each identifier, each level of similarity with other elements if any, and each state. The state of an element depends on the type of table shown at a time and it can be:

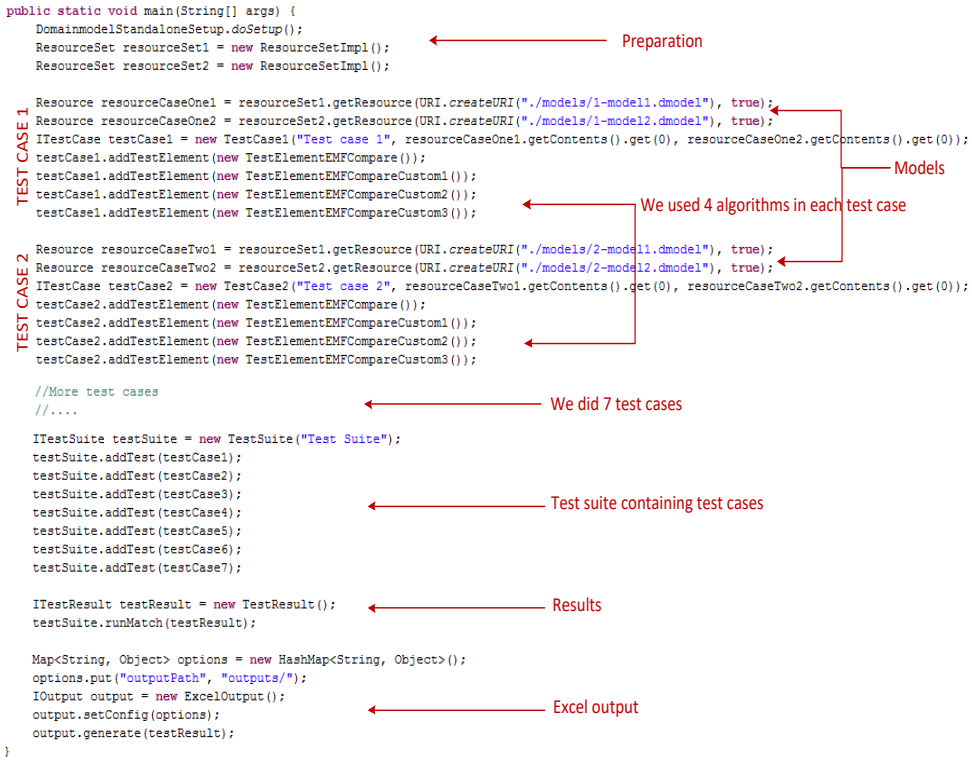
- NONE. When a relation is not yet confirmed.
- OPTIMAL_MATCHED, OPTIMAL_UNMATCHED. When there are (or not) relations between elements according to the optimal algorithm.
- MATCHED, UNMATCHED. When there are (or not) relations between elements according to the test elements.
- SUCCESS, FAILURE, LACK FAILURE. It is used to make comparisons between the optimal algorithm and each of the test elements.




- **7. Basic elements for the match subsystem**

## *4.3 Creating test programs*

Fig. 8 shows the source code needed to create a complete test, which is very compact. The first line is used to initialize the necessary components to work with the DSL shown in the case study (Fig. 2). The next two lines are used to create sets of resources, used by EMF to work with software models.




8. Test created using MCTest

In the figure, we only show two of the seven test cases, but the process of creating test cases is always the same. For example, in the first test case, two models, which correspond to the two models of the first case of Fig. 2, are loaded. Subsequently, a test case is created, passing as parameters a name and the two loaded models. The name of the class used for the first test case is *TestCase1*, as it is the class that defines the optimal solution that is expected from the comparison of both models using the method `public IMatchModel getOptimalMatchModel()`. The next step consists of adding test elements to evaluate them through the use of the MCTest engine. Fig. 8 shows that in this example, the same algorithms are compared in the two test cases, but it would not always be so. Furthermore, in the second test case, the loaded models are different. These models correspond to the second case of Fig. 2.

After defining the test cases, the next step is to create test suites to organize them. To keep the example simple, we have only created one suite, which includes all the test classes. Further, the interface called *ITestResult* contains the structure of the classes in which the feedback is kept. These classes, by default, do not provide a user-friendly interface. For this reason, the interface called *IOutput* is used to decorate the feedback with a non-programmatic technology. By default, MCTest offers output in Excel[4] format with all relevant information, but it could be easily extended with other technologies.

## *4.4 Default feedback*

Initially, MCTest offers a range of information as output, which is easily adaptable and expandable: 1- the internal structure of the 2 models that are compared in each test case, 2- the sizes of the models that are compared, 3- the optimal result of the comparison of the two models, 4- the results obtained for each algorithm studied, 5- the results obtained compared to the optimal result for each algorithm studied, 6- the successes and failures of each algorithm, 7- the success rate, 8- the time spent by each algorithm, and 9- a summary table for each test case.

**Model 1**

| | | |
|---|---|---|
| 12254719 | DomainModel | |
| | elements | [27916412, 11832081, 1497769, 25120699, 20275290, 32643052] |
| 27916412 | DataType | |
| | name | Integer |
| 11832081 | DataType | |
| | name | String |
| 1497769 | DataType | |
| | name | Boolean |
| 25120699 | Entity | |
| | name | Session |
| | superType | [] |
| | features | [8993129, 31742556] |
| 8993129 | Feature | |
| | name | title |
| | type | [21694491] |
| 21694491 | TypeRef | |
| | multi | false |
| | referenced | [11832081] |
| ... | | |

**Model 2**

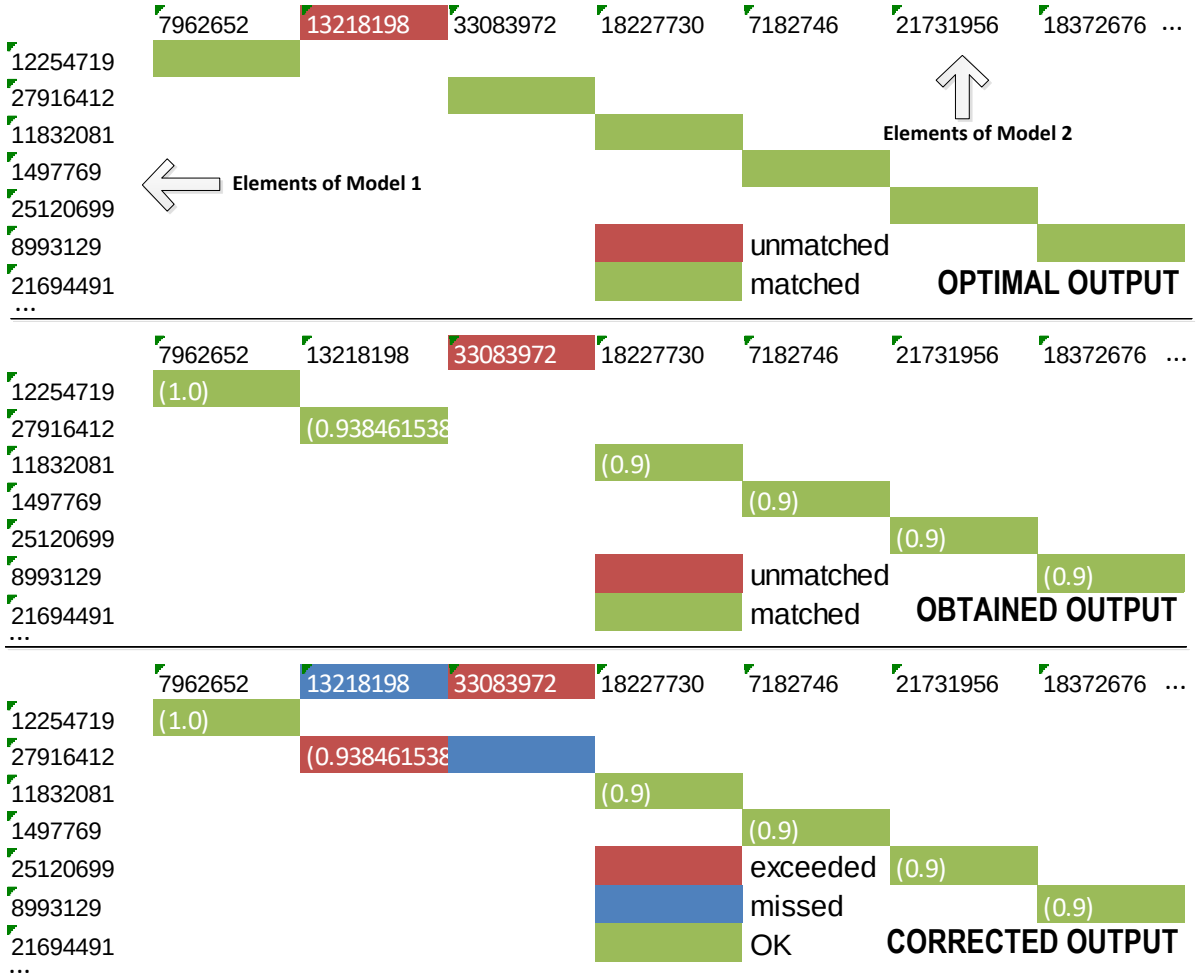
| | | |
|---|---|---|
| 7962652 | DomainModel | |
| | elements | [13218198, 33083972, 18227730, 7182746, 21731956, 21555096, 16822261] |
| 13218198 | DataType | |
| | name | SInteger |
| 33083972 | DataType | |
| | name | Integer |
| 18227730 | DataType | |
| | name | String |
| 7182746 | DataType | |
| | name | Boolean |
| 21731956 | Entity | |
| | name | Session |
| | superType | [] |
| | features | [18372676, 15229036] |
| 18372676 | Feature | |
| | name | title |
| | type | [16865950] |
| ... | | |

**9. Fragment of the model representation**

[4] Office Excel. Microsoft Corporation. http://office.microsoft.com/en-us/excel/ (December 12, 2010)

Fig. 9 shows, for reasons of space, only a fragment of the visualization in Excel format of the two input models. In this case, they correspond to the models of the first case shown in Fig. 2. At the first level of indentation, they are all the elements of the models, which correspond to the grammar elements and the inferred metamodel of the DSL. They are identified by the hash code obtained by processing them with Java. Attributes and references of each element of the grammar are in the next level of indentation. These two elements can be easily differentiated because the attributes are assigned a specific value, while the references point to the hash code value of other elements of the model. Although Fig. 9 does not show all the information of the models, the structure corresponds exactly to the grammar shown in Fig. 1, which is also used by Xtext to generate the corresponding metamodel for the DSL.

Other interesting information is shown in Fig. 10 and is also automatically generated in the Excel spreadsheet. The first table is automatically inferred by MCTest from the optimal result implemented for each test case. The second table is automatically inferred from the obtained results provided by the algorithm of each test element of each test case. Finally, the third table is also automatically obtained, this time by comparing the first and second table.




**10**. Fragment of the output tables

A field marked as matched represents a relationship between the two models. For example, the optimal or expected output shows a relationship between the element of the first model that has a hash value of 12254719 and the element of the second model that has a hash value of 7962652. When the field is marked as unmatched (on the hash code of some element of one of the models), it means that that item does not appear in the other model. Thus, the element of the second model whose hash is 13218198 does not match any of the elements of the first model. In other words, it is a new element that has been added. The numerical values inside the fields represent the degree of similarity that the model comparison algorithm identifies between the elements of the

two models.

In Fig. 10, the table that provides more relevant information, in terms of the quality of each algorithm, is the last. Fields marked as OK represent successes of the algorithm being tested over the specified optimal result. Fields marked as exceeded represent incorrect statements of the algorithm being tested. Finally, fields marked as missed represent places where a correct mark should appear but does not, always compared with the optimal result.

### *4.5 Adaptability and extensibility of the feedback*

Although MCTest provides a wealth of information natively, the design of the tool and its open nature, allow other developers to modify or extend the feedback that is offered.

Fig. 11 shows the main elements used in the feedback subsystem. The *ITestResult* interface serves as the basis for creating feedback algorithms. The methods of which it is composed are:

- *addTestCase*. It adds a new test case to the feedback.
- *addModels*. It adds two independent models that are compared in a test case.
- *addMatchModel*. It adds a new correspondence model for two independent models that are compared in a test case.
- *getModel*. It returns one of the models that are compared in a given test case.
- *getOptimalMatchTable*. It returns an optimal correspondence table between two models. That is, it returns the exact relations between two models. The *IMatchTable* interface contains information such as the degree of similarity between two elements or the state of each of the table elements (e.g., matched, unmatched, etc.).
- *getObtainedMatchTables*. It returns a set of correspondence tables between two models by applying

different algorithms. It is possible to apply various algorithms or test elements on the same test case. The idea is that the results should be as close as possible to the optimal expected result.

- *getCorrectedMatchTables*. It returns a set of correspondence tables. In this case, the information provided is the difference between the optimal table and each of the correspondence tables obtained by applying different algorithms.
- *getObtainedMatchInfos*. It returns information obtained for each algorithm applied to each test case. The *TestElementMatchInfo* class contains information such as the size of the models, the number of successes in the comparison, the number of failures, or the time spent for each algorithm.
- *getOptimalMatchInfo*. It returns information obtained for each global test case. The *TestCaseMatchInfo* class currently does not provide any information but it exists to facilitate additions in the future.
- *getTestCases*. It returns the list of all test cases.

All the needed to modify the information provided as feedback is to create an implementation of the *ITestResult* interface which implements each of the methods as desired. Another option, potentially the simplest, is to use the *TestResult* reference implementation and extend it as desired.

Furthermore, the *IOutput* interface contains only two methods:

- *generate*. It is used to perform the processing that converts the programmatic output to another output created with other technologies (e.g., plain text, HTML, XML, etc.).
- *setConfig*. It is used to enter configuration information (e.g., the output patch for the files that can be generated during the conversion process).

Fig. 11 provides an excerpt of a class that extends the *IOutput* interface with the aim of providing the MCTest

output in Excel format. In this case, a *WritableWorkbook* object is used from the Java Excel API[5] library, a mature, and open source java API enabling developers to read, write, and modify Excel spreadsheets dynamically.

```
public interface ITestResult {
    void addMatchModel(String testCase, IMatchModel matchModel,
            TestCaseMatchInfo testCaseMatchInfo);
    void addMatchModel(String testCase, IMatchModel matchModel,
            TestElementMatchInfo testElementMatchInfo);
    void addModels(String testCase, EObject model1, EObject model2);
    void addTestCase(String testCase);
    EObject getModel(String testCase, ModelSide modelSide);
    List<IMatchTable> getCorrectedMatchTables(String testCase);
    List<TestElementMatchInfo> getObtainedMatchInfos(String testCase);
    List<IMatchTable> getObtainedMatchTables(String testCase);
    TestCaseMatchInfo getOptimalMatchInfo(String testCase);
    IMatchTable getOptimalMatchTable(String testCase);
    List<String> getTestCases();
}
```

```
public interface IOutput {
    Object generate(ITestResult testResult);
    void setConfig(Map<String, Object> config);
}
```

```
public class ExcelOutput implements IOutput {
    private Map<String, Object> config;
    private WritableWorkbook workbook;
    private int row;

    @Override
    public Object generate(ITestResult testResult) {
        this.createWorkbook();
        this.createSheets(testResult);
        this.closeWorkbook();
        return null;
    }
    ...
```

**11**. **Basic elements for the feedback subsystem**

# 5 Evaluation

To show how the MCTest tool works in the case study of this paper, we performed the following global tests:

- Seven test cases. We created a test case for each of the cases shown in Fig. 2. We assigned the optimal

[5] Java Excel API. http://jexcelapi.sourceforge.net/ (August 5, 2011)

expected result to each test case to be compared with the results of each test element.

- Four test elements. We created test elements based on the generic match algorithm provided by the EMF Compare tool. Each test element has been configured by changing some parameters of the generic algorithm to be compared on the basis of a common reference. The 4 test elements have been assigned to each of the test cases, obtaining a total of 28 tests without requiring further configuration.

## 5.1 Adaptability of EMF Compare

The EMF Compare tool is highly configurable. For example, it allows new heuristics to be added or existing ones to be deleted. It also allows them to be modified, which could lead to very different matches being detected between elements.

The basic heuristics are run by classes extending the abstract class called *AbstractSimilarityChecker*. The main heuristic is *StatisticBasedSimilarityChecker*, which performs comparisons using statistical calculations, and which is the option used by many tools that make generic and independent of the metamodel comparisons, treating models as graphs. *DistinctEcoreSimilarityChecker* is a specialization of *StatisticBasedSimilarityChecker*, designed to work with several metamodels at the same time. Both *StatisticBasedSimilarityChecker* and *DistinctEcoreSimilarityChecker* can be decorated by other classes, which make them better adapted to certain conditions. Thus, *XMIIDSimilarityChecker* and *EcoreIDSimilarityChecker* serve to perform comparisons using an identifier attribute when available, greatly facilitating the process in a way equivalent to that done in works like Alanen and Porres [11].

Checkers are responsible for deciding whether an element does or does not match another element and to what degree they are equivalent. Regardless of whether the basic checker is *StatisticBasedSimilarityChecker* or *DistinctEcoreSimilarityChecker*, there are a number of parameters that are hard coded into the source code but

that could be modified. Some of these include the following:

- The general threshold used to determine whether an element of a model is similar to another (by default 0.96). All values range from 0 (totally different) to 1 (identical) and are primarily determined by the results of the heuristics.
- The minimum number of structural elements that an element must have to make the comparison of its elements with respect to others, by using a criterion based on the content of the elements (by default, 5).
- The threshold for which one element is considered almost equal to another (by default, 0.999999).
- Other thresholds used to check whether two objects are similar or not (e.g., based on the content, the relationships, the number of attributes).
- The weights of the comparison criteria based on the content, the name, or the position of the elements.

In addition to adding, removing, or changing heuristics, we could even change the entire match algorithm, which by default is called *GenericMatchEngine*. To do this, the interface *IMatchEngine* should be implemented. Then, it would be necessary to create an algorithm that makes use of the data structures, techniques, filters, or heuristics that are deemed appropriate.

All this opens the door to a huge number of combinations that could be studied, for example, to improve the success rate of the match algorithm, to reduce their processing times, to do empirical studies of certain algorithms, or to try to justify the choice of one parameter value instead of another.

Given the many combinations that are possible with EMF Compare, the other tools that exist or may exist in the future to compare models, the semantics of each language, and the possible evolution of the structures used to define models and / or metamodels, we think that MCTest can facilitate the implementation of comparisons.

Moreover, it allows results to be obtained automatically, which can be viewed, understood, stored, and processed immediately.

## *5.2 Test elements*

The first test element encapsulates the algorithm with the default settings and heuristics that EMF Compare includes. Initially, it uses *XMIIDSimilarityChecker* to decorate *EcoreIDSimilarityChecker*, which in turn decorates *DistinctEcoreSimilarityChecker*. We have not changed anything about the algorithm, so the result would be the same as that obtained by any other person, doing the same tests with EMF Compare.

The second test element encapsulates a custom heuristic (we called it *NoSimilarSimilarityChecker*). The idea is that when the algorithm asks the heuristic whether two elements are similar, the heuristic must respond no, and that in fact they have absolutely nothing to do with each other.

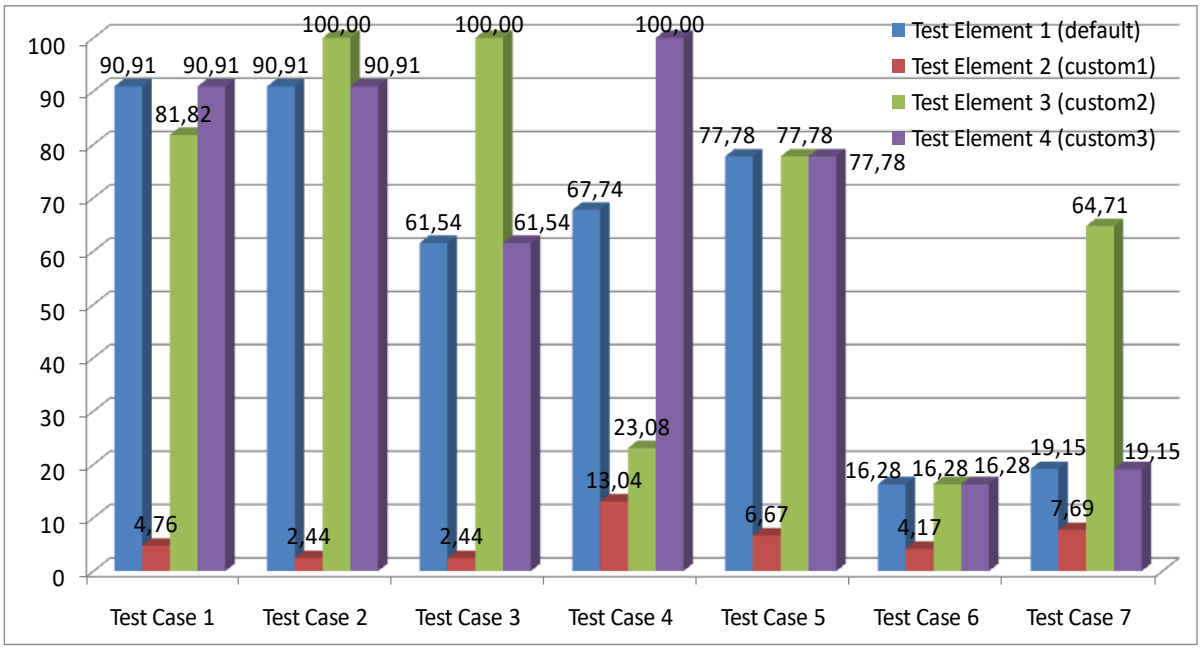
The third test element also encapsulates a custom heuristic (we called it *SimilarSimilarityChecker*). The idea is the opposite of the previous heuristic. In this case, when the algorithm asks the heuristic whether two elements are similar, the heuristic must respond yes, and that in fact they are exactly the same element. Thus, both the second and the third test elements can be considered two pseudorandom heuristics, which should also be executed with great speed.

Finally, the fourth test element, like the first test element, encapsulates the default algorithm of EMF Compare. However, at runtime, it is possible to perform some basic configurations using a data structure containing a *Map* of pairs <*String*, *Object*>. In this case, we have configured it to use *StatisticBasedSimilarityChecker* instead of

*DistinctEcoreSimilarityChecker*. This checker is specially designed to work with the same metamodel for each case.

## *5.3 Results*

We have evaluated the default algorithm provided by EMF Compare, and the 3 modifications on it (4 test elements in total) through the use of the MCTest tool to show the results. Fig. 12 and 13 are extracted from the output of MCTest with such algorithms but the tool is independent of the underlying technology and thus it could be used to evaluate other algorithms. Fig. 12 shows the results obtained for the 28 tests done using MCTest (7 test cases with 4 test elements), focusing on the success rate of the algorithms. Clearly, the second algorithm is the one that is least effective. However, it is not possible to clearly indicate the better of the other three algorithms, as their relative performance depends on the case.

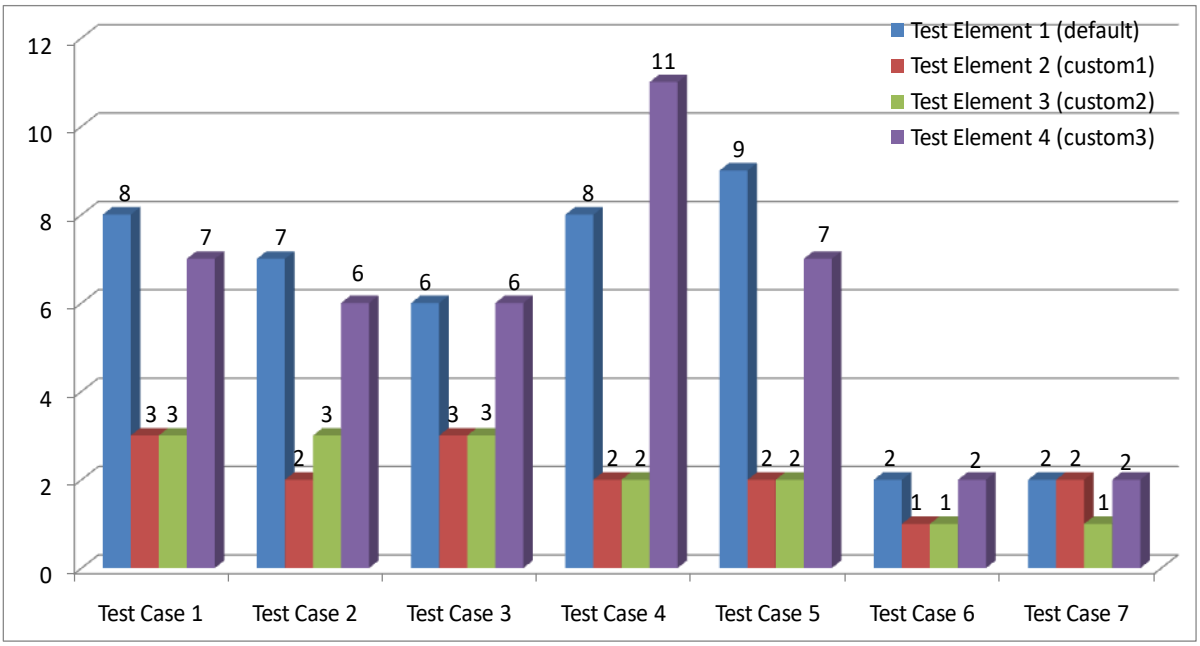



**12. Success rate (%)**

Fig. 13 shows the results obtained for the 28 tests done using MCTest in terms of the time spent by the

algorithms to perform the necessary calculations. Surprisingly, the third and pseudo-random algorithm, which is much faster than the default algorithm incorporated by EMF Compare, can rival its percentage of success in several test cases. This leads us to believe that the generic algorithm of EMF Compare could be improved in a future work.

6




7 **13**. Time spent (ms)

# 8 Conclusions and future work

The current tools used to compare models can be improved. This opens the door to new research to find better algorithms for calculating the matches and differences between models. MCTest allows the testing of algorithms for determining the relationships between models, by measuring parameters, such as the successes, failures, or time spent by the algorithm. Thus, MCTest can help to increase the quality of the available approaches for comparing models. In fact, using MCTest, it was very easy to test different algorithms with different test cases,

showing results in an easily understandable interface for which we will carry out usability tests to find weaknesses and to improve the ease of use for next releases of the tool.

A crucial future work will entail providing a plug-in to integrate the tool into the Eclipse Platform, thus facilitating its use through a graphical interface rather than programmatically.

In terms of the scope of the MCTest tool, the next step will be to provide information about the Diff operator to complement the output from the Match operator, and thus to provide much more information to the creators of algorithms for comparing models.

Moreover, we will also provide as output (by default, in an Excel spreadsheet) information about computational resources used by each algorithm, an aspect that may be important for evaluating implementations. In addition, we will provide a deeper comparison between algorithms by using specific graphics.

The so-called 3-Way differences [24] were not taken into account in this study because the comparison using a third document, the predecessor of the other the, is more oriented to the existence of different branches in the history of a model maintained by a version control system. However, it could also be an interesting extension in a future work.

Focusing on our case study, the next step will be to implement the *IMatchEngine* interface provided by EMF Compare with the aim of obtaining an algorithm to solve the seven problems of our case study in which the genetic algorithm has not worked properly. Furthermore, the objective is that the algorithm does not lose accuracy or efficiency compared to the performance offered by default by EMF Compare for other generic cases. The computational complexity of the new algorithm will be measured in further research because it is out of the scope of this paper. Moreover, we have left open the possibility of testing other research algorithms different from those provided by EMF Compare for a further research, as our tool is fully independent of the technology

and the focus of this work is the tool for studying algorithms, not to create the best algorithm.

Finally, we will also release the source code of MCTest (https://sourceforge.net/projects/mctest/) to the open source community to encourage the development and improvement of the tool.

## Acknowledgments

This work has been partially funded by the Government of the Principality of Asturias (Regional Ministry of Education and Science)

## 9 References

[1] Kent S. Model Driven Engineering. In: IFM ’02: Proceedings of the Third International Conference on Integrated Formal Methods. London, UK: Springer-Verlag; 2002. p. 286–298.

[2] Bézivin J. On the Unification Power of Models. In: Software and System Modeling. vol. 4; 2005. http://www.sciences.univ-nantes.fr/lina/atl/www/papers/OnTheUnificationPowerOfModels.pdf.

[3] Frankel DS. Model Driven Architecture: Applying MDA to Enterprise Computing. John Wiley & Sons; 2003.

[4] Miller J, Mukerji J, Belaunde M, Burt C, Cummins F, Dsouza D, et al. MDA Guide, v1.0.1. Object Management Group; 2003. http://www.omg.org/docs/omg/03-06-01.pdf.

[5] Nejati S, Sabetzadeh M, Chechik M, Easterbrook S, Zave P. Matching and Merging of Statecharts Specifications. In: ICSE ’07: Proceedings of the 29th international conference on Software Engineering. Washington, DC, USA: IEEE Computer Society; 2007. p. 54–64.

[6] Oliveira HLR, Murta LGP, Werner C. Odyssey-VCS: a flexible version control system for UML model elements. In: SCM. ACM; 2005. p. 1–16.

[7] Lin Y, Zhang J, Gray J. Model comparison: A key challenge for transformation testing and version control in model driven software development. In: Control in Model Driven Software Development. OOPSLA/GPCE: Best Practices for Model-Driven Software Development; 2004. .

[8] Xing Z, Stroulia E. UMLDiff: an algorithm for object-oriented design differencing. In: ASE ’05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering. New York, NY, USA: ACM; 2005. p. 54–65.

[9] Mandelin D, Kimelman D, Yellin D. A Bayesian approach to diagram matching with application to architectural models. In: ICSE ’06: Proceedings of the 28th international conference on Software engineering. New York, NY, USA: ACM; 2006. p. 222–231.

[10] Maiden N, Sutcliffe A. Exploiting reusable specifications through analogy. Commun ACM. 1992;35(4):55–64.

[11] Alanen M, Porres I. Difference and Union of Models. In: Proceedings of UML 2003; 2003. p. 2–17.

[12] Brun C, Pierantonio A. Model Differences in the Eclipse Modeling Framework. UPGRADE, The European Journal for the Informatics Professional. 2008 April;9(2):29–34.

[13] Kolovos DS, Di Ruscio D, Pierantonio A, Paige RF. Different models for model matching: An analysis of approaches to support model differencing. In: CVSM ’09: Proceedings of the 2009 ICSE Workshop on Comparison and Versioning of Software Models. Washington, DC, USA: IEEE Computer Society; 2009. p. 1–6.

[14] Hunt JW, Mcllroy MD. An algorithm for differential file comparison. Bell Telephone Laboratories; 1977. Computing Science Technical Report No.41.

[15] Hirschberg DS. Algorithms for the Longest Common Subsequence Problem. J ACM. 1977;24(4):664–675.

[16] Myers EW. An O(ND) Difference Algorithm and Its Variations. Algorithmica. 1986;1:251–266.

[17] Farail P, Gaufillet P, Canals A, Le Camus C, Sciamma D, Michel P, et al. the TOPCASED project: a Toolkit in OPen source for Critical Aeronautic SystEms Design. In: Embedded Real Time Software (ERTS). Toulouse; 2006. .

[18] Reddy R, France R. Model Composition - A Signature-Based Approach. In: in "Aspect Oriented Modeling (AOM) Workshop, Montego; 2005. .

[19] Tai KC. The Tree-to-Tree Correction Problem. J ACM. 1979;26(3):422–433.

[20] Cobena G, Abiteboul S, Marian A. Detecting Changes in XML Documents. In: In ICDE; 2001. p. 41–52.

[21] Zhang K, Shasha D. Simple fast algorithms for the editing distance between trees and related problems. SIAM J Comput. 1989;18(6):1245–1262.

[22] Wang W. Evaluation of UML Model Transformation Tools. Business Informatics Group, Vienna University of Technology; 2005.

[23] Khuller S, Raghavachari B. Graph and network algorithms. ACM Comput Surv. 1996;28(1):43–45.

[24] Ohst D, Welle M, Kelter U. Differences between versions of UML diagrams. In: ESEC/FSE. vol. 28. New York, NY, USA: ACM; 2003. p. 227–236.

[25] Selonen P. A Review of UML Model Comparison Approaches. In: Proceedings of Nordic Workshop on Model Driven Engineering; 2007. .

[26] OMG. Ontology Definition Metamodel. Object Management Group; 2005. http://www.omg.org/docs/ad/05-08-01.pdf.

[27] Lin Y, Gray J, Jouault F. DSMDiff: A Differentiation Tool for Domain-Specific Models. European Journal of Information Systems,. 2007;16:349–361.

[28] Ledeczi A, Maroti M, Bakay A, Karsai G, Garrett J, Thomason C, et al. The Generic Modeling

Environment. In: Workshop on Intelligent Signal Processing, Budapest, Hungary. vol. 17; 2001. .

[29] Melnik S, Garcia-molina H, Rahm E. Similarity flooding: A versatile graph matching algorithm. In: 18th International Conference on Data Engineering; 2002. p. 117–128.

[30] Selonen P, Kettunen M. Metamodel-Based Inference of Inter-Model Correspondence. In: CSMR ’07: Proceedings of the 11th European Conference on Software Maintenance and Reengineering. Washington, DC, USA: IEEE Computer Society; 2007. p. 71–80.

[31] Treude C, Berlik S, Wenzel S, Kelter U. Difference computation of large models. In: ESEC-FSE ’07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering. New York, NY, USA: ACM; 2007. p. 295–304.

[32] Rivera JE, Vallecillo A. Representing and Operating with Model Differences. In: TOOLS EUROPE 2008. vol. 11 of LNBIP. Springer; 2008. p. 141–160. http://dx.doi.org/10.1007/978-3-540-69824-1_9.

[33] Gronback RC. Eclipse Modeling Project: A Domain-specific Language Toolkit: A Domain-Specific Language (DSL) Toolkit. 1st ed. Addison-Wesley Educational Publishers Inc; 2009.

[34] Gronback RC. Eclipse Modeling Project and OMG Standard. In: Eclipse Modeling Symposium; 2006. .

[35] OMG. Meta Object Facility 2.0. OMG; 2005. http://www.omg.org/spec/MOF/2.0/PDF/.

[36] Gruschko B, Kolovos DS, Paige RF. Towards synchronizing models with evolving metamodels. In: In Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR; 2007. .

[37] Steinberg D, Budinsky F, Paternostro M, Merks E. EMF: Eclipse Modeling Framework 2.0. Addison-Wesley Professional; 2009.

[38] Toulmé A. Presentation of EMF Compare Utility. In: EclipseCon; 2007. .

[39] Efftinge S, Völter M. oAW xText: A framework for textual DSLs. openArchitectureWare; 2006.

[40] Howe DC. A WordNet library for Java/Processing; 2011. http://www.rednoise.org/rita/wordnet/.

[41] Fellbaum C, editor. WordNet. An Electronic Lexical Database. Cambridge, MA ; London: The MIT Press; 1998.

[42] Beck K. Simple Smalltalk Testing: With Patterns. The Smalltalk Report. 1994;4(2):16–18.

[43] Kung DC, Gao J, Toyoshima PH, Chen C. On regression testing of object-oriented programs. Journal of Sys. 1996;32(1):21–40.

[44] Gamma E, Helm R, Johnson R, Vlissides J. Design patterns: elements of reusable object-oriented software. Addison-Wesley; 1995.