EMFCompare matching testsuite

What is the *de.fzi.emfcompare.matching.testsuite* ?

The EMFCompare testsuite is an Eclipse plug-in based framework to support the testing of custom EMFCompare MatchEngines implementations. With a predefined layout and a custom domain specific language (DSL) to easily describe the expected output of your matcher even for complex models, it aims at making the testing process as convenient and structured as possible.

What is it good for?

When developing a custom MatchEngine for EMFCompare you might want to test the results of your implementation separately from the EMF comparison result to ensure a stable and reliable matching even for complex or large model instances. Instead of comparing each matching result of EMFCompare by hand or having to manually define and traverse an unhandy *org.eclipse.emf.compare* model, this testsuite allows you to specify expected matches for any two model instances recursively with a DSL and provides tools to automatically verify your MatchEngine implementation.

Contents of this document

Structure

de.fzi.emfcompare.matching.matches.feature

Feature that contains all necessary packages needed by Eclipse for the Matches DSL and the Xtext UI to describe expected EMFCopare matches.

de.fzi.emfcompare.matching.matches.*

These projects contain the Xtext source code and generated artifacts for the Matches DSL and the UI components.

de.fzi.emfcompare.matching.testsuite.feature

Contains all packages necessary for the complete EMFCompare matching test suite. This feature does only include the de.fzi.emfcompare.matching.matches project and does therefore not contain the Eclipse editor for the Matches DSL.

de.fzi.emfcompare.matching.testsuite

de.fzi.emfcompare.matching.caex30.tests

This project contains an examplary implementation of the testsuite for the CAEX and Recipients metamodels using both methods of specifying expected matches: the Matches DSL and EMFCompare models

Usage

Matches Language

To specify expected Matches with the Matches domain specific language and the Eclipse editor the **EMFCompare Matches Language Feature** [de.fzi.emfcompare.matching.matches.feature] needs to be installed in the Eclipse Instance.
Then files with the file ending '.matches' can be edited using the Matches Editor.
[Jump directly to the complete syntax]

Specifying Matches

Matches can be specified as single Matches:

```
// Always: Match [left model element] with [rightmodel element]
Match "path from left root" with "path from right root"
```

or as recursive Matches (Tree Matches), thus meaning not only shall the two named elements be matched but also each of their children. By default the Matches Language then matches the first child of the left element with the first child of the right element, the second child on the left with the second child on the right and so on ..

```
// Always: Match Tree [left model element] with [rightmodel element]
Match Tree "path from left root" with "path from right root"
```

[More complex Tree Matches are discussed after the next section]

Identifying Elements

The Elements that are to be matched can be addressed four different ways:

Path

Paths are declared using double quotes and allow to navigate freely through the model, addressing all elements that have at least one locally unique feature/value pair. The default delimiter is a single slash **'/'** and the default feature that is used for navigation is **'name'**. Commonly known path components such as **'../'** *(Move one level up)* or **'./'** *(stay at the same element)* can also be used:

```
// Some exemplary path declarations
"/path/from/root"
"./relative/path"
"another/relative/path"
"../relative/path/after/moving/one/level/up"
"unnecessary/repititive/../repititive/../repititive/path"
```

For navigating by different feature names than **'name'** the respective path segment is defined as follows: *'@feature_name=feature_value'*. Valid paths could look like this:

```
// Some exemplary path declarations
"path/@id=42/element1/@value=14"
"./@id=some/../@idd=valid/@id=path"
```

ID

If the Element has a globally unique ID, it can also be addressed as follows: *'uuid=uid_of_Element'*.

```
Match uuid="uuidOfTheLeftElement" with "path from right root"
```

Position

Another way of addressing model elements is by stating their position in the parent Container. This means any element is addressable them with a positive integer $\in\mathbb{Z}^+\cup {0}$ from the context of their parent.

```
Match 0 with "path/from/right/root"
Match 1 with 2
Match 2 with "./relative/path"
```

Nothing

When a model element is created, deleted or missing there is no other element to match it with. For these cases the Matching language reserves the keyword **nothing** which can replace any Path, ID or position, such as:

```
Match nothing with "path/from/right/root"
Match Tree "./relative/path" with nothing
Match nothing with nothing
```

Specifying complex Tree Matches

When matching two trees of elements with another, some special cases have to be addressed:

- One element possesses more child elements than the other:
  In this case the spare child elements are matched with *nothing*
- What if a child element contains more elements?
  For every two child elements (one from the left and one from the right model) of a Tree Match the Testsuite actually creates a new Tree Match during execution and therefore also matches child elements of childs recursively.

Trees often are very similar except for a few elements, that may have changed order, got deleted or created or even got moved to another sub-tree. For those instances the Matches language provides an advanced construct to match just slightly different model trees.

Say for example two trees are exactly the same, except the first element got deletet and the second element was swapped with the third. Instead of having to formulate each match separately the Matches DSL allows you to write it as this:

```
Match Tree "left" with "right" except {
    0 with nothing
    1 with 2
    2 with 1
}
```

Remaining child elements are matched as if all mentioned elements above never existed, meaning *3 with 2*, *4 with 3* and so on .. This means that line 3 or 4 could even be dropped from the code and the snipped would still specify the exact same matches:

```
Match Tree "left" with "right" except {
    0 with nothing
    1 with 2
}
```

Instead of position information, elements can also be addressed by absolute/relative paths and id's. This means if an element got moved into a sub-tree or even out of a tree, the specification would look something like this:

```
Match Tree "left" with "right" except {
    0 with "/movedElement"
    1 with "subTree/element"
}
```

Additionaly it is possible to nest **except** Statements, to comfortably describe nearly any possible matches:

```
Match Tree "left" with "right" except {
    0 with "/movedElement" except {
        0 with nothing except {
            3 with "../someElem"
        }
    }
    1 with "subTree/element"
}
```

Specifying custom Delimiters and Feature Names

The default path delimiter and the default feature name that is accessed if a path segment doesn't specify a concrete feature can be configured by adding the following block at the beginning of the Matches snippet:

```
using {
delimiter:  "\\"
defaultIdentifier:  "id"
}
```

Complete Syntax

The complete syntax for any file of the Matches Language looks as follows:

```
Compare
left: "relative or absolute path to the left/before model file" //optional
right: "relative or absolute path to the right/after model file" //optional

using {                     // this block is also optional
delimiter: "/"              // default delimiter: '/'
defaultIdentifier: "name"   // default Identifier: 'name'
}

Match "Element Left" with "@id=Element_Right"

Match Tree "Path" with "Path"

Match Tree "Path" with "Path" except {
    0 with 1
    "relative/Path" with "/root/Path"
}
```

Implementation Examples

For concrete exemplary instances and models see the *de.fzi.emfcompare.matching.caex30.tests* project.

Simple Example #1

```
left
|
+-- element1
+-- element2
|  +-- element2_1

right
|
+-- element2
+-- element2_1
```

```
Compare
Match Tree "left" with "right" except {
    0 with nothing
    1 with 0
    "element2/element2_1" with 1
}
```

or

```
Compare
Match Tree "left" with "right" except {
    0 with nothing
    1 with 0 except {
        0 with "../element2_1"
    }
}
```

Test Framework

Structure of implementing tests

The general idea behind the framework is to detect tests automatically in the executing project, and therefore not having to maintain a list of availiable tests. Because of this a test-project is optimaly structured as this:

```
test-project
|
+-- beforeModels // Contains multiple 'before' models
|  +-- model1.model
|
+-- afterModels // Contains multiple 'after' models
|  +-- model1.model
|
+-- expectedMatches     // Contains a description of the matches
|   +-- model1.matches  // expected by EMFCompare either in the
|                       // .compare, .matches or any other format
+-- src         // Test code ...
```

Naming corresponding models and matches resources with the exact same file name also provides the benefit of a shorter .matches header with:

```
Compare
```

vs

```
Compare
left: "beforeModels/model1.model"
right: "afterModels/model1.model"
```

Using the Framework

The key components are the implementations:

- **AbstractMatchingTestBuilder** - Combines multiple implementations of the *AbstractMatchingTestsSuite* into an executable JUnit Test
- **AbstractMatchingTestSuite** - Allows the configuration of Locations for the models and matches files and their extensions, as well as a custom *AbstractMatchingTestCase* implementation. During execution the TestSuite executes the specified TestCase for each matching before_model, after_model and matches_resource triplet.
    - **DSLMatchingTestSuite** - A specific implementation of the *AbstractMatchingTestSuite* that by default scans the folders '/before', '/after', '/expectedMatches' and uses the Matches DSL for matches specification and the

DSLMatchingTestCase. (Is further customizable)

- **AbstractMatchingTestCase** - Implementations of this class are given a triplet of before_model, after_model and matches_resource and have to assert that the matches_resource equals the result of an EMFCompare execution.
    - **DSLMatchingTestCase** - A specific implementation for any model types and matches descriptons that use the Matches DSL. (Is further customizable, e.g. the IMatchEngine)
- **MatchingTestsFactory** - A utility factory that allows some configurations to the classes presented above, without having to implement them.

Using the above introduced components a matching test can be created by creating a static 'suite()' method that returns the result of AbstractMatchingTestBuilder.buildSuite().

For example:

```
public class RunAllTests {
    /**
     * The <b>static</b> suite method is picked up by JUnit
     * and executed as a TestSuite
     */
    public static void Test suite() {
        MatchingTestsFactory factory = new MatchingTestsFactory();
        AbstractMatchingTestBuilder suiteBuilder = factory
        .createMatchingTestBuilder("de.fzi.emfcompare.matching.caex30.tests");

        suiteBuilder.addMatcherTest(
            factory.factory.createDSLMatchingTestSuite(
                "CAEXMatchingTestSuite",
                "caex",
                new MatchEngineFactoryImpl().matchEngine)
        );

        return suiteBuilder.buildSuite();
    }
```

See '*de.fzi.emfcompare.matching.caex30.tests/ src/ de.fzi.emfcompare.matching.caex30.tests/RunAllTests.xtend*' for a reference implementation.