Extending JUnit 4 with Java Annotations and Reflection to Test Variant Model Transformation Assets

Fábio Paulo Basso, Toacy Cavalcante Oliveira
COPPE, Universidade Federal do Rio de Janeiro (UFRJ),
Rio de Janeiro, RJ, Brazil
{fabiopbasso,toacy}@cos.ufrj.br

Kleinner Farias
PIPCA, Universidade do Vale do Rio dos Sinos,
São Leopoldo, RS, Brazil
kleinnerfarias@unisinos.br

ABSTRACT
Software Product Line (SPL) techniques are widely used to represent variability and commonality in reusable software assets. Similarly, model transformations are also software assets and can be reused with the same techniques. However, their applicability in the model transformations domain demands an extra effort to test the generated/adapted assets. Automated test cases should consider isolated transformations and also their combined use in a model transformation chain, that can vary according to different needs in software projects, e.g. libraries and frameworks. In order to facilitate the specification of automated test cases, this paper presents a JUnit extension to support unit and integration tests that execute dynamic SPL-based model transformation chains.

Categories and Subject Descriptors
D.2.2 [**Software Engineering**]: Object-oriented design methods, Model Driven Development (MDD), Context specific languages - API languages.

General Terms
Design, Theory.

Keywords
Model transformation chain, Software Product Lines, MDE, Java annotations, Java reflection, Unit tests, Integration tests.

1. INTRODUCTION
Model Driven Engineering (MDE) [27] is a software development strategy where models are used to generate code or other models [25]. In real world scenarios, where models can become complex and diverse, the whole transformation strategy is typically broken down into several transformation algorithms that are combined in a Model Transformation Chain (MTC) [36]. According to Baudry et al., researchers and practitiones should exhaustively test MTCs because they are complex and error-prone tasks [7]. In this sense, testing transformation assets is a critical task [21], since transformations are frequently changed to support increments [20].

Hervieu et al. [14] and Perrouin et al. [30] claim that it is even more difficult to specifying automated tests for variant transformation assets (e.g. model transformations adapted for software projects using SPL), since it is harder to construct test logic than it is to construct those for a regular software line [22]. For example, a domain model composed of variant transformation assets allows the execution of dynamic MTCs, requiring unit and integration tests. Therefore, it is necessary a solution to facilitate the specification of automated test cases considering variant transformations.

In this sense, Offutt et al. [28] and McGregor et al. [23] introduce some techniques to test SPLs. In addition, Reuys et al. [31] present a top-down solution for automated test case generation, taking as input a model and generating specific test cases according variabilities in a domain model. Using a bottom-up solution, proposals [33][19] apply reverse-engineering techniques in existing software assets to extract relevant test cases to test SPLs. Currently, some works are moving towards agile testing [13], considering Test Driven Development (TDD) and SPL [17][22][26][24]. In this sense, bottom-up solutions are more adequate to apply TDD than top-down is, because variabilities are identified and extracted while the SPL is being constructed. Bottom-up is also recommended to develop variant model transformations that are incrementally constructed with MTCs [20].

Along the development of test cases, we have found out that existing JUnit 4 API [16], used to specify and execute unit tests in Java, lacks in functionalities to apply TDD in variant model transformations. In order to provide a specific solution, this paper presents an extended JUnit API to automate test cases for variant transformations, with similar performance to the regular API. We introduce this solution into a model transformation engine, namely WCT [5]. The validation of this work is a set of automated test cases that deals with a complex and real scenario in adapting large scale transformations, presenting some results in comparison with previous experiences in [4].

This paper is organized as follows. Section 2 presents a motivating example. Section 3 introduces the proposed automated test API.. Section 4 evaluates the proposed extension and describes the ongoing works. Section 5 presents drawbacks and limitations of the proposed extension. Section 6 present the related works. Finally, Section 7 shows the conclusions.

2. MOTIVATING EXAMPLE
This section presents a practical scenario extracted from industrial projects using large-scale model transformations. This scenario will help demonstrate the issues related to the practice of testing model transformation assets. This experience reported by example

SAC'14 March 24-28, 2014, Gyeongju, Korea.

http://dx.doi.org/10.1145/2554850.2555054

is relevant for studies related to reuse techniques, applied in model transformations in following works [11][14][32][37][2].

FOMDA (Features-Oriented Model-Driven Architecture) is a methodology to specify generative and dynamic model transformation chains [4]. In order to support these specifications, WCT is a tool that allows designing and also executing transformations based on three kinds of assets: Feature Model [18][10] (Figure 1 A); Model Transformations (Figure 1 B); and Model Transformation Chains (Figure 1 C).

A platform domain model (PDM) [35] is represented a Features Model (illustrated in Figure 1 (A) and Figure 2) that exposes the system's characteristics. In addition, each feature found in the PDM can be related to a set of Model Transformers that will eventually generate code (or other models) that represents such feature with more details. Another important aspect of our approach is combing Transformers into a Model Transformation Chain. This step allows defining the sequence in which Model Transformers are executed to generate a given product from the Product Line (Figure 1 (C) and Figure 3 (A)).

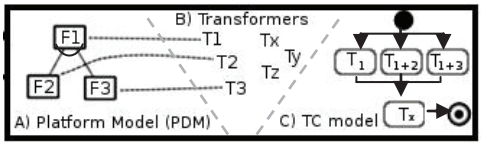


Figure 1. Domain Models Used in FOMDA Methodology

In order to exemplify transformations based on real scenario, reported in an industrial effort in tailoring transformation assets in [5], next section exemplifies some points that varied in the generation of source-code for Object Relational Mappings (ORM) in support for development of information systems. The relationship below *"ORM"* is a mandatory XOR. This means that it is necessary to select one and only one feature among *"JPA"*, *"XDoclet"*, *"JDO"* and so on. These features support the specification of ORM into entity classes, illustrated in Figure 4 (B). Examples of source-code for JPA and XDoclet are presented in Figure 4 (A and C). Such examples can be generated with the support of two different model transformations, exemplified in [5] using model-to-code generations. In contribution, next section exemplifies the use of variant transformations.

2.1 Transformation Chain Domain Model

Figure 3 shows an example of variant transformations. This example prestens a diagram with transformations that generates the model layer of an information system, i.e. a Transformation Chain Domain Model (TCDM). Figure 4 (B) illustrates an element of a model layer, which is used as input for transformations. The TCDM owns an abstraction (dotted element), used to generalize transformations that perform ORM transformations for different target implementation technologies shown in the PDM. Abstractions groups mutually exclusive transformations, represented by the relationship stereotyped with «requires», between transformations and features, and with the stereotype «XOR», owned by the abstraction. Considering this example, the abstraction *"ORM to PSM"* is replaced, in runtime or by generating a concrete MTC, by: 1) *"Generate JPA"*, whose result from a transformation is shown in Figure 4 (A), or; 2) by *"Generate XDoclet"*, whose result of a transformation is shown in Figure 4 (C).

2.2 Platform Domain Model

An example of a PDM is shown in Figure 2 as a feature model where the filled circle means the feature is mandatory and, otherwise, it means feature is optional. Therefore, *"Remote Layer"* is an optional feature and *"Model Layer"* is mandatory.

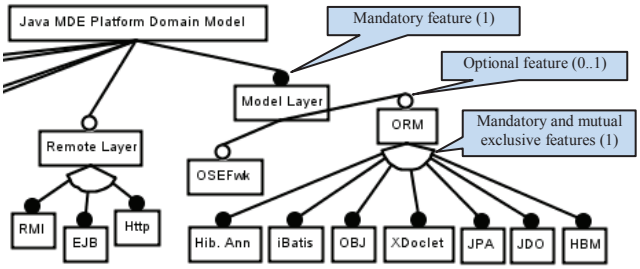
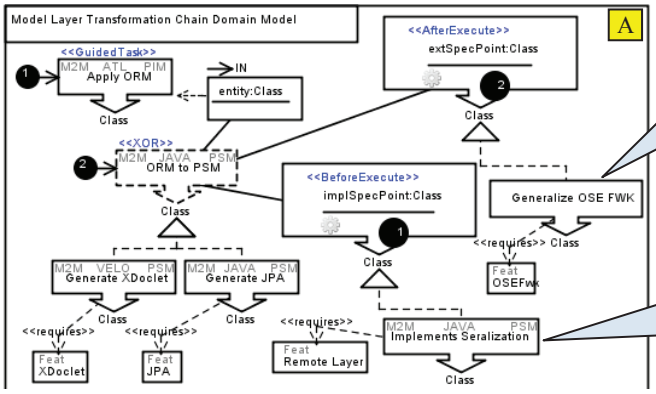


Figure 2. Screenshot of a Platform Domain Model (PDM)

The *"Remote Layer"* variant is used to integrate a subsystem, that runs in desktop or mobile platform, with the application logic, that runs on a web server. In this case, when generating source-code in Figure 4 (A) or (C), the class named *Person* must implement the *Serializable* Java interface, as shown in Figure 4 (D). With this in mind, when the feature *"OSEFwk"* is selected, then the class *Person* must extend the class *com.osefwk.Entity*, as exemplified in Figure 4 (E). In this sense, the transformation named *"Generate OSE FWK"* is executed or generated into an MTC.

With support of WCT, one can fragment a transformation into independent modules that are used in runtime or combined through generative techniques discussed in [4][5]. These elements are also abstractions illustrated in Figure 3 (A) with model elements named *"implSpecPoint"* and *"extSpecPoint"*. The execution of transformation showed in Figure 3 (B) results in Figure 4 (E) in case the feature *"OSEFwk"* is selected. Figure 3 (C) illustrates another model-to-model transformation that results into




```
B
public class TransformToOseFwk extends AbstractTransformer{
1  public Object doTransformation() {
2     Class entity = (Class) getParameter("extSpecPoint");
3     if (entity.containsStereotype("Entity")) {
4        entity.addGeneralizationFrom("com.osefwk.Entity");
      }
5     return entity;
```

```
C
public class TransformToSerializable extends AbstractTransformer{
1  public Object doTransformation() {
2     Class entity = (Class) getParameter("implSpecPoint");
3     if (entity.containsStereotype("Entity")) {
4        entity.addRealizationFrom("java.lang.Serializable");
      }
5     return entity;
```

Figure 3. Screenshot of a Transformation Chain Domain Model

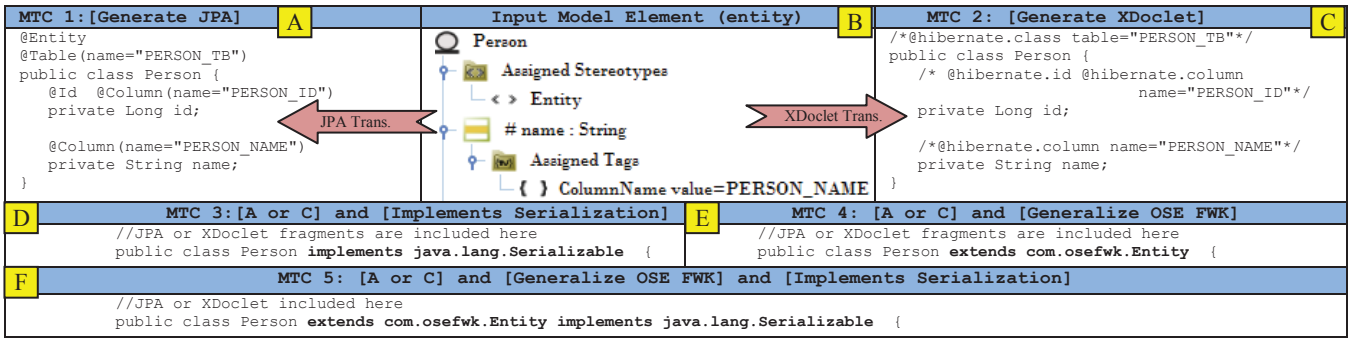


Figure 4. Exemplification of the Results from Different Model Transformation Chains Obtained in TCDM shown in Figure 3

the source-code shown in Figure 4 (D) in case the feature *"Remote Layer"* is selected. If both features are selected, then both model transformations are executed or included in a generated MTC, generating the code illustrated in Figure 4 (F).

The building of application generators considering variants requires the writing of unit and integration test cases for each possible combination among features and transformations [7]. This leads to a complex scenario to test dynamic software product lines, requiring a procedure depicted by McGregor [23]. Although a procedure is important to write test cases, this paper does not attempt to procedures, but to an extension for a JUnit required to perform variant test cases, as discussed in the next section.

3. PROPOSED AUTOMATED TEST API

In order to allow the application of TDD in variant model transformation assets, JUnit was extended through the class *junit.framework.TestCase*, inherited by *FomdaTestCase* to override *setUp* and *tearDown* operations and adding new operations. It was also extended with custom Java annotations and Java reflection. Java annotation is the extension mechanism provided since version 5, allowing the addition of compiled extra-information to classes, attributes, operations, and parameters. We are using annotations to specify information commonly used by test cases to test transformation units. Moreover, through Java reflection we are also injecting dependencies for variant model transformations, making test cases also variant and ruled by a domain context.

The program shown in Figure 6 is a derived JUnit test case to test variant transformations. Lines 1 and 2 show a new Java annotation developed to handle common test tasks applied for model transformations, such as to open a transformation chain, to apply model-to-model and model-to-code transformation, to open and save a UML model target of a transformation among others. The task in these lines informs the test case to open a transformation chain file named *"config/transf_chain.fomda"*. This is executed before any test operation annotated with *@Test*.

3.1 Extended JUnit 4 with Java Reflection

Our proposed test case must inherit from *FomdaTestCase*. It owns at least an operation annotated with *@Test*, where test logic is programmed, and others annotated with *@Before*, used to ensure that test case pre-conditions are satisfied before executing any logical test. It is important to notice that such annotations were introduced by the regular JUnit 4 and are not part of our proposed annotations. Thus, the other annotations and functionalities are our contributions as an extension to JUnit 4 test engine.

This particular kind of test executes transformations using a TCDM specified with the *@IOTask* annotation shown in line 2. TCDM owns transformation specified in diverse transformation languages. However, we have exemplified only Java-based transformations. Then, the lines 4 and 5 show how to inject a transformation dependency to an attribute whose type handle executions, linking to the real transformation shown in Figure 3 (C). The injected instance of the model transformer named *"Implements Serialization"*. Thus, the executor assumes the task to orchestrate different and heterogeneous transformation languages (e.g., Java or Velocity). In [7], the authors advocate that this is a big challenge in current MDE tools, deserving in-depth discussions.

The extended JUnit test engine searches into the TCDM the transformation *"Implements Serialization"*: Lines 4 and 5 ensure that the required model transformer was found in the TCDM and it is placed into the attribute named *"executor"*. The *FomdaTestCase*, using Java Reflection, performs this. Moreover, line 9 ensures that the PDM was correctly imported and line 10 ensures that *"Remote Layer"* feature is selected in the PDM.

```
1   @FomdaTask(  tasks = { //Initialization tasks in order to open a domain model (PDM and TCDM)
2         @IOTask(inputFilePath = "config/transf_chain.fomda", kind = IOKind.INPUT, fileKind = WctFileHandlerKind.FOMDA)  }    )
3   public class ImplementsSerializationUnitTestCase extends FomdaTestCase {
        //This is our annotation that injects a transformation shown in Figure 3 (C)
4         @InjectTransformerConfig(reference = @TransformerReference(name = "Implements Serialization"))
5         private TransformationExecutor executor;
6         @Before()// This is a JUnit 4 annotation executed before the operation named testTransformations() in line 11
7         public void testInjectedDependencies() throws Throwable {
8           assertTrue(executor != null); //Ensure that the transformation executor was successful injected
9           assertNotNull(getFomdaModel());//Ensure that the transformation chain was imported correctly
10          assertTrue(getFomdaModel().isSelectedFeature("Remote Layer"));//Ensure that, in the PDM, the feature 'Remote Layer' is selected
          }
11        @Test()// This is a JUnit 4 annotation that executes the test logic
12        public void testTransformations() throws Throwable {
            // Programmatically creates an element of type class (analog to Figure 4 (B)) to test the injected instance in line 5
13          Model m = ElementFactory.instance().createModel();
14          Class entity = ElementFactory.instance().createClass(m);
15          entity.setName("Product");// Specifying programmatically the class shown in Figure 4 (B)
16          entity.assignStereotype("Entity");
17          Class value = (Class) executor.execute("implSpecPoint",entity);// Executes the transformation: parameter name and value
18          assertTrue(value.isRealizationFrom("java.lang.Serializable"));//Assert that transformation returned the correct result
```

Figure 5. Example of a unit test case to assert if model transformer 'Implements Serializable' do not fails

3.2 Unit Test Cases

The lines from 13 to 18 show the logic related to the test case: (a) UML elements of type *Model* and *Class* (lines 13 and 14) were programmatically created to be used as transformation input parameters, see the *entity:Class* input parameter shown in Figure 3 (A); (b) in line 17 a transformation is executed, using as input the created class and returning other element of type Class as a result; and (c) Finally, in line 18, the test asserts that the returned value from a transformation (a Class element) owns a Realization relationship with the data type *"java.lang.Serializable"*.

3.3 Testing Mutually Exclusive Transformations

Another type of unit test is shown in Figure 6. Such example ensures that abstract transformers are replaced in a MTC by concrete model transformers. In this sense, it asserts that between mutually exclusive transformers *"Generate JPA"* and *"Generate XDoclet"*, the last one is used in the generated MTC. Accordingly, lines 14 and 15 show the injection of a transformer named *"ORM to PSM"*. Notice that abstractions propagates it use in a MTC for a child. Therefore, or *"Generate JPA"* or *"Generate XDoclet"* is injected in line 14.

```
1  @FomdaTask(
2    tasks = {
     // ... open mtc "config/transf_chain.fomda",
     //after, import/open a UML model used as input
3      @IOTask(
4        inputFilePath = "xmis/mdwe_sample1.xmi",
5        kind=IOKind.INPUT, fileKind = WctFileHandlerKind.XMI,
6        inputModels={
     //assert that the imported model owns entityHash
7          @AssertInputModel(
     //this name is used further in test algorithm
8            targetKeyName="entityHash",
9            targetElements={
     //searches inside model for an element
10             @SearchElement(
           //from a specific instance
11                 elementType=org.wct.uml.Class.class,
           //and stereotyped with <<Entity>>
12                 stereotypedWith={"Entity"}
                ) } ) } ) } )
13 public class OrmToPsmTestCase extends FomdaTestCase{
14   @InjectTransformerConfig(
       reference = @TransformerReference(name = "ORM to PSM"))
15   private TransformationExecutor executor;
     @Before()// This is a JUnit 4 annotation
16   public void checkPreConditions() throws Exception{
17       assertTrue(isSelectedFeature("JPA"));
18       assertNotNull(executor);
19       assertTrue(executor.getName().equals("Generate JPA"));
     }
     @Test()// This is a JUnit 4 annotation
     public void testExecuteGenerateJPA() throws Throwable{
20       Model model = getInputElementAsModel(0);
21       assertNotNull(model);
22       List<org.wct.uml.Class> inputList =
                 getInputModelElement("entityHash");
23       if(inputList != null && !inputList.isEmpty()){
24          for(org.wct.uml.Class entClass : inputList){
25            assertNotNull(entClass);
26            Object obj = executor.execute("entity", entClass);
              ...
```

Figure 6. A unit test case supporting abstract MTs test

A test unit must assert the validity of transformations for a specific scenario: not a variant chain, but for the adapted transformations. In this sense, it is necessary to constraint that the test case shown in Figure 6 executes the operation *testExecuteGenerateJPA* only if JPA is used in support for ORM. With this in mind, line 17 specifies this constraint, which requires the selection of the feature *"JPA"* before the execution of the unit test case in line 20. Therefore, in case *"JPA"* is selected, the injected transformation must be named as *"Generate JPA"* as asserted in line 19.

3.4 Initializing the Test Case with Queries

Figure 6 presents other annotations and properties added to *@FomdaTask*. Lines 4 and 5 configure the test case to import a UML model [9] in XMI file format [27]. It is used in line 20, inside the test logic, as input for a transformation in line 26. It is possible also to import EMF-based models and extend the proposed API to import/export models in other languages.

In order to apply queries in the initialization of the test case, the line 6 assert that the input model (the UML model) contains a sort of elements decorated with some tags and stereotypes (line 12). Line 8 allow to place all found model elements that fulfill the conditions specified in lines 11 and 12 into a hash table, retrieved in line 22 of the test algorithm. Therefore, the exemplified annotations allow retrieving entity classes as the one illustrated in Figure 4 (B), iterating from a list (line 24) populated in test case initialization in line 8. Then, for each entity class, one of the concrete children for *"ORM to PSM"* is executed (lines 26 and 27).

3.5 Integration Test Cases

Integration test cases must assert that a sequence of transformations was correctly executed. In this sense, Figure 7 illustrates an integration test case for transformations of a TCDM discussed and exemplified in [5] that complements the TCDM illustrated in Figure 3 (A). This example brings many transformations representing all the Model Driven Architecture (MDA) recommended views [27], including Computation Independent Model (CIM); Platform Independent Model (PIM), Platform Specific Model (PSM) and Source-code.

```
1 public class DynamicMTCIntegrTestCase extends FomdaTestCase{

2      @InjectTransformerConfig(
        reference = @TransformerReference(name = "ORM to PSM"))
3      private TransformationExecutor ormExec;

4      @InjectTransformerConfig(
        reference = @TransformerReference(name = "Generate Code"))
5      private TransformationExecutor defCodeGen;

6      @InjectTransformerConfig(
        reference = @TransformerReference(name = "Apply ORM"))
7      private TransformationExecutor ormWizard;

8      @InjectTransformerConfig(
        reference = @TransformerReference(
           name = "Reverse Code to Model"))
9      private TransformationExecutor revEng;

       @Test()// This is a JUnit 4 annotation
10     public void testIntegration() throws Throwable{
   //Reverse a simple class to a model element in a CIM view (MDA)
11       org.wct.uml.Class cimClass = (org.wct.uml.Class)
           revEng.execute(("clazz", "src/test/SomeEntity.java");
12       assertNotNull(cimClass);
   //Use of a wizard that refines the CIM into a PIM view (MDA)
13       Object obj = ormWizard.execute("entity", cimClass);
14       org.wct.uml.Class pimClass = (org.wct.uml.Class) obj;
15       assertNotNull(pimClass);
   //Transform a PIM into a JPA or XDoclet dependent PSM
16       obj = ormExec.execute("entity", pimClass);
17       org.wct.uml.Class psmClass = (org.wct.uml.Class) obj;
18       assertNotNull(psmClass);
   //Generates Source-code using default source-code generator
19       defCodeGen.execute("entity", psmClass);
```

Figure 7. An integration test case testing MTC sequences

The difference between a unit test case and an integration test case is the need for execution of more than one model transformation in the test logic, as shown between lines 11 to 19. This brings an extra effort to write these types of test cases, since many possible MTCs may be acquired from the TCDM, as illustrates Figure 4 (A to F). In order to assert that transformations are correctly chained and executed, current proposals for integration test cases (owning MTC logic) develop many test cases for each chain derived from

the TCDM. This implies at least six possible MTCs to perform the example shown in Figure 4, requiring even more if other transformations for other features for *"ORM"* are used.

On the other hand, our proposal uses a single test case, facilitating the specification of automated integration test cases. In this sense, Line 11 exemplifies the execution of a transformation that applies reverse engineering from source-code to a UML model. This model is used as input for the second transformation task shown in line 13, which displays a wizard to annotate entity classes with a UML Profile. Then the *"ORM to PSM"* derived transformation is executed in line 16 and, finally, the entity class is used as input for a transformation from model-to-code that generates a Java source-code on line 19.

3.6 Preconditions to Execute Test Cases

It is possible to ensure that a complete set of features is selected in the PDM to attempt to a specific generated chain as shown in Figure 8. This is demonstrated in annotations supported inside the *@FomdaTask* into the property shown in line 3. The annotation *@AssertFeatures* is used to ensure that features are selected (line 5), hence others not, as well as to check if some features are mutually exclusive (line 11). This is an important aspect to test, because the PDM evolves and test cases must firstly ensure that their pre-conditions to execute a test are satisfied. Accordingly, the XOR relationships can be replaced by inclusive OR relationships in PDM, invalidating the test case logic. Therefore, in case a set of features are not more defined as XOR in the PDM, the current algorithm of a test case must be changed.

```
1  @FomdaTask(
2    selected features agree in pre-conditions
3  assertFeatures={
4     @AssertFeatures(
        //a rule can be IS_SELECTED, IS_NOT_SELECTED, IS_XOR,
        //IS_OR, IS_OPTIONAL, IS_MANDATORY, IS_DEPENDENCY
5       rule=AssertFeatureKind.IS_SELECTED,
6       features={
9         @FeatureReference(featureName="Remote Layer"),
10        @FeatureReference(featureName="JPA")
        }    ),
      @AssertFeatures(
11      rule=AssertFeatureKind.IS_XOR,
        features={
12        @FeatureReference(featureName="JPA"),
13        @FeatureReference(featureName="XDoclet")
   }     )     }   )
14 public class FeatureDependentTestCase extends FomdaTestCase{
```

Figure 8. Preconditions to execute a test case

3.7 Variant Test Suites

Aforementioned annotations allow embedding test cases inside a generic test suite as shown in Figure 9. In this sense, some test cases that do not satisfy the rules are ignored in the initialization. In this sense, one can register in *@Suite.SuiteClasses,* some test cases developed for mutually exclusive transformations that should not be executed in the same test suite. Therefore, pre-conditions ensure that some tests are not executed.

```
@Suite.SuiteClasses({// This is a JUnit 4 annotation
          ImplementsSerializationUnitTestCase.class,
          GenerateDBScriptTestCase.class,
          OrmToPsmTestCase.class,
          DynamicMTCIntegrTestCase.class,
          FeatureDependentTestCase.class
})public class TestSuite extends FomdaTestCase{   }
```

Figure 9. JUnit Test Suites with Variant Test Cases

3.8 Generating Test Cases

The discussed set of annotations and functionalities available in the FomdaTestCase class is strictly applicable to test dynamic compositions among TCDM and PDM elements. In other words, they are very useful to test model transformations that run inside WCT transformation engine. However, it is also possible to use TCDM and PDM to generate model transformation assets that can be used and executed in other transformation engines. This is exemplified through generative techniques in [33][3] for software test cases extractions. Moreover, in [5] we exemplified the generation of many types of model transformation assets and in [6] we demonstrate how to generate MTCs taking as input the TCDM and PDM.

In fact, test cases are also model transformation assets, target for adaptations inside a TCDM. Therefore, the same concepts we have been applying in model transformation fragments [5][6] are also applicable to generate reusable test cases.

3.9 Test Cases in Action

Figure 10 (A) shows a screenshot of the execution of aforementioned test cases. In the failure trace area is shown one test case that uses the annotation *@AssertInputModel*, reporting that some element required to execute the test case is not available in the input model. This figure also demonstrates the use of existing Eclipse plugins to execute JUnit test case. Therefore, only the JUnit class was extended, not requiring extending plugins to execute variant transformations and variant test cases.

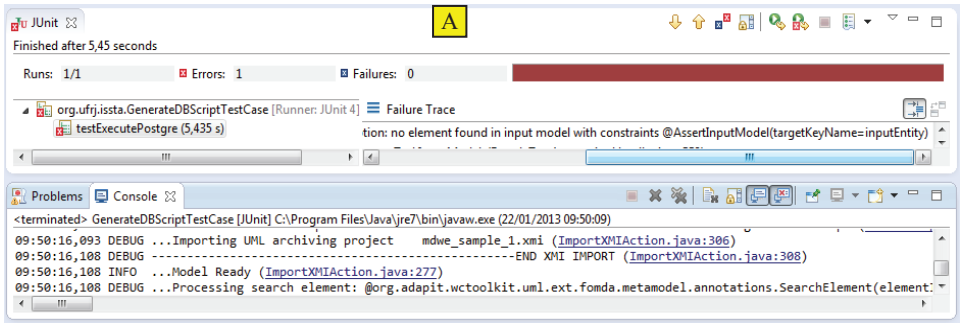
4. VALIDATION AND ONGOING WORKS

The FOMDA methodology and WCT tool have been used since 2007 at Adapit, a small Brazilian software development company, to create model-driven information systems with Java programming language. The company used the tool to develop some projects discussed in [5] that required the use of variant MTCs.

In order to have a clearer idea about these variants, we have computed a total of 343 model transformation assets, including the small ones shown in Figure 3 (B) and (C). Thus, do not confuse this number with complete model transformations. In this sense, a total of 193 model-to-text (53 white box and 140 compiled black box) and 150 model-to-model transformation units of type compiled black box. These assets are tested with 41 unit test cases and only 5 integration tests, developed to execute dynamic MTCs. The low number of integration tests is justified because we use dynamic compositions discussed in Section 3. On the other hand, this number of integration would be bigger if MTCs had been generated through generative techniques.

In order to improve FOMDA methodology, we are changing our practices towards the design of TCDM. Before the development of proposed Java annotations, we used to use a top-down approach by firstly designing the PDM; then TCDM; then we develop variant model transformations; to then execute tests. This top-down approach is suggested by domain engineering proposals [23] [10] and, so far, has been satisfactory for our necessities. However, we are researching the implications in using agile testing [13], which requires a bottom-up approach, starting from test cases. Thus, we are adapting Test Driven Development (TDD) practices [13] to create model transformation assets and upload them into the TCDM with Java Reflection.

For instance, we noticed that the Java reflection reverse annotated model transformers into the TCDM, similarly as Kim et al. [19]. This allows uploading information related to features and also to compose transformations into TCDM. This practice has been used in a case study regarding development of started from scratch variant model transformations in support for wireless sensor networks domain in [29], whose preliminary results are promising.



B

Number of features		148
Number of transformations		45
Try	With Injection (ms)	Without (ms)
1	4381	4474
2	3965	3518
3	3960	3385
4	3989	3544
5	3881	3612
6	4361	3590
7	4685	3657
8	4230	4439
9	4090	3821
10	4063	3640
Average	4160,5	3768

Figure 10. A) Screenshot of the Extended JUnit Test API in Action, Executed in an Eclipse IDE. B) Test Case Benchmarks

According to our observations in the recent study, the performance of the proposed extension for JUnit is similar to the regular API. This is illustrated on the benchmark shown in Figure 10 (B) considering a PC with Windows 7 32 bits, Intel Core 2 Duo 2.93 GHz, 3 GB RAM. This figure compares the required milliseconds to execute two integration tests owning 16 transformations: one using manual instantiation for transformations and the other one using dependency injections (using XOR transformations) with annotations of type *@InjectTransformerConfig*. The dependency injection requires 392 milliseconds more than manually instantiated requires. The benchmark considered a domain model composed by 148 features in the PDM and 45 transformation units in the TCDM. It is also necessary 3629 milliseconds on average to load the domain model for each test case execution.

The performance is a problem when executing transformations that requires the display of graphic user interfaces (GUI), such as the transformation task referred in Figure 7, line 13. This is a limitation of the JUnit engine that manages all instances of Java objects, overloading the Java Virtual Machine (JVM).

5. DRAWBACKS AND LIMITATIONS

Except for the generative approach, discussed in Section 3.8, that is independent from a model transformation engine; our proposed extensions require the use of WCT as a transformation engine. This is a limitation for the applicability of the proposed JUnit extension, but the proposed annotations can be reused and re-implemented in support for other engines. Moreover, WCT has an extensible framework to include other transformation languages. Through these extensions, our proposal can be applied to execute any Java program.

A limitation of this work is that it exemplifies model transformations developed in Java, while other technologies are used to write model transformers. Considering that different model transformation frameworks are available and some do not support the same set of languages [39][34][5], transformation rules programmed with heterogeneous languages impose threats to the validity of the exemplified transformation assets. This limitation in our work is suppressed by T-Core framework, which supports the execution and validation of heterogeneous model transformations in MTCs [34].

Another threat to the proposed solution is the interoperability of input and output models used among transformations. Some transformation languages such as ATL and QVT [15] require importing and exporting a model before and after executing a transformation in an MTC. In this sense, a model can be exported by a transformation 1 in a version not supported by a sequent transformation 2. This kind of constraint should be detected in a test case and during the transformation chain design. We still have no solution for this eventual problem during the design of TCDM, but test cases are capable to detect fail in a sequence of transformations.

6. RELATED WORK

Literature of the area reports the use of automated test cases applied to software product lines that does not require modifications in current test engines. Santos et al. applied a study in two products to extract features that are used in test cases to ensure that derived products are in conformance with the expected results [33]. Their work focuses on techniques to extract information from existing source-code and uploads this information in a feature model. Kim et al. have also used Java annotations in software product source-code to extract fragments to be included in reusable core assets [19]. They suggest that some features are behavior-irrelevant for tests, since they do not change the application logic, and can be excluded from the test cases.

These works show a clear difference from our proposal: they did not use annotations to extend JUnit to execute dynamic programs, as we are proposing. On the other hand, we did not extract features of a software domain model. Moreover, in our solution we need to assert that correct model transformation are used in a dynamic MTC and also ensure that transformation return the correct result after execution. Therefore, our work is not directly related to these works, since dynamic and variant model transformations require some different test techniques than those used to test fragments for a family of software applications.

Regarding the test for model transformation chains, Küster et al. used incremental development of model transformations using automated tests to assert their validity in an MTC [20]. This is the most similar proposal regarding automated test cases that we have found in literature. Although providing very important guidance to develop incremental transformations with test driven development practices, the tests are applied over an MTC simpler than the ones that we have exemplified, since they did not consider variant MTCs. In order to execute a dynamic MTC, our model transformation assets must deal with commonalities and variability, which should be tested. Therefore, besides validating the return of transformations, test cases must ensure that bindings between transformation input and outputs attempt to specific configurations established when a new set of transformation compositions is selected for a particular software project.

Other work is dedicated to applying validation in model transformation compositions. Fleurey et al presented strategies that adapt traditional tests to better suit model transformation compositions, considering metamodels and model transformation chains [12].

More recently, in order to apply these validations, Etien et al. [11] and Yie et al. [39] extended these concepts with tool solutions to design MTCs and validate their compositions through IO parameters. Although these works are interesting and contribute to validate a composition established in a TCDM, they are not applicable to the type of test cases that we have exemplified in this paper. Therefore, these contributions don't relate directly to our proposal.

Hervieu et al. executed a similar study in the industry towards the use of SPL-based techniques in support for test adaptations [14]. They propose adapting test specifications created to test transformation chains that must consider a PDM. Although an interesting work, since it was applied in industry and reinforce the necessity to specific techniques to deal with model transformation assets, they do not have an approach to support the test of variant model transformations. Moreover, their study is focused in reusing textual guidance for test cases, not in adapting automated test cases.

Our work is complementary and presents a contribution in comparison to related works. In order to provide a specific solution to test variant model transformation assets, our contribution facilitates the specification of automated JUnit test cases.

In this sense, our contribution is directly important to test model transformation assets specified and generated through some related works as follows: 1) Almeida et al. proposed a solution to compose model transformations in MTCs [1]; 2) Boas proposed the design of a MTC using workflow to model a transformation process according MDA views [8]; 3) Basso et al. [4] and Völter et al. [37] applied SPL-based techniques to specify dynamic MTC; 4) Vanhooff et al. proposed a MTC modeling language to generate specifications used by some transformation execution engines [36] and Wagelaar et al. proposed a framework to chain black-box model transformations [38], similar as those exemplified in Section 3; 5) Etien et al. complemented these works to include validation for metamodels interoperated among different transformation compositions [11], similarly as Yie et al. that applied validation between transformation IO parameters considering an MTC specification [39]; 6) Rosenmüller et al. applied techniques to control dynamic SPLs [32] and, despite not being related to model transformation reuse, can also be applied in this context; 7) Aranega et al. [2] and Basso et al. [6] applied SPL-based techniques to fragment and merge model transformation assets. Although these works present an important contribution as a means of reuse techniques, they have not tackled automated test cases to validate the generated assets. Therefore, we present singular contributions for MDE-based techniques.

7. CONCLUSION REMARKS

In order to reuse model transformations, some MDE proposals are using Software Product Line (SPL) techniques to fragment and merge pieces of transformations. This brings an extra-effort to test the adapted assets, since many test cases must be specified to test the generated products. Due to the necessity to adopt a test driven development practice, we present a solution to specify automated test cases for variant model transformation assets.

In this sense, this paper bridges the gap between variant model transformations and automated test cases. Thus, we presented an extension of the JUnit 4 API that allows specifying variant executions for test cases. The exemplified scenario is based on a real experience and allowed to unveil some import direction towards future works to improve test practices and a methodology, namely FOMDA, used to design reusable model transformations.

Differently from our previous works that focused on constructing transformation assets with the FOMDA methodology in a top-down solution, this one presents a solution to automate unit and integration tests that allows applying a bottom-up methodology, starting from tests. In this sense, a set of Java annotations was presented. They are reversed with Java reflection, allowing the execution of dynamic and variant test cases.

Due to the use of SPL-based techniques in model transformation assets, it is necessary to assert that these assets are valid after adaptations. With this in mind, some automated test cases were developed to test each possible fail and error condition regarding this context. So far, the set of test cases exemplified in this paper has never been tackled before in the literature of the area. Therefore, this paper presents a singular contribution for MDE researchers and practitioners.

8. ACKNOWLEDGMENTS

This work was partially supported by the Brazilian agencies CAPES and CNPq.

9. REFERENCES

[1] Almeida, J., Dijkman, R., Sinderen, M., and Pires, L. *Platform-independent modeling in MDA: Supporting abstract platforms*. In Proc. of Model-Driven Architecture: Foundations and Applications, June 2004. pp 217-231.

[2] Aranega, V., Etien, A., and Mosser, S. *Using feature models to tame the complexity of model transformation engineering*. In ACM/IEEE 15th International Conference on Model Driven Engineering Languages and Systems MODELS 2012.

[3] Asaithambi, S.P.R., and Jarzabek, S. *Towards Test Case Reuse: A Study of Redundancies in Android Platform Test Libraries*. In ICSR 2013, pp. 49–64.

[4] Basso, F. P., Oliveira, T. C. and Becker, L. B. *Using the FOMDA Approach to Support Object-Oriented Real-Time Systems Development*; In Proc. of International Symposium on Object and Component-Oriented Real-Time Distributed Computing. Gyeongju, Korea. 2006. pp. 374-381.

[5] Basso, F. P., Pillat, R. M., Oliveira, T. C. and Becker, L. B. *Supporting large scale model transformation reuse.* In 12th International Conference on Generative Programming: Concepts & Experiences (GPCE'13), Indianapolis, USA. 2013. pp 169-178.

[6] Basso, F. P., Pillat, R. M., Oliveira, T. C. and Fabro, M. D. D. *Generative Adaptation of Model Transformation Assets: Experiences, Lessons and Drawbacks*. In Proceedings of ACM Symposium on Applied Computing (SAC'14) Gyeongju, Korea. March 24 - 28, 2014 (to appear).

[7] Baudry, B., Ghosh, S., Fleurey, F., France, R., Traon, Y. L., and Mottu, J. M. *Barriers to systematic model transformation testing.* In Communications of the ACM Volume 53, Issue 6, 1 June 2010, pp 139-143.

[8] Boas, G. *From the Workfloor: Developing Workflow for the Generative Model Transformer*. OOPSLA 2005.

[9] Boch, G., Rumbaugh, J. and Jacobson, I. *The Unified Modeling Language: User Guide*, Addison-Wesley, 1999.

[10] Eisenecker, U. and Czarnecki, K. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley, 2000.

[11] Etien, A., Muller, A., Legrand, T., and Blanc, X. *Combining Independent Model Transformations*. In Proceedings of ACM Symposium on Applied Computing (SAC'10). 2010.

[12] Fleurey, F., Steel, J. and Baudry, B.. *Validation in model-driven engineering: Testing model transformations*. In 1st International Workshop on Model, Design and Validation, SIVOES - MoDeVa 2004. pp 29-40.

[13] Hellmann, T. D., Sharma, A., Ferreira, J., and Maurer, F. *Agile testing: Past, present, and future - Charting a systematic map of testing in agile software development.* Proceedings - 2012 Agile Conference, Agile 2012. pp 55-63.

[14] Hervieu, A., Baudry, B., and Gotlieb, A. *Managing execution environment variability during software testing: An industrial experience.* In International Conference on Testing Software and Systems, ICTSS 2012. pp 24-38.

[15] Jouault, F. and Kurtev, I. *On the Architectural Alignment of ATL and QVT*. In: Proceedings of ACM Symposium on Applied Computing (SAC 06), Model Transformation Track. April 2006.

[16] *JUnit 4 API*. At December 2013. Available at <http://en.wikipedia.org/wiki/JUnit>.

[17] Kakarontzas, G., Stamelos, I. and Katsaros, P. *Product line variability with elastic components and test-driven development*. In International Conference on Computational Intelligence for Modelling Control and Automation, CIMCA 2008. pp 146-151.

[18] Kang, K. C., Kim, S., Lee, J., Kim, K., Shin, E. and Huh, M. *A feature-oriented reuse method with domain-specific reference architectures*. Ann. Softw. Eng., Jan. 1998, 5:143–168.

[19] Kim, C. H. P., Batory, D. S., and Khurshid, S. *Reducing combinatorics in testing product lines*. In Proceedings of the tenth international conference on Aspect-oriented software development, AOSD '11, 2011, pp 57-68.

[20] Küster, J. M., Gshwind, T., and Zimmermann O. *Incremental Development of Model Transformation Chains Using Automated Testing.* Springer-Verlag MODELS 2009, Berlin Heidelberg 2009. LNCS 5795, pp 733-744, 2009.

[21] Lúcio, L., Bruno, B., and Vasco, A. A Technique for Automatic Validation of Model Transformations. In Model Driven Engineering Languages and Systems (MODELS 10), 2010. pp 136-150.

[22] McGregor, J. D. "*Agile software product lines, deconstructed"*. Journal of Object Technology, 7(8), Nov., 2008, 7-19.

[23] McGregor, J.*Testing a Software Product Line*, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania, CMU/SEI-2001-TR-022, 2001.

[24] Mohan, K., Ramesh B. and Sugumaran, V. *"Integrating software product line engineering and agile development"*. IEEE Software, 2010.

[25] Nanda, M. G., Mani, S., Sinha, V. S., and Sinha, S. *Demystifying Model Transformations: An Approach Based on Automated Rule Inference.* OOPSLA 2009, Orlando, USA.

[26] Noor, M. A., Rabiser R., and Grünbacher, P. "*Agile product line planning: a collaborative approach and a case study"*. Journal of Systems and Software, 2007, 81(6), 868-882.

[27] Object Management Group - *MDA Specifications*. June 2011. Available at <http://www.omg.org/mda/specs.htm>.

[28] Offutt, J., and Abdurazik, A. *Generating Tests from UML Specifications*, 2nd Intl. Conference on UML'99, 1999.

[29] Paulon, A. R., Fröhlich, A. A., Becker, L. B., and Basso, F. P. *Model-Driven Development of WSN Applications*. In III Simpósio Brasileiro de Engenharia de Sistemas Computacionais (SBESC) 2013.

[30] Perrouin, G., Sen, S., Klein, J., Baudry, B. and Traon, Y. L. *Automated and scalable T-wise test case generation strategies for Software Product Lines.* ICST 2010 - 3rd International Conference on Software Testing, Verification and Validation. 2010. pp 459-468.

[31] Reuys, A., Kamsties, E., Klaus, P., and Reis, S. *Model-Based System Testing of Software Product Families*. In Advanced Information Systems Engineering, Lecture Notes in Computer Science, volume 3520. 2005. pp 519-534.

[32] Rosenmüller, M., Siegmund, N., Pukall, M., and Apel,S. *Tailoring dynamic software product lines*. In 10th International Conference on Generative Programming (GPCE'11), 2011. 47(3):3–12.

[33] Santos, A., Gaia, F., Figueiredo, E., Neto, P. S., and Araújo, J. *Test-based SPL extraction: an exploratory study*. In Proceedings of ACM Symposium on Applied Computing (SAC'13). 2013. pp 1031-1036.

[34] Syriani, E., Vangheluwe, H. and LaShomb, B. *T-Core: a framework for custom-built model transformation engines*. Software & Systems Modeling Journal. DOI: 10.1007/s10270-013-0370-4. 2013.

[35] Tekinerdogan, B., Bilir, S. and Abatlevi, C. *Integrating Platform Selection Rules in the Model Driven Architecture Approach*. In Proc. of Model-Driven Architecture: Foundations and Applications, June 2004. pp 184-200.

[36] Vanhooff, B., Baelen, S. V., Hovsepyan, A. Joosen, W. and Berbers, Y. *Towards a Transformation Chain Modeling Language*. Springer-Verlag and SAMOS 2006, Berlin Heidelberg. LNCS 4017, 2006, pp. 39–48.

[37] Völter, M. and Groher, I. *Handling variability in model transformations and generators*. In Proceedings of the 7th OOPSLA Workshop on Domain-Specific Modeling (DSM'07), 2007.

[38] Wagelaar, D. *Blackbox Composition of Model Transformations using Domain-Specific Modeling Languages*. In First European Workshop on Composition of Model Transformations. 2006.

[39] Yie, A., Casallas, R., Deridder, D. and Wagelaar, D. *Realizing model transformation chain interoperability*. Software & Systems Modeling, 2012, 11(1):55–75.