Automatic Derivation of Software Test-Cases Code from Formal Models

Ilan Rosenfeld[1] and Claudia Pons[1,2]
[(1)] Facultad de Informática, Universidad Nacional de La Plata (UNLP)
Calle 50 esq. 120, La Plata, Buenos Aires
[(2)] Universidad Abierta Interamericana (UAI)
cpons@info.unlp.edu.ar

Abstract

Model-Driven Testing or MDT is a new and promising approach for software testing automation that can significantly reduce the efforts in the testing cycle of every software development. It consists in a black box test that uses structural and behavioral models to automate the tests generation process. In this context, we developed a tool which allows developers to translate a data model with formal constraints to its corresponding Java code, automating the generation of strong test-cases codes and specifying them not only in java language but also in two formal languages, such as OCL and Alloy. This tool gives a trustworthy and verifiable support with different techniques. In this way, the test-cases code generation process is improved and its quality enhanced..

1. Introduction

In the last few years, model-driven development [1] (MDD) has become very popular in the software engineering environment. The development of new technologies and innovations which aim to give models the main and active role in the software development process, against traditional approaches, let the design and software be independent from the architecture and platform, with system portability. Through a series of transformations, a platform independent model is translated into source code, dependent on a specific platform. As a consequence, the system productivity is enhanced, its quality enriched, and its comprehension, evolution, maintenance and reutilization are improved.

The success of any MDD project depends heavily on the quality of the source models. They must be accurate, consistent and complete.

When thinking about models, we use to consider graphic notations such as UML [3]. Usually, UML models consist of diagrams completed with natural language descriptions. The problem of these descriptions is that even though they are easy to write and understand, they are ambiguous. To overcome this problem, OCL (Object Constraint Language) [4] was born. It is a textual language with a formal foundation, based on the Set Theory and First-order Logic, but with an object-oriented nature that facilitates its understanding. OCL is the standard language to define integrity constraints on UML models. In this way, the combination UML/OCL is considered a formal language.

One of the branches of MDD is the Model-Driven Testing (MDT) [2], a new approach for software testing automation, which can significantly reduce the efforts in the tedious testing cycle of software development. It consists in a black box testing technique that uses structural and behavioral models to automate the generation of test-cases code.

After analyzing several automatic code generation tools from software models, we conclude that they are not taking full advantage of what formal modeling languages offer to testing automation. For this reason our work consisted in building a new software tool for automating the generation of the code of test-cases, but with strong formal foundation.

The tool allows developers to automatically generate Java code from a UML/OCL model, including both the model classes and their test-cases code. The generated test-cases code is written in Java but it is enhanced with formal specifications which allow the static and dynamic formal analysis of the system. In this way, the test-cases code generation process is improved and its quality enhanced.

The rest of the paper is organized as follows. Section 2 describes the basic features of a new software tool for test-cases code generation. Section 3 presents an extension of the tool which improves the tests through the application of a richer formalism. Section 4 discusses a set of related works. Finally, conclusions are presented in section 5.

2. A Tool for Test-cases Code Generation

In this section we describe the characteristics of a new software tool for automating the generation of test-cases code. The tool was developed taking advantage of the Eclipse Modeling Project (https://eclipse.org/modeling) that focuses on the evolution and promotion of model-based development technologies within the Eclipse community by providing a unified set of modeling frameworks, tooling, and standards implementations. First, we briefly describe the main elements of Eclipse that we included in our development. Then, we explain the construction process and features of the tool.

2.1. Eclipse Modeling Tools

The Eclipse Modeling Framework (EMF) [6] represents a set of plugins that can be used to model a data model and generate code or other kind of output based on that model. There is a difference between the metamodel and the concrete model: the metamodel describes the model structure, whereas the model is a concrete instance of it. It provides a framework that can be included to store the model information, which uses a default data format called XMI (XML metadata interchange) to persist model data. It allows the developer to create metamodels by different means, for example: XMI, Java annotations, XML schemas, etc.

Papyrus [7] is a subproject component that aims to provide an integrated environment and usable by the user to edit any type of EMF model, supporting UML and related modeling languages such as MARTE. Papyrus provides diagrams editors for EMF based modeling languages such as UML2 and the chance of integrating these editors (which might be GMF based or not) with other tools. It also offers an advanced support for UML profiles, allowing the user to define standard UML2 based DSL editors and their extension mechanism. Its main feature, related to what was mentioned above, is a very powerful set of personalization mechanisms that can be used to create user defined Papyrus perspectives, having the same appearance and simulating a domain specific language (DSL) editor.

Acceleo [8] is an open source project, licensed under EPL (Eclipse public license), available for free. It was designed for MDA technologies developers to increment their software development productivity. It allows the generation of files using UML, MOF and EMF modules. It has a complete integration both with Eclipse and the EMF framework, code and model synchronization, incremental generation, easy updating and templates handling, syntax coloring, auto-complete and errors detection. It requires having a previous knowledge both in Java and modeling.

2.2. Test Code Generation Process

Starting from an OCL/UML data model, the Java code will be automatically generated, creating the classes with their corresponding test-cases code and an OCL file which will contain all the formal constraints in a centralized form. The process is carried out in three steps, as described below.

2.2.1. Creating the data model with Papyrus

When creating a Papyrus project with the Eclipse IDE, a default UML class diagram will be created in three formats: traditional model view (.di), XML annotations (.notation) and Directories tree (.uml). The focus of the tool is on the .di file, from where we can create a traditional class diagram, such as the one displayed in figure 1. The model in the figure represents a university institution, containing Students, Teachers, Subjects, Careers and Careers Plans, among others. The diagram also includes a set of OCL restrictions (the palette Constraint elements) representing invariants and being associated with specific classes. For example, Student are not allow to be enrolled in more than one career, being reflected in the following OCL invariant,

Context Student inv:
self.careers -> size() = 1

We can also see a more complex invariant, defining that in order to teach a subject, a teacher must have a specialty on its area, being written as follows,

Context Subject inv:
self.teachers->forAll(o | o.specialtie->includes(self.area))

In Papyrus, the OCL invariants are associated to a model class through a pointing arrow, as we can see in the figure 1.

There are other OCL constraints at the model, not visible at first sight, which represent the pre and post conditions of its defined operations. For example, for the *enrolSubject(subject)* operation of the Student class, which enrolls the student to a subject, there is an OCL pre condition specifying that in order to enroll in a subject a student must have already passed all its correlatives, as follows,

context *Student::enrolSubject(subject)*
pre*: self.passedSubjects->includesAll(subject.correlatives)*

Also, another precondition which checks that the subject inscription is enabled is defined as follows,

context *Student::enrolSubject(subject)*
pre*:subject.inscriptionAllowed=true*

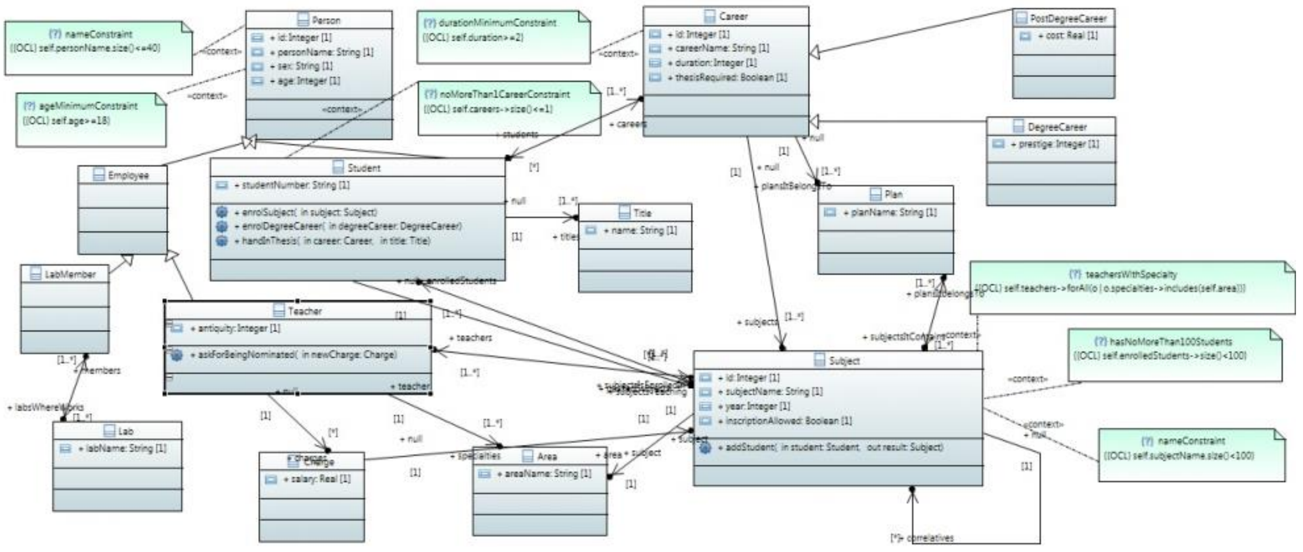


Figure 1. Class diagram

Then, we define two post conditions for the method. The first one is called *isEnrolledInSubject* and checks that the specified subject has been actually added to the collection, with the following body,

***post**: self.subjectsIsEnrolledIn-> includes(subject)*

The second one, named *adds1Subject*, specifies that,

***post**: self.subjectsIsEnrolledIn->size() = self.subjectsIsEnrolledIn@pre->size()+1*

This last one checks that the collection size is incremented in one. The expression *self.subjectsIsEnrolledIn@pre* represents the objects collection before its modification.

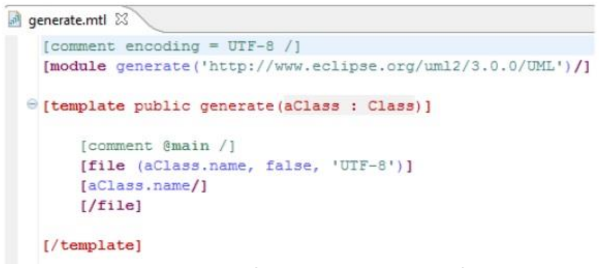
The implementation code should check that all the pre and the post conditions are valid when executing the methods.

The tool we implemented also allows us to define the body of each class method in a different range of languages and formats. In the case study of this paper we define methods bodies using OCL; since this format is quite similar to the Java syntax, its later translation (from the model class into the Java .class file) will be almost direct.

2.2.2. Acceleo translation code

For this case study, we choose the UML metamodel type (the tool give us the chance of using other types). When generating the acceleo file, the following elements will be generated:

- ✓ Two java classes, *Activator.java* and *Generate.java*: configuration files, specifying the included libraries among other things. In this case, we will leave their default values.
- ✓ An Acceleo module called *generate.mtl*: we will write our translation code into this module. Its default code can be seen in figure 2 (to comprehend its syntax you can check the official documentation in [8]).


generate.mtl

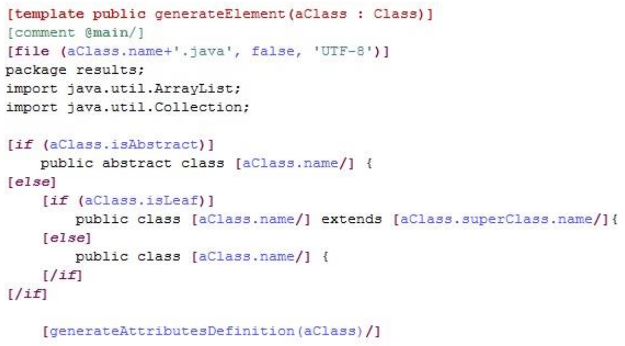
```
[comment encoding = UTF-8 /]
[module generate('http://www.eclipse.org/uml2/3.0.0/UML')/]

[template public generate(aClass : Class)]

    [comment @main /]
    [file (aClass.name, false, 'UTF-8')]
    [aClass.name/]
    [/file]

[/template]
```

Figure 2. Default generate.mtl file

The first step we must do is to choose an UML model from which generate the corresponding classes, so we will attach the recently created model as the source model in the Acceleo configuration.

Since this code is extensive and the main objective of the work is not to analyze it in detail, we will just focus on its most relevant parts. The code loops over every class of the UML source model, and for each one it creates a .java class with its name, and another one

TestClassToTest.java which tests it. Also, it creates the integration test, which runs every other generated test in only one step and returns its verdict, and the file *University*.ocl, that will have every OCL constraint defined associated to its context and centralized. We can see the beginning of the final Acceleo code in figure 3.



```
[template public generateElement(aClass : Class)]
[comment @main/]
[file (aClass.name+'.java', false, 'UTF-8')]
package results;
import java.util.ArrayList;
import java.util.Collection;

[if (aClass.isAbstract)]
    public abstract class [aClass.name/] {
[else]
    [if (aClass.isLeaf)]
        public class [aClass.name/] extends [aClass.superClass.name/]{
    [else]
        public class [aClass.name/] {
    [/if]
[/if]

    [generateAttributesDefinition(aClass)/]
```

Figure 3. Acceleo code

The *generateElement* template is executed for each class of the model. An abstract class, a subclass or a regular class are created on each of them, also invoking the method *generarDefAtributos* that, representing another template, defines the class attributes, always with protected as the access modifier to make them accessible from the whole package and from their tests.

For each class, an internal class representing a checker is also generated (see figure 4). This checker consists in two methods, *respectInvariants(classInstance)* and *respectCondition(condition).* Its main objective is to use it whenever a class instance needs to be updated to ensure its invariants keep respected.

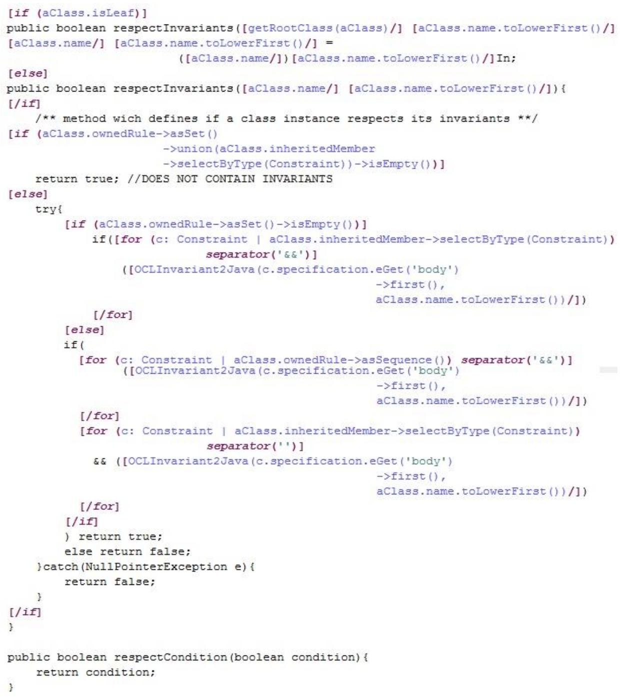
At the same time, the class constructor is generated, which checks through the checker that an instance respects its invariants when assigning its attributes. If not, it returns an instance with all its default values.

Generated getters are regular getters, returning the desired attribute. Instead, setters follow this procedure:

1. Save the current instance state through the *saveState* generated method
2. Set the attribute value based on the received parameter.
3. Check the instance still respects its invariants. If not, goes to step 4
4. Return the instance with its previous status, using the *returnState* generated method.

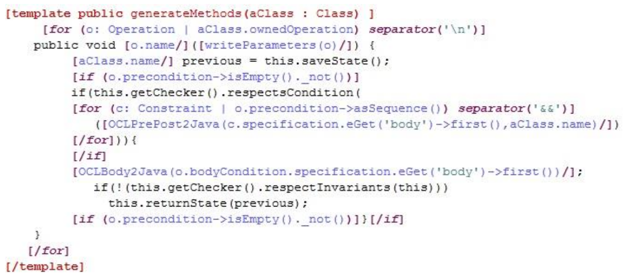
When defining each class method (figure 5), a copy of the object is generated with the name *previous*. Then, the method preconditions are checked. If they fail, the method execution terminates without modifying the instance. If they succeed, the method is executed and then the instance invariants are checked; if they are not being respected, the instance is returned to its previous status using the created copy, having the method no effect on the instance.

Generated tests for each class extend from the special class *TestCase* in order to test their methods through the JUnit[9] library.



```
[if (aClass.isLeaf)]
public boolean respectInvariants([getRootClass(aClass)/] [aClass.name.toLowerFirst()/]
[aClass.name/] [aClass.name.toLowerFirst()/] =
                    ([aClass.name/])[aClass.name.toLowerFirst()/]In;
[else]
public boolean respectInvariants([aClass.name/] [aClass.name.toLowerFirst()/]){
[/if]
    /** method wich defines if a class instance respects its invariants **/
[if (aClass.ownedRule->asSet()
                    ->union(aClass.inheritedMember
                    ->selectByType(Constraint))->isEmpty())]
    return true; //DOES NOT CONTAIN INVARIANTS
[else]
    try{
        [if (aClass.ownedRule->asSet()->isEmpty())]
            if([for (c: Constraint | aClass.inheritedMember->selectByType(Constraint))
                            separator('&&')]
                ([OCLInvariant2Java(c.specification.eGet('body')
                                                ->first(),
                                                aClass.name.toLowerFirst())/])
            [/for]
        [else]
        if(
          [for (c: Constraint | aClass.ownedRule->asSequence()) separator('&&')]
                ([OCLInvariant2Java(c.specification.eGet('body')
                                                ->first(),
                                                aClass.name.toLowerFirst())/])
          [/for]
          [for (c: Constraint | aClass.inheritedMember->selectByType(Constraint))
                            separator('')]
            && ([OCLInvariant2Java(c.specification.eGet('body')
                                                ->first(),
                                                aClass.name.toLowerFirst())/])
          [/for]
        [/if]
        ) return true;
        else return false;
    }catch(NullPointerException e){
        return false;
    }
[/if]
}

public boolean respectCondition(boolean condition){
    return condition;
}
```

Figure 4. Checker generation



```
[template public generateMethods(aClass : Class) ]
    [for (o: Operation | aClass.ownedOperation) separator('\n')]
   public void [o.name/]([writeParameters(o)/]) {
        [aClass.name/] previous = this.saveState();
        [if (o.precondition->isEmpty()._not())]
        if(this.getChecker().respectsCondition(
        [for (c: Constraint | o.precondition->asSequence()) separator('&&')]
           ([OCLPrePost2Java(c.specification.eGet('body')->first(),aClass.name)/])
        [/for])){
        [/if]
        [OCLBody2Java(o.bodyCondition.specification.eGet('body')->first())/];
           if(!(this.getChecker().respectInvariants(this)))
             this.returnState(previous);
        [if (o.precondition->isEmpty()._not())]}[/if]
    }
   [/for]
[/template]
```

Figure 5. Generation of each class methods

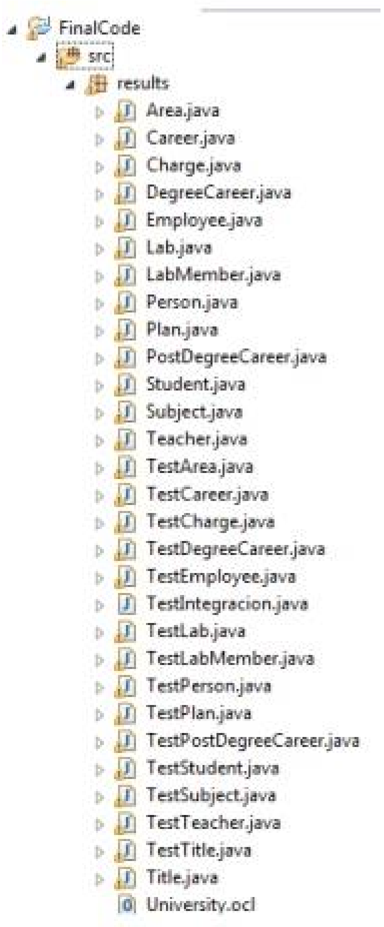
The main idea is to associate a checker mock (simulated object to which a specific behavior is set using Mockito [10] library) to the class instance and then verify for each method to test that, if we set that the object passes the method preconditions and the invariants are always respected (which includes after executing the method body) through its checker, then method post conditions must hold as well. Any other case (passing preconditions but not the invariants or not passing preconditions) may lead to pass post conditions or not, but we do not consider this.

Now that we have defined and analyzed our code, the next step is its execution. Running the *generate.mtl* file as an Acceleo application, the classes and test cases code defined before will be generated.

2.2.3. Analyzing the results

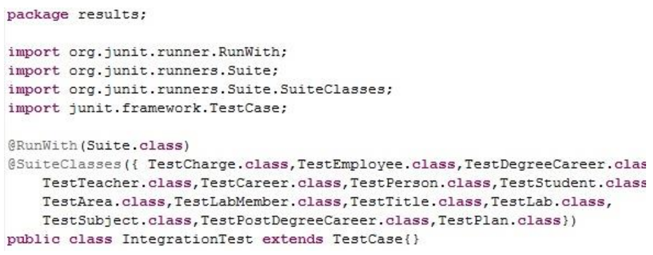
After executing the *generate.mtl* file, the corresponding *.java* classes and the *.ocl* le are generated (see in figure 6). Integration test can be run in order to check in just one step that every generated test passes, as shown in figure 7. Regarding to the generated code for each class, we can see a part of the Student class code in figures 8 and 9, and an example Test in figure 10, in which, after validating invariants and pre conditions, it must be assured that post conditions hold as well.

Methods that include *OCLToJava* on their names translate OCL bodies to its corresponding Java code.



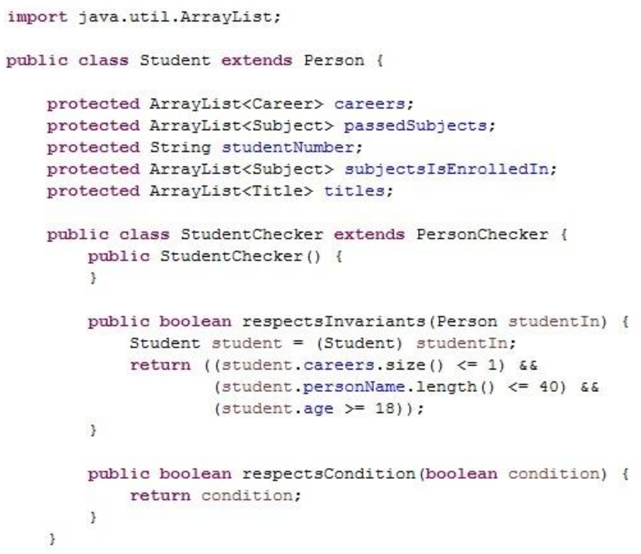
```
FinalCode
  src
    results
      Area.java
      Career.java
      Charge.java
      DegreeCareer.java
      Employee.java
      Lab.java
      LabMember.java
      Person.java
      Plan.java
      PostDegreeCareer.java
      Student.java
      Subject.java
      Teacher.java
      TestArea.java
      TestCareer.java
      TestCharge.java
      TestDegreeCareer.java
      TestEmployee.java
      TestIntegracion.java
      TestLab.java
      TestLabMember.java
      TestPerson.java
      TestPlan.java
      TestPostDegreeCareer.java
      TestStudent.java
      TestSubject.java
      TestTeacher.java
      TestTitle.java
      Title.java
      University.ocl
```

Figure 6. Created files after executing the Acceleo code



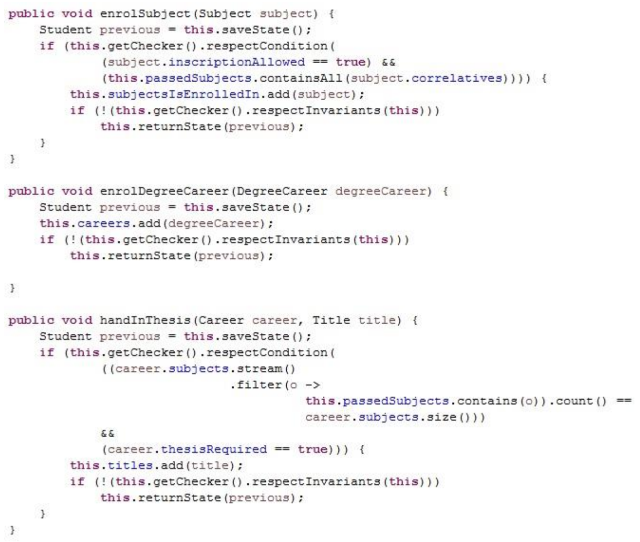
```java
package results;

import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
import junit.framework.TestCase;

@RunWith(Suite.class)
@SuiteClasses({ TestCharge.class,TestEmployee.class,TestDegreeCareer.clas
    TestTeacher.class,TestCareer.class,TestPerson.class,TestStudent.class
    TestArea.class,TestLabMember.class,TestTitle.class,TestLab.class,
    TestSubject.class,TestPostDegreeCareer.class,TestPlan.class})
public class IntegrationTest extends TestCase{}
```

Figure 7. Integration Test code and its execution result in JUnit



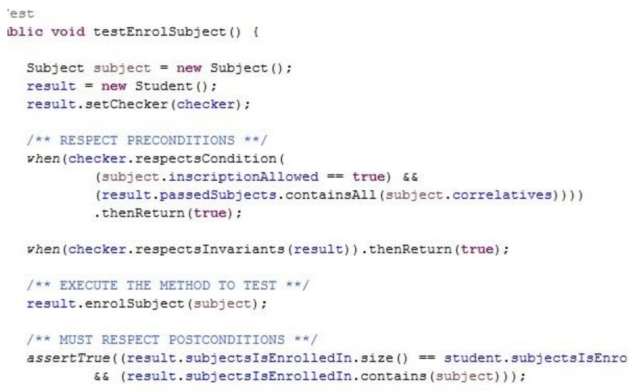
```java
import java.util.ArrayList;

public class Student extends Person {

    protected ArrayList<Career> careers;
    protected ArrayList<Subject> passedSubjects;
    protected String studentNumber;
    protected ArrayList<Subject> subjectsIsEnrolledIn;
    protected ArrayList<Title> titles;

    public class StudentChecker extends PersonChecker {
        public StudentChecker() {
        }

        public boolean respectsInvariants(Person studentIn) {
            Student student = (Student) studentIn;
            return ((student.careers.size() <= 1) &&
                    (student.personName.length() <= 40) &&
                    (student.age >= 18));
        }

        public boolean respectsCondition(boolean condition) {
            return condition;
        }
    }
```

Figure 8. Student class and its internal checker



```java
public void enrolSubject(Subject subject) {
    Student previous = this.saveState();
    if (this.getChecker().respectCondition(
            (subject.inscriptionAllowed == true) &&
            (this.passedSubjects.containsAll(subject.correlatives)))) {
        this.subjectsIsEnrolledIn.add(subject);
        if (!(this.getChecker().respectInvariants(this)))
            this.returnState(previous);
    }
}

public void enrolDegreeCareer(DegreeCareer degreeCareer) {
    Student previous = this.saveState();
    this.careers.add(degreeCareer);
    if (!(this.getChecker().respectInvariants(this)))
        this.returnState(previous);

}

public void handInThesis(Career career, Title title) {
    Student previous = this.saveState();
    if (this.getChecker().respectCondition(
            ((career.subjects.stream()
                            .filter(o ->
                                    this.passedSubjects.contains(o)).count() ==
                                    career.subjects.size()))
            &&
            (career.thesisRequired == true))) {
        this.titles.add(title);
        if (!(this.getChecker().respectInvariants(this)))
            this.returnState(previous);
    }
}
```

Figure 9. Student class generated methods



```
'est
iblic void testEnrolSubject() {

   Subject subject = new Subject();
   result = new Student();
   result.setChecker(checker);

   /** RESPECT PRECONDITIONS **/
   when(checker.respectsCondition(
           (subject.inscriptionAllowed == true) &&
           (result.passedSubjects.containsAll(subject.correlatives))))
           .thenReturn(true);

   when(checker.respectsInvariants(result)).thenReturn(true);

   /** EXECUTE THE METHOD TO TEST **/
   result.enrolSubject(subject);

   /** MUST RESPECT POSTCONDITIONS **/
   assertTrue((result.subjectsIsEnrolledIn.size() == student.subjectsIsEnro
           && (result.subjectsIsEnrolledIn.contains(subject)));
```

Figure 10. Example test: Student class method

3. Improving Tests with a Richer Formalism

The translation described above allows us to automatically obtain the code of the test cases. These tests will be executed dynamically.

At the same time we will offer another level of verification, in order to improve the coverage of the tests.

We will use the formal language Alloy [5] that allows the static verification of models.

Alloy is a formal modeling language, with formal syntax and semantics, based on first-order relational logic. Its main target is the formal specification of object-oriented models. At a glance, Alloy is similar to UML class diagrams and OCL, but having simpler and cleaner semantics, and being also supported by a rich verification tool named Alloy Analyzer [12]. The Alloy Analyzer analyzes model properties automatically. It applies a delimited verification, limiting the number of objects in each class to a fixed number and checking assertions over the specification within that limit. It uses a SAT-solver to answer verification queries, converting them to logic Boolean formulas.

3.1. Translating from OCL to Alloy

As described in previous section, our translation generates the Java code plus an .ocl file with every OCL constraint that appeared on the source UML model (see figure 11). Since Java handles OCL in its libraries, using EMF let us check the model consistency at an OCL level.

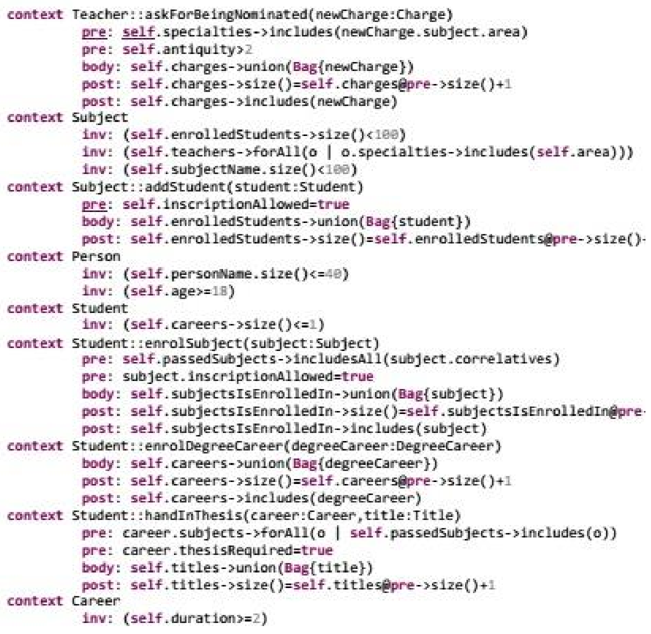
Then, we will use the AlloyMDA[11] tool to translate the OCL code we have generated to its correspondent Alloy code, from which we will be able to use the Alloy Analyzer to check its consistency.

In our case study, by executing the following command:

$runghc OCL2Alloy < University.ocl University.uml

We obtain the Alloy code, printed by console, as we can see in figure 12. We must copy this code and paste it in an .als file called *University.als*. In the Alloy code, the expression *sig*, abbreviation for *signature*, represents a set of objects (similar to a Java class). These signatures may or not have a set of attributes. For example, the class Career has the expression *some Subject*, where *some* means "at least one" (there are other expressions such as l*one* or *at most one*, *one* or *exactly one*, etc.). Another relevant expression in the code is *subjectsIsEnrolledIn* : *Subject* some ->Time. This is translated as a set of subjects in which the student is enrolled at a certain moment. Time appears here since the collection can be modified by some method, having to access it in its different states over the variable time.

After defining the signatures, another kind of expressions are introduced, which are headed by the key word *pred* (abbreviation for predicate). They represent the definition of properties, returning the analyzer *true* or *false* if it can find instances that satisfy the predicate or not. It is a way of verifying that our original methods are executed successfully.



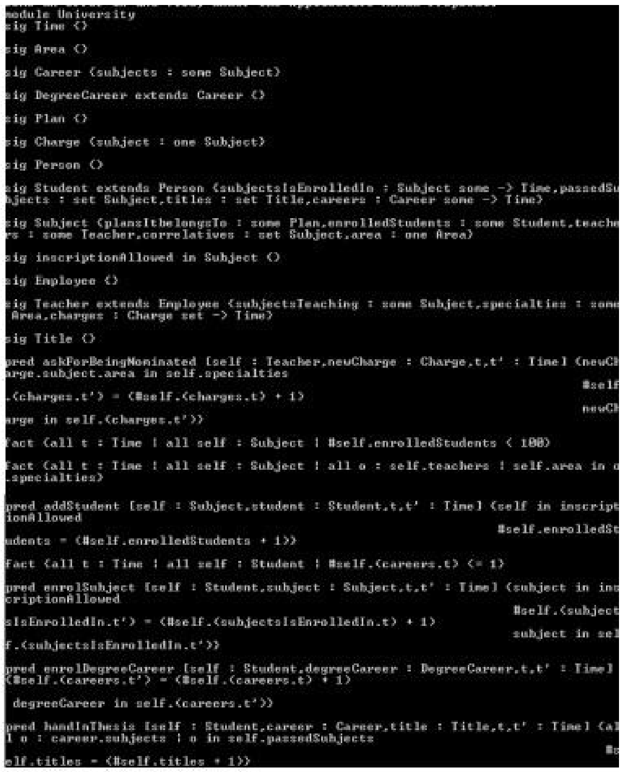
```
context Teacher::askForBeingNominated(newCharge:Charge)
        pre: self.specialties->includes(newCharge.subject.area)
        pre: self.antiquity>2
        body: self.charges->union(Bag{newCharge})
        post: self.charges->size()=self.charges@pre->size()+1
        post: self.charges->includes(newCharge)
context Subject
        inv: (self.enrolledStudents->size()<100)
        inv: (self.teachers->forAll(o | o.specialties->includes(self.area)))
        inv: (self.subjectName.size()<100)
context Subject::addStudent(student:Student)
        pre: self.inscriptionAllowed=true
        body: self.enrolledStudents->union(Bag{student})
        post: self.enrolledStudents->size()=self.enrolledStudents@pre->size()-
context Person
        inv: (self.personName.size()<=40)
        inv: (self.age>=18)
context Student
        inv: (self.careers->size()<=1)
context Student::enrolSubject(subject:Subject)
        pre: self.passedSubjects->includesAll(subject.correlatives)
        pre: subject.inscriptionAllowed=true
        body: self.subjectsIsEnrolledIn->union(Bag{subject})
        post: self.subjectsIsEnrolledIn->size()=self.subjectsIsEnrolledIn@pre-
        post: self.subjectsIsEnrolledIn->includes(subject)
context Student::enrolDegreeCareer(degreeCareer:DegreeCareer)
        body: self.careers->union(Bag{degreeCareer})
        post: self.careers->size()=self.careers@pre->size()+1
        post: self.careers->includes(degreeCareer)
context Student::handInThesis(career:Career,title:Title)
        pre: career.subjects->forAll(o | self.passedSubjects->includes(o))
        pre: career.thesisRequired=true
        body: self.titles->union(Bag{title})
        post: self.titles->size()=self.titles@pre->size()+1
context Career
        inv: (self.duration>=2)
```

Figure 11. Generated OCL Centralized Code

Then, we can see the key word *fact*, which represents a restriction assumed to always hold (in other words, an invariant). For example, we see the following fact expressing that a subject can never have more than 99 students,

fact *{all t:Time | all self:Subject | #self.enrolledStudents <100}*

The expression *all* t:Time gives the fact its invariant character, holding over the time.



```
module University
sig Time {}

sig Area {}

sig Career {subjects : some Subject}

sig DegreeCareer extends Career {}

sig Plan {}

sig Charge {subject : one Subject}

sig Person {}

sig Student extends Person {subjectsIsEnrolledIn : Subject some -> Time,passedSu
bjects : set Subject,titles : set Title,careers : Career some -> Time}

sig Subject {plansItbelongsTo : some Plan,enrolledStudents : some Student,teache
rs : some Teacher,correlatives : set Subject,area : one Area}

sig inscriptionAllowed in Subject {}

sig Employee {}

sig Teacher extends Employee {subjectsTeaching : some Subject,specialties : some
 Area,charges : Charge set -> Time}

sig Title {}

pred askForBeingNominated [self : Teacher,newCharge : Charge,t,t' : Time] {newCh
arge.subject.area in self.specialties
                                                                          #self
.(charges.t') = (#self.(charges.t) + 1)
                                                                          newCh
arge in self.(charges.t')}

fact {all t : Time | all self : Subject | #self.enrolledStudents < 100}

fact {all t : Time | all self : Subject | all o : self.teachers | self.area in o
.specialties}

pred addStudent [self : Subject,student : Student,t,t' : Time] {self in inscript
ionAllowed
                                                               #self.enrolledSt
udents = (#self.enrolledStudents + 1)}

fact {all t : Time | all self : Student | #self.(careers.t) <= 1}

pred enrolSubject [self : Student,subject : Subject,t,t' : Time] {subject in ins
criptionAllowed
                                                                  #self.(subject
sIsEnrolledIn.t') = (#self.(subjectsIsEnrolledIn.t) + 1)
                                                                  subject in sel
f.(subjectsIsEnrolledIn.t')}

pred enrolDegreeCareer [self : Student,degreeCareer : DegreeCareer,t,t' : Time]
(#self.(careers.t') = (#self.(careers.t) + 1)

 degreeCareer in self.(careers.t')}

pred handInThesis [self : Student,career : Career,title : Title,t,t' : Time] {al
l o : career.subjects | o in self.passedSubjects
                                                                             #s
elf.titles = (#self.titles + 1)}
```

Figure 12. Alloy code generated by AlloyMDA

3.2. Using the Alloy Analyzer

After generating the Alloy code, we can take advantage of its formal analyzer to verify the source UML/OCL model. This analyzer was developed to support lightweight formal methods. As such, its main objective is to provide a complete automatic analysis, unlike the theorem testing techniques usually used in similar specification languages. It works through a reduction to SAT, using first order logic to translate Alloy specifications to very long boolean expressions that can be automatically analyzed by a SAT solver (explaining why from an Alloy logic expression, its analyzer can try to find a satisfying model).

Clearly, the best feature of this tool is finding at least one model which does not satisfy it, revealing the presence of errors. The analyzer can be freely downloaded from [5] in .jar format, being portable and its main screen is displayed in figure 13.

Alloy Analyzer 4.2
File Edit Execute Options Window Help
New Open Reload Save Execute Show

Alloy Analyzer 4.2 (build date: 2012-09-25 15:54 EDT)

Figure 13. Alloy Analyzer main screen

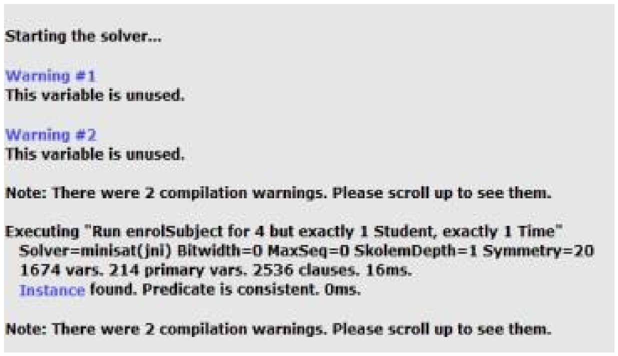
After opening our .als file, in order to run it, we must specify with the special command run the predicates to run with their scope (setting boundaries). The errors we might find or not will occur inside this scope, being possible to have more/other errors outside.

That is to say, if an example is found, the predicate can be satisfied. On the other hand, if no examples are found, the predicate will be invalid (false for every possible example), or maybe valid but outside the specified scope. We now specify our command to execute the .als file:

run enrolSubject for 4 but exactly 1 Student, exactly 1 Time

In this case, we test the predicate *enrolSubject* with a scope that will limit our search to those instances that have at least 4 instances of each signature, except from Student, which will have just one object. Also, for the sake of simplicity we execute it for just one time instance.

Figure 14 displays the messages returned by the tool console after running the Alloy analyzer. Messages include some irrelevant warnings, the analyzer configuration data, if some instances were found or not, the time it took to execute the analysis and its verdict. In this example the analyzer reported that the model is consistent and let us visualize the generated instance (figure 15).



```
Starting the solver...

Warning #1
This variable is unused.

Warning #2
This variable is unused.

Note: There were 2 compilation warnings. Please scroll up to see them.

Executing "Run enrolSubject for 4 but exactly 1 Student, exactly 1 Time"
   Solver=minisat(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1674 vars. 214 primary vars. 2536 clauses. 16ms.
   Instance found. Predicate is consistent. 0ms.

Note: There were 2 compilation warnings. Please scroll up to see them.
```

Figure 14. Alloy Analyzer results

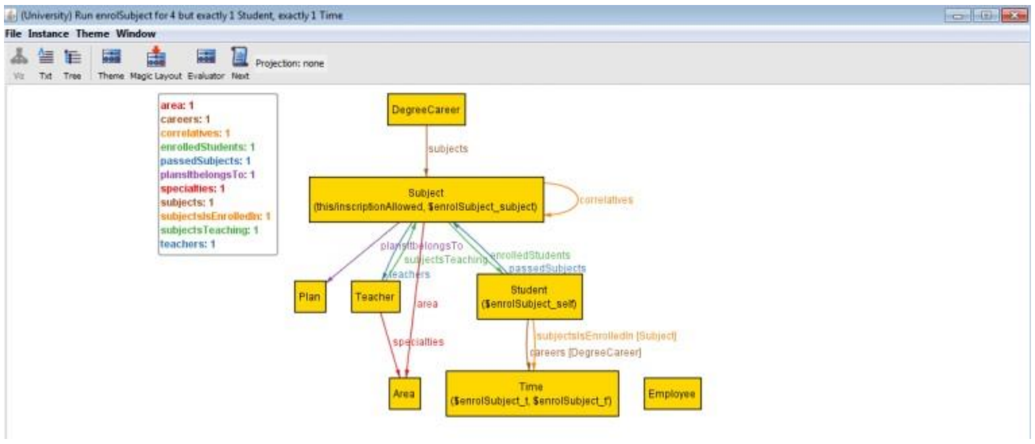


Figure 15. model instance found by the analyzer

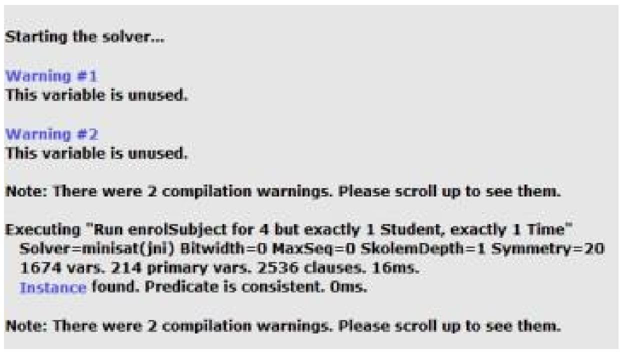
To exemplify what happen when the analyzer deals with an inconsistent model, we add an inconsistency in the case study. Knowing that we have the following constraint in the class *Subject*,

self.enrolledStudents->size() < 100

We will add a new constraint to the same class,

self.enrolledStudents ->size() > 100

Straightforwardly we can see that the model is going to be inconsistent, since there is no subject that can have less than 100 students and at the same time more than 100. Nevertheless, our Java code will be generated as before and its test will still be successful (since we will use mock objects that will always/never pass their invariants). But after translating the OCL code to its correspondent Alloy code, the situation will be different. If we execute the Alloy analyzer, we will get the answer in figure 16, without finding model instances, and warning us that our model might be inconsistent.



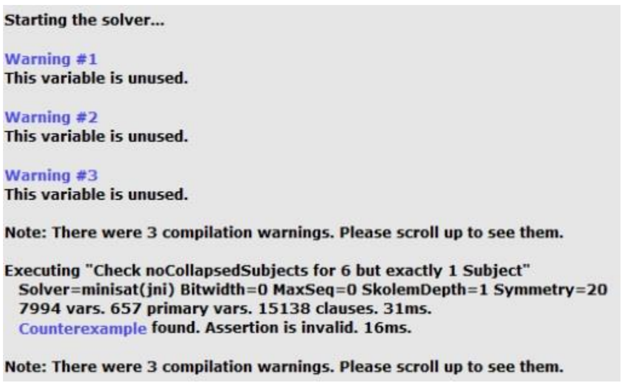
```
Starting the solver...

Warning #1
This variable is unused.

Warning #2
This variable is unused.

Note: There were 2 compilation warnings. Please scroll up to see them.

Executing "Run enrolSubject for 4 but exactly 1 Student, exactly 1 Time"
   Solver=minisat(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   1674 vars. 214 primary vars. 2536 clauses. 16ms.
   Instance found. Predicate is consistent. 0ms.

Note: There were 2 compilation warnings. Please scroll up to see them.
```

Figure 16. No model instances to show

To find concrete evidence of the violation of the model specification, we can use the Alloy command *check* which, given an assertion, looks for counterexamples that let us observe how certain facts are violated. In this case, the facts will be the two previously mentioned invariants, and the assertion will be created with the name *noCollapsedSubjects*, specifying that no subject will count with more than 100 students:

***assert** noCollapsedSubjects { no s:Subject —*
#s.enrolledStudents < 100}

Now we must execute the check command, invoking the assertion: check noCollapsedSubjects for 101 but exactly 1 Subjec.t After executing the analyzer (for this example we used 5 students instead of 100 to have a better response time), we get the answer showed in figure 17, having found a counterexample. Then we can visualize it as shown in figure 18.



```
Starting the solver...

Warning #1
This variable is unused.

Warning #2
This variable is unused.

Warning #3
This variable is unused.

Note: There were 3 compilation warnings. Please scroll up to see them.

Executing "Check noCollapsedSubjects for 6 but exactly 1 Subject"
   Solver=minisat(jni) Bitwidth=0 MaxSeq=0 SkolemDepth=1 Symmetry=20
   7994 vars. 657 primary vars. 15138 clauses. 31ms.
   Counterexample found. Assertion is invalid. 16ms.

Note: There were 3 compilation warnings. Please scroll up to see them.
```

Figure 17. The analyzer has found a counterexample

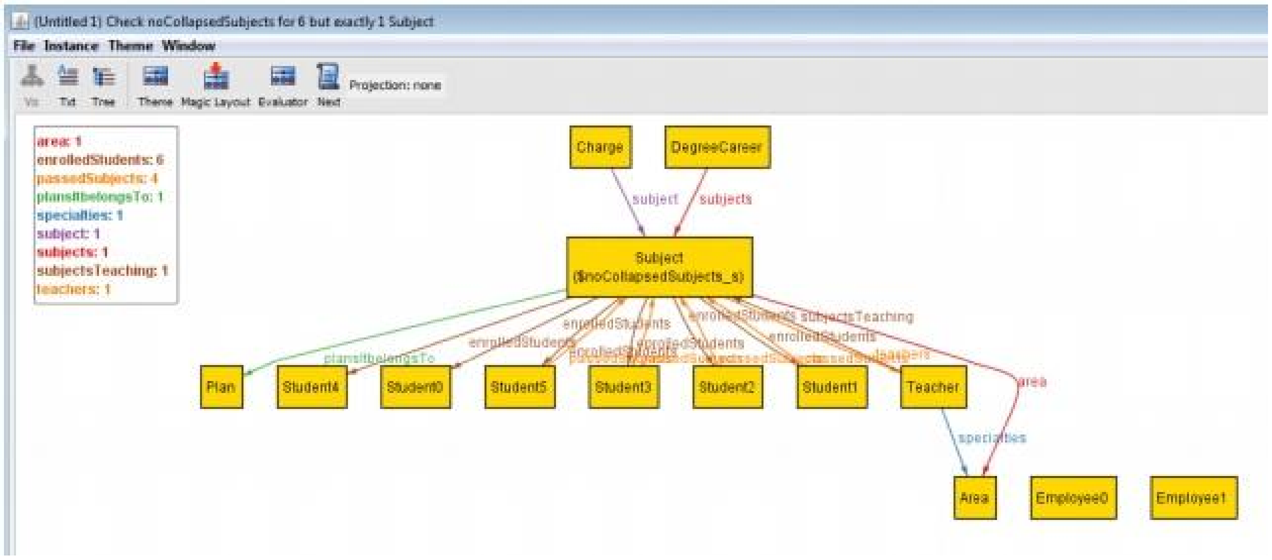


Figure 18. Generated counterexample

4. Related Work

Several tools provide support for automatic test code generation from software models. We summarize here the most relevant ones:

TestEra[12] is a testing framework based on Java programs specification. To test a method, it uses the specification of the methods pre conditions to generate tests inputs and the post conditions to check the output correctness. This framework introduced the black box systematic testing in Java programs using Alloy as a technology that allows a limited and exhaustive testing, where a program is tested against every non-equivalent entry within a specific input space. It was very effective when trying to find bugs in several applications which take structured tests in a complex way.

UML base model and OCL verification: Modeling languages such as UML and OCL are increasingly used in early stages of the system design, offering a huge set of constructions. As a result, existent verification engines just support a small part of them. In [13] a new approach is proposed, using model transformations to unify different descriptions meanings in a base model. Along the transformation, constructions are expressed in a complex language with a small group of what are called core elements. This simplification allows interacting with a wide range of verification engines with different advantages and weaknesses.

Model-based tests generation for web applications: In [14] a tool to filter/setup test code within the project is introduced, based on PGBT models. These models are written in a DSL called PARADIGM and consist in UI test patterns (UITP), describing the test objectives. To generate test cases code, the tester must provide test input data to each UITP model. Nevertheless, without a test case generation algorithm filter/configuration, the test cases quantity may be so big that it turns unmanageable. So, the approach in [14] introduces a technique to define test cases code generation parameters to generate a reasonable number of them, comparing the different test strategies and measuring the model tool performance against a capture-replay tool which is used for web testing.

UML modeling environments for tests creation are usually uncomfortable and force users to know many UML details. The Fokus!MBT tool [15] is a multiparadigm test modeling environment based on the UML testing profile and an industry notation adopted by the OMG for model-based testing. Fokus!MBT simplifies the creation and authorship of test models with a specific methodology support.

5. Conclusion and Future Works

We developed a tool which allows translating a data model with formal constraints to its corresponding Java code, automating the generation of strong test cases codes and specifying them not only in the Java language but also in two formal languages, such as OCL and Alloy. In a few steps a regular UML and Java user with some OCL knowledge can define a data model and count with the needed tools to verify whether that model is consistent and to automatically generate the system code with associated test-cases code. This gives developers a trustworthy and verifiable support with different techniques.

Comparing it with the related works we described before, we can point the following advantages:

- ✓Dual verification: we achieve both static and dynamic verification.
- ✓UML-Alloy connection: generally, the proposed tools associate UML/OCL with MDT or OCL with Alloy. In this case, we consistently integrate the three of them.
- ✓Better Tools: we made use of stronger and newer tools such as Acceleo, Papyrus and Mockito against MOFScript and EasyMock.
- ✓Complete process: generally, only one part of the software development process is optimized/automated. In this case, we provide a code ready for production and which is verifiable, adaptable and usable for a wide range of users.

To extend the proposed solution, we are working on the following lines:

- ✓After modifying the code we got in the first instance and also modifying the original model, regenerate the code with the Acceleo tool without altering the updates we have made or the text which was delimited by special markers.
- ✓Have less abstract tests and try not to use mocks, in order to generate more specific tests and more related with each method to make them more trustworthy.
- ✓When finding an inconsistence in the source model, generate counterexamples in the natural/Java language, so that users who do not understand formal verification can understand and help to fix them.
- ✓Allowing the developer to select the programming language for the generated code (additionally to Java).

6. References

[1] Claudia Pons, Roxana Giandini, Gabriela Perez. Desarrollo de Software Dirigdo por Modelos. Universidad Nacional de La Plata. Editorial: McGraw-Hill Educacion y Edulp. Marzo 2010

[2] Mark Utting and Bruno Legeard. Practical Model Based Testing: A tools approach. 2007

[3] Unified Modeling Language^{TM} (UML) http://www.omg.org/spec/UML/

[4] Mandana Vaziri and Daniel Jackson. Some Shortcomings of OCL, the Object Constraint Languag e of UML. MIT Laboratory for Computer science . December, 1999

[5] Alloy: http://alloy.mit.edu/alloy/

[6] Eclipse Modeling Framework EMF: http://eclipse.org/modeling/emf/

[7] Papyrus: http://eclipse.org/papyrus

[8] Acceleo: http://wiki.eclipse.org/Acceleo

[9] JUnit: http://junit.org/junit4/ [10 Mockito: http://site.mockito.org/

[11] AlloyMDA: http://sourceforge.net/p/alloymda/wiki/Home/

[12] Shadi Abdul Khalek, Guowei Yank, Lingmin g Zhang, Darko Marinovt, Sarfraz Khurshid. TestEra: A tool for testing Java Programs using Alloy speci cations. Electrical and Computer Engineering , University of Texas at Austin.

[13] Frank Hilken, Philipp Niemann, Robert Wille and Martin Gogolla. Towards a Base Model for UML and OCL Veri cation. University of Bremen, Computer Science Department

[14] Miguel Nabuco, Ana C.R. Paiva. Model-based test case generation for Web Applications. Department of Informatics Engineering. Faculty of Engineering of University of Porto

[15] Marc-Florian Wendland, Andreas Homann, Multi-Paradigmatic Test Modeling Environment. Fraunhofer, Ina Schieferdecker. Fokus!MBT – A multiparadigm test modeling environment.