Improving Model-Driven Software Testing by using Formal Languages

Ilan Rosenfeld [1] and Claudia Pons[1,2,3] and Gabriel Baum [1]

[1] Facultad de Informática, Universidad Nacional de La Plata
[2]Comision de Investigaciones Científicas CIC
La Plata, Buenos Aires, Argentina.
[3] Universidad Abierta Interamericana, UAI
Ciudad de Buenos Aires, Argentina.
Ilan.Rosenfeld@gmail.com, cpons@info.unlp.edu.ar, gbaum@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 a software development. It consists in a black box test that uses structural and behavioral models to automate the tests generation process. In this paper, we describe a tool that allows developers to translate a software model written in UML with OCL 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, which are OCL and Alloy. This tool provides more reliable support by amalgamating different techniques, which strengthens the testing process.
Keywords: model driven testing, UML, OCL, Java, testing, formal languages.

Abstrato. O Model-Driven Testing ou MDT é uma nova e promissora abordagem para automação de testes de software que pode reduzir significativamente os esforços no ciclo de testes de um desenvolvimento de software. Consiste em um teste de caixa preta que utiliza modelos estruturais e comportamentais para automatizar o processo de geração de testes. Neste artigo, descrevemos uma ferramenta que permite aos desenvolvedores traduzir um modelo de software escrito em UML com restrições formais OCL para seu código Java correspondente, automatizando a geração de códigos de casos de teste e especificando-os não apenas na linguagem Java, mas também em dois linguagens formais, que são OCL e Alloy. Esta ferramenta fornece suporte mais confiável, combinando diferentes técnicas, o que fortalece o processo de teste.

Palavras-chave: teste orientado por modelo, UML, OCL, Java, teste, linguagens formais.

Introduction

The Model-Driven Software Development Process (MDD) [Brambilla et al 2012] [Stahl and Voelter 2006] is a discipline that is generating a lot of expectations as an alternative to conventional methods of software production. MDD set out a new way of understanding development and maintenance of software systems by using models as main artifacts in the development process. In MDD, the models are used to direct tasks related to comprehension, design, construction, tests, deployment, operation, management, maintenance and modification of systems. A great number of theoretical and practical studies are involved in this approach. Moreover, experiences surveyed by´[Di Ruscio, et al 2014] and by the Object Management Group [OMG 2015] reported on existing tools that make this approach real at a commercial level, with several examples of successful introduction of MDD in different organizations

The success of any MDD project heavily depends on the quality of the source models that should be accurate, consistent and complete. The Unified Modeling Language [UML 2017] is a general-purpose modeling language that is intended to provide a standard way to visualize the design of a system. The creation of UML was motivated by the desire to standardize the heterogeneous notational systems and approaches to software design. The UML was adopted as a standard by the Object Management Group (OMG). On the other hand, the Object Constraint Language [OCL 2017] is a textual language with formal foundation, based on 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 modeling language.

The ultimate goal of MDD is to generate software automatically from the models, so that the target software is correct by construction. However, this dream has not been achieved yet since the generated code must usually be completed by hand, which introduces errors. Thus the testing cycle cannot be ignored as a substantial part of the software development process.

In this regard, one of the branches of MDD is the Model-Driven Testing (MDT) [Utting and Legeard 2007], 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 and test-cases data sets.

There exist a significant number of tools that generate code from software models, but few of them take full advantage of what formal modeling languages offer for automation of the testing cycle. For this reason, the construction of a new software tool to automate the generation of test-cases code was developed using the formal foundation of the modeling notations, in order to obtain better benefits.

This tool, named MDT+, allows developers to automatically generate Java code from UML/OCL models, including both the system classes and their test-cases code. The generated test-cases code is written in Java and it is executable. Additionally, test code is enhanced with formal specifications which allow the application of model checking techniques as a complement to testing. In this way MDT+ combines static and dynamic formal analysis of the system, improving the efficacy of the analysis process.

The rest of the paper is organized as follows. Section 2 explains the technological background. Section 3 describes the basic features of MDT+. Section 4 presents an extension of the tool which improves the tests through the application of a richer formalism. Section 5 discusses a set of related works. Finally, conclusions are presented in section 6.

Eclipse Modeling Tools

MDT+ was developed taking advantage of a number of existing tools, in particular the Eclipse Modeling Project [EMP 2017] 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. In this section, the main Eclipse elements that were included in the development are briefly described.

Eclipse Modeling Framework (EMF).

The Eclipse Modeling Framework [EMF 2017] includes a set of plugins that can be used to specify a data model and generate code or other kind of output based on that model.

Papyrus.

Papyrus (2017) is a subproject component that aims to provide an integrated and usable environment to edit any type of EMF model. Papyrus provides diagrams editors for EMF-based modeling languages such as UML2 and offers the chance of integrating these editors with other tools. It also offers an advanced support for profiles, allowing the user to define standard UML2-based Domain Specific Language (DSL) editors and their extension mechanisms.

Eclipse Acceleo.

Eclipse Acceleo (2017) is an open source code generator implementing the OMG's MOF Model to Text Language (MTL) standard that uses any EMF-based models (e.g., UML, SysML, domain specific models, etc.) to generate any kind of code (e.g.,Java, C, PHP, etc.).

MDT+. A Tool for Test-cases Code Generation

In this section, we describe the characteristics of MDT+, the software tool that was built to automate the generation of test-cases code. Starting from an OCL/UML system model, the Java code is automatically generated, creating the classes with their corresponding test-cases code and an OCL file which contain all the formal constraints in a centralized form. The process is carried out in three steps, as described below following a running example.

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, which allows the visualization of 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.

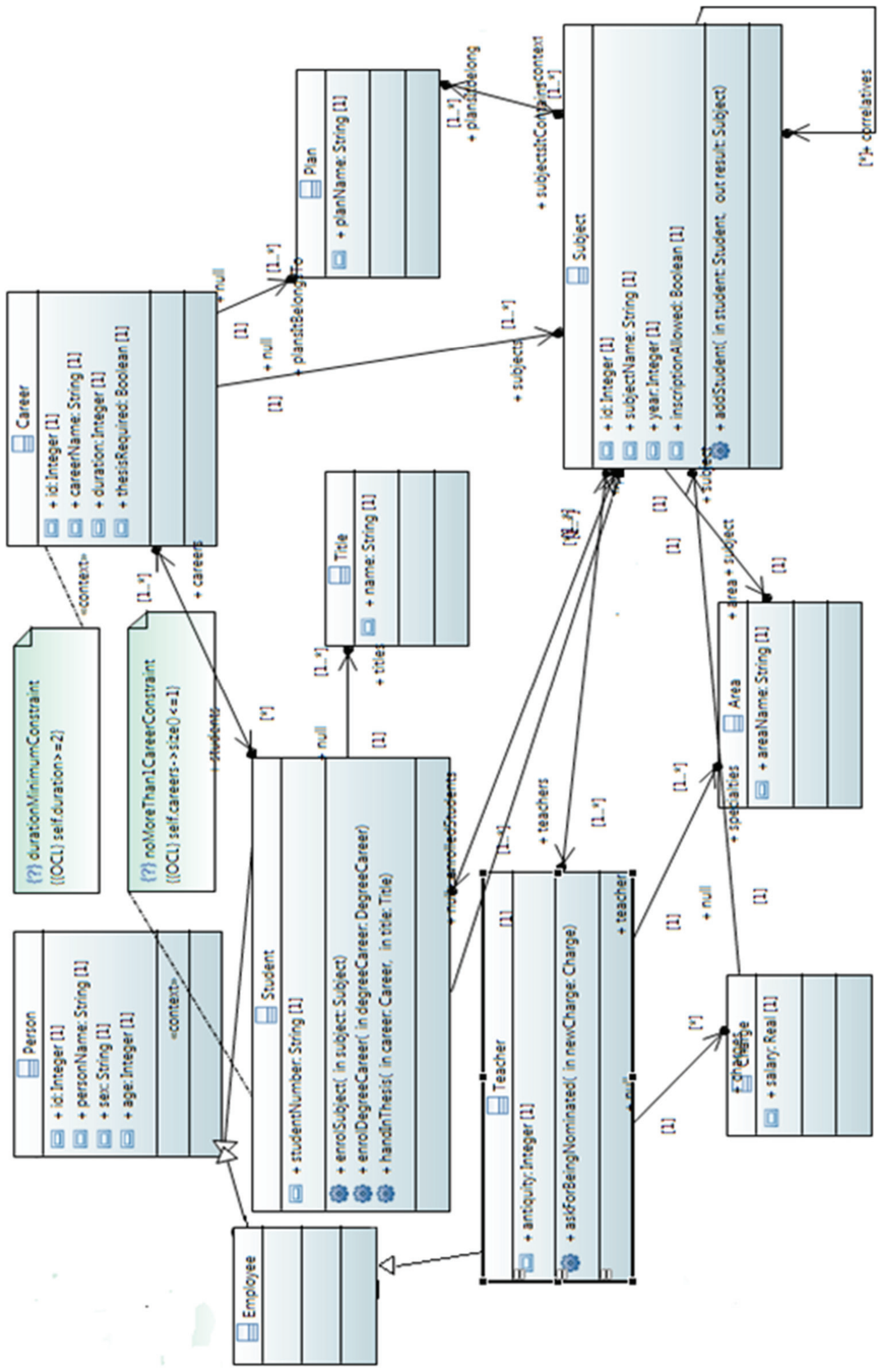


Fig. 1. Class diagram.

The diagram also includes a set of OCL restrictions (the palette Constraint elements) representing invariants associated with specific classes. For example, students are not allowed to be enrolled in more than one career, and in order to teach a subject, teachers must be experts on their area, which is reflected in the following OCL invariants,

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

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

Additionally, the operation pre and post conditions can be specified using OCL. For example, the following OCL expressions state that in order to enroll in a subject a student must have already passed all its correlatives and the subject inscription is enabled,

```
Context Student::enrolSubject(subject)
pre:self.passedSubjects->includesAll(subject.correlatives)
pre:subject.inscriptionAllowed=true
```

Besides, a set of post conditions for the operation can be specified. The first one checks that the specified subject has been actually added to the collection and the second one specifies that the collection size is incremented in one after executing the method,

```
context Student::enrolSubject(subject)
  post: self.subjectsIsEnrolledIn-> includes(subject)
  post: self.subjectsIsEnrolledIn->size() =
  self.subjectsIsEnrolledIn@pre->size()+1
```

A correct implementation code should hold all the invariants, pre and post conditions defined in the model. Consequently, the test cases will check that those constraints hold when executing the methods.
MDT+ also allows developers to define the body of each class method in a different range of languages and formats. In the case study of this paper methods specifications are defined in OCL, since this format is quite similar to the Java syntax, its later translation (from the model class into the Java file) is straightforward.

Translating the UML model to Java code with Tests

MDT+ includes the following components in order to translate the UML/OCL model to executable Java code equipped with tests:

- ✓ Two java classes, Activator.java and Generate.java, which are configuration files, specifying the included libraries among other things.

- ✓ An Acceleo module called generate.mtl which contains the translation algorithm (from UML model to java code), written in the Acceleo language.

First of all the MDT+ user chooses the UML model from which generate the corresponding classes, for example, the UML model displayed in figure 1 can be used as the source model in the Acceleo configuration.

Then, the Acceleo algorithm loops over every class of the UML source model, and for each one it creates two artifacts: a regular .java class and a checker class for testing purposes. Also, the algorithm creates the integration test, which runs every individual generated test in a single step.

In parallel, the file *University.ocl* is created, containing every modeled OCL constraint associated to its context, in a centralized way.

Each internal checker class consists of two methods, *respectInvariants*(classInstance) and *respectCondition*(condition), to chek invariants, pre and post conditions respectively.

The class constructor checks through the checker that any new instance respects its invariants. Then, the checker is invoked any time the instance is updated.

Generated getters are regular getters, returning the desired attribute.

On the other hand, setters follow this process:

1) Save the current instance state through the *saveState* generated method;
2) Set the attribute value to the input value;
3) Check if the instance still respects its invariants. If not, return the instance with its previous status, using the *returnState* generated method.

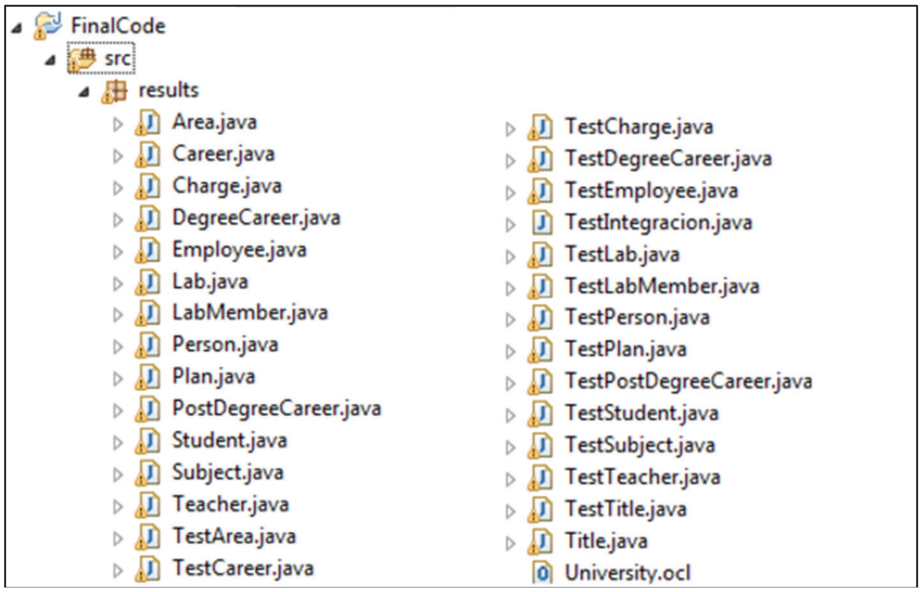
When defining each class method, a copy of the object is generated with the nickname “previous”. Then, the method pre conditions are checked. If they fail, the method execution terminates without modifying the instance. If they succeed, the method is executed and then the instance invariants are checked; then if invariants do not hold, the instance is returned to its previous status using the created copy, having the method no effect on the instance.

Additionally, the tests generated by MDT+ extend from the special class *TestCase* in order to apply the JUnit library [JUnit 2017]. MDT+ associates a simulated object (i.e., a mock object) using the Mockito library [Mockito 2017]. This tool attaches a specific behavior to the class instances in order to verify for each method that, if the pre conditions and invariants hold, the post conditions hold as well.

Analyzing the results

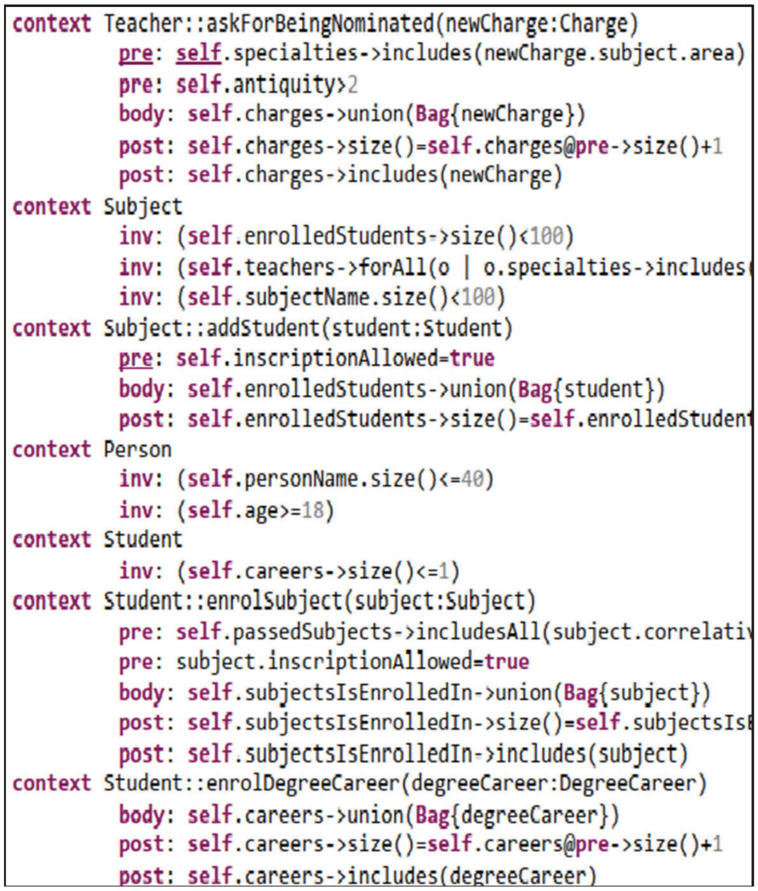
After executing the *generate.mtl* file, the corresponding .java classes and the *.ocl* file are generated (see figure 2 and figure 3 respectively)). Method bodies specifications written in OCL are translated to its corresponding Java code.

Integration test can be run in order to check in a single step that every generated test is satisfied, as shown in figure 4. Regarding the generated code for each class, a part of the Student class code is displayed in figures 5 and 6.



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

Fig. 2. Files generated by the Acceleo code running.



```
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(
        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.enrolledStudent
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.correlati
        pre: subject.inscriptionAllowed=true
        body: self.subjectsIsEnrolledIn->union(Bag{subject})
        post: self.subjectsIsEnrolledIn->size()=self.subjectsIsE
        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)
```

Fig. 3. Generated OCL Centralized Code.

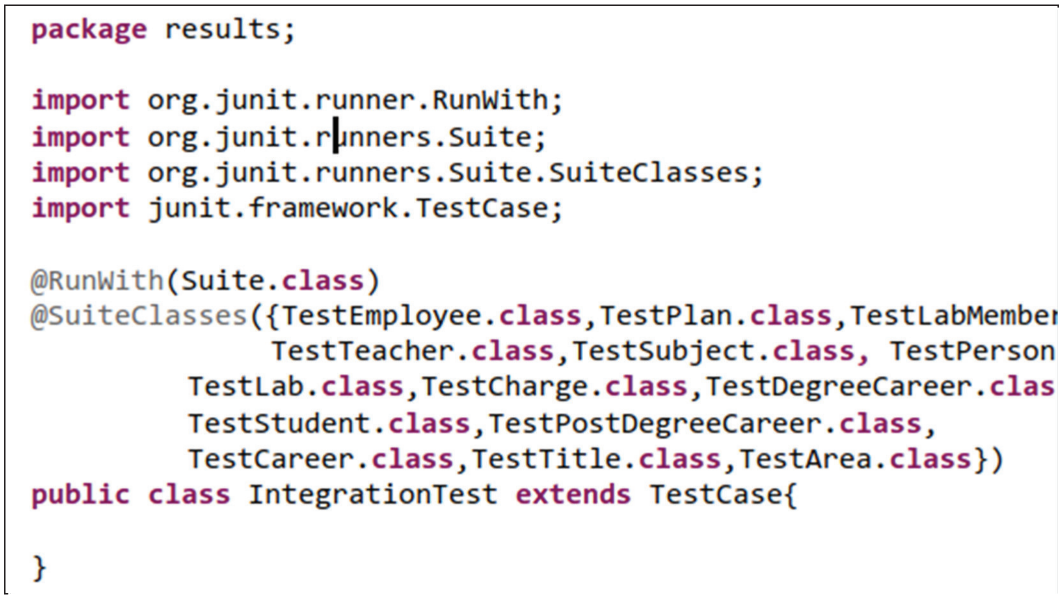

```
package results;

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

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

}
```

Fig. 4. Integration Test code and its execution result in JUnit.

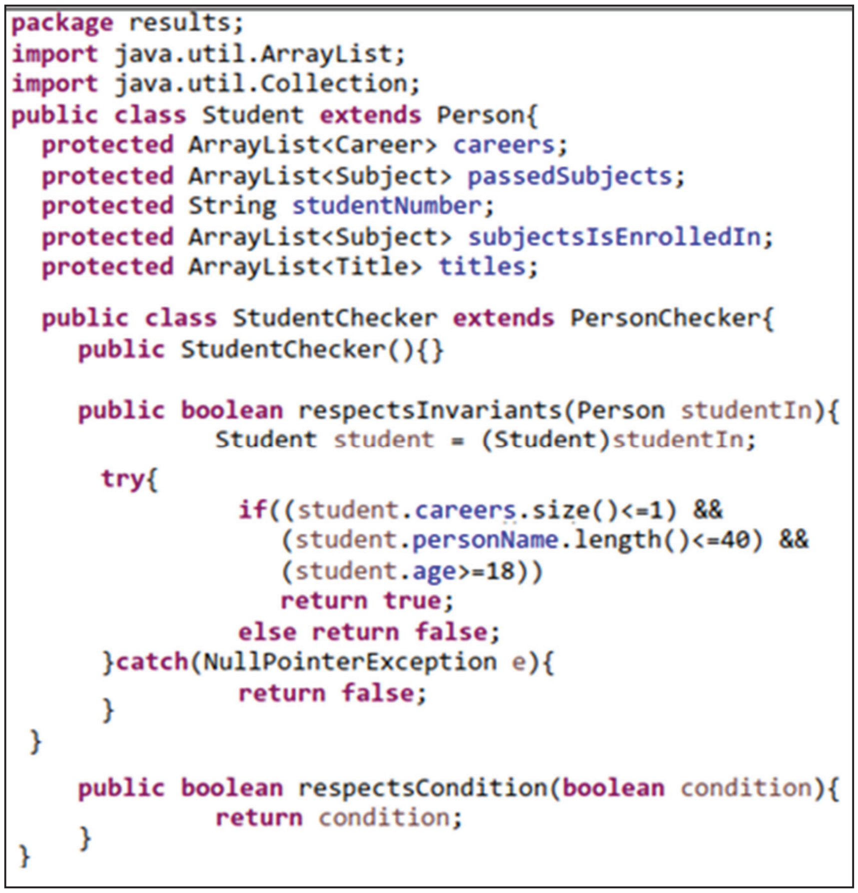
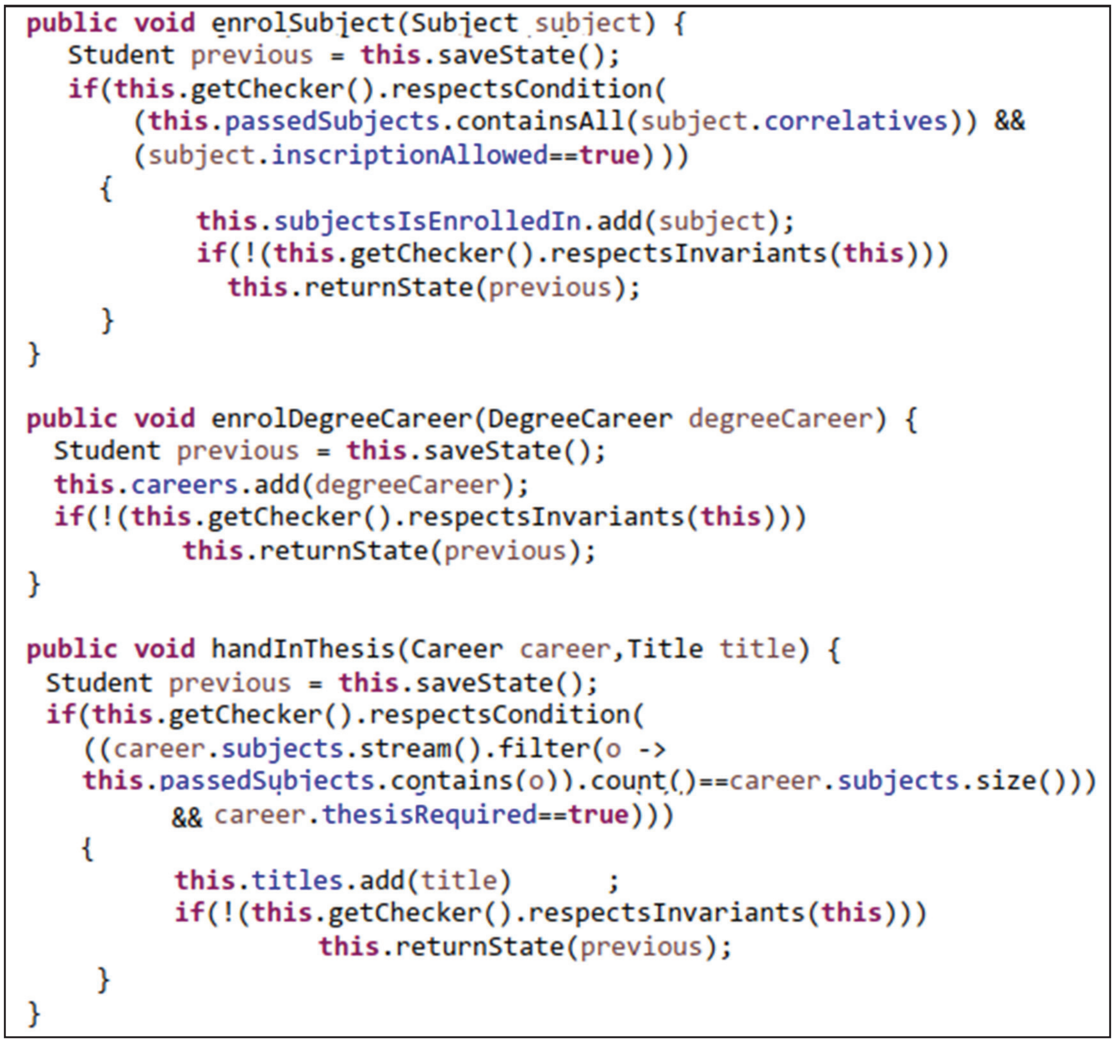

```
package results;
import java.util.ArrayList;
import java.util.Collection;
public class Student extends Person{
  protected ArrayList<Career> careers;
  protected ArrayList<Subject> passedSubjects;
  protected String studentNumber;
  protected ArrayList<Subject> subjectsIsEnrolledIn;
  protected ArrayList<Title> titles;

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

    public boolean respectsInvariants(Person studentIn){
            Student student = (Student)studentIn;
     try{
            if((student.careers.size()<=1) &&
               (student.personName.length()<=40) &&
               (student.age>=18))
               return true;
            else return false;
     }catch(NullPointerException e){
            return false;
     }
 }

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

Fig. 5. Student class and its internal checker.



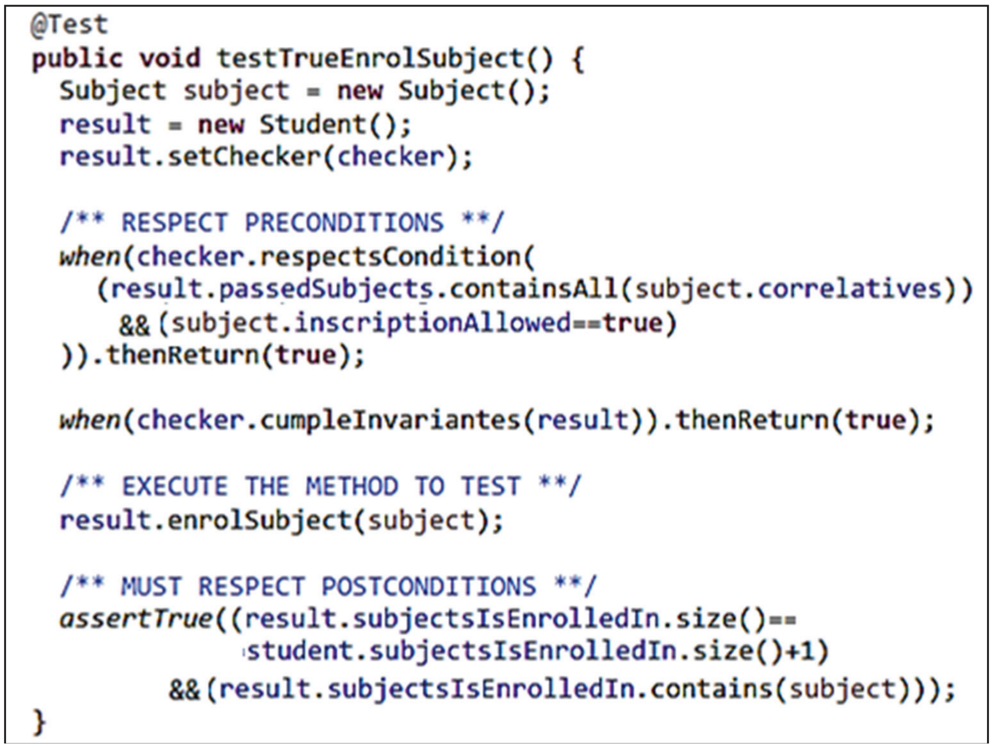
```
public void enrolSubject(Subject subject) {
   Student previous = this.saveState();
   if(this.getChecker().respectsCondition(
       (this.passedSubjects.containsAll(subject.correlatives)) &&
       (subject.inscriptionAllowed==true) ))
     {
           this.subjectsIsEnrolledIn.add(subject);
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

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

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

Fig. 6. Student class generated methods.

Then, figure 7 illustrates a Test that performs the validation of invariants, pre conditions, and post conditions.

Improving Tests with a Richer Formalism

The process described above allows developers to automatically obtain the code of the test cases from the UML models. These tests are executed dynamically while the program is running or during the testing phase using testing inputs, which should be obtained applying appropriate techniques that are out of the scope of MDT+.



```
@Test
public void testTrueEnrolSubject() {
  Subject subject = new Subject();
  result = new Student();
  result.setChecker(checker);

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

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

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

  /** MUST RESPECT POSTCONDITIONS **/
  assertTrue((result.subjectsIsEnrolledIn.size()==
              student.subjectsIsEnrolledIn.size()+1)
         && (result.subjectsIsEnrolledIn.contains(subject)));
}
```

Fig. 7. Example test: Student class method.

At the same time, MDT+ offers another level of analysis, enabling the static checking of model consistency, prior to execution. Static checking is achieved by integrating the formal language Alloy [Jackson 2006]. Alloy is a 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 analysis tool named Alloy Analyzer [Alloy 2017]. The Alloy Analyzer applies a bounded verification, limiting the number of objects that populate each class and checking assertions over the specification within that bound. It uses a SAT-solver to answer verification queries, converting them to Boolean formulas.

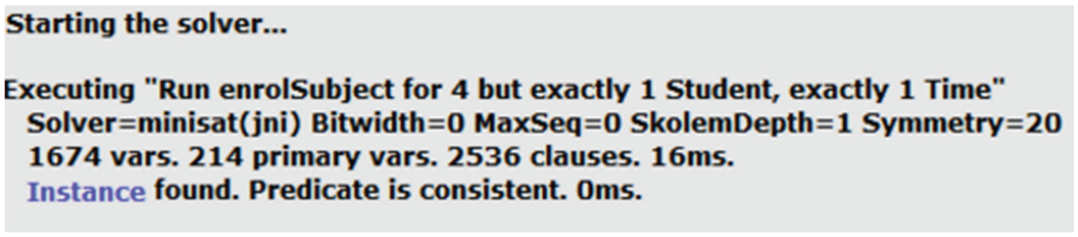
MDT+ uses the AlloyMDA tool [Cunha et al 2015] to translate the generated OCL code to its correspondent Alloy code, from which the Alloy Analyzer is used to check consistency. Figure 8 shows the Alloy code obtained from the UML/OCL model in figure 1.

When the Alloy Analyzer is executed, the constraints to be checked within a scope (setting boundaries) are specified using the special command **run.** The potential errors will occur within this scope, being possible to have more/other errors outside. That is to say, if an example is found, the constraints are satisfied. On the other hand, if no example is found, the constraints are invalid (false for every possible example), or may be valid but outside the specified scope. The following command raises the checking for the .als file:

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

In this case, the constraint *enrolSubject* is tested with a scope that limit the search to those instances that have at most 4 instances of each signature, except for Student, which has just one object. Also, for the sake of simplicity just one time instance is considered.

Figure 8 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 9 shows the example that was found.



```
Starting the solver...

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.
```

Fig. 8. Alloy Analyzer results.

Related Work

Several tools provide support for automatic test code generation from software models. The ones most closely related to MDT+ are summarize here.

TestEra [Khalek et al 2011] is a Java testing framework based on formal specifications. To test a method, it uses the methods pre conditions specification to generate tests inputs and the post conditions to check the output correctness. TestEra supports specifications written in Alloy and uses the SAT-based back-end of the Alloy tool-set for systematic generation of test suites. Each test case is a JUnit test method, which performs three key steps: (1) initialization of pre-state, i.e., creation of inputs to the method under test; (2) invocation of the method; and (3) checking the correctness of post-state.

Modeling languages UML and OCL offer a huge set of constructs. In [Hilkenet al 2014] an approach is proposed, using model transformations to unify different description means within a so-called base model. Along the transformation, complex language constructs are expressed with a small subset of so-called core elements. This simplification allows interacting with a wide range of verification engines with different advantages and weaknesses.

In [Kuhlmannal 2011] a method for efficiently searching for model instances is provided. The existence or non-existence of model instances with certain properties allows significant conclusions about model properties. The approach is based on the translation of UML and OCL constraints into relational logic and its analysis with SAT solvers. The proposal is realized by integrating a model validator as a plugin into the UML and OCL tool USE.

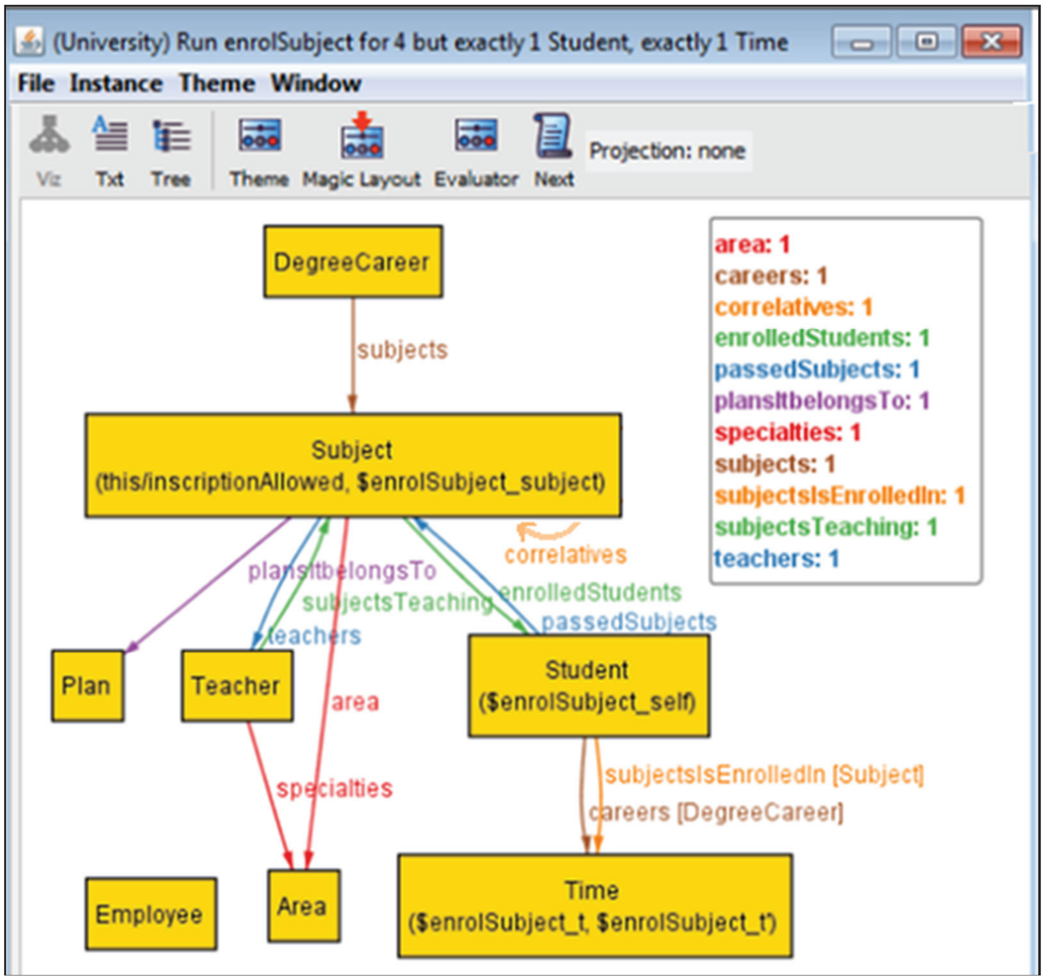


Fig. 9. Model instance found by the analyzer.

In [Nabuco 2014] a tool to filter/setup test cases from models is introduced. 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 in the model. The tool offers a filtering mechanism in order to generate a reasonable number of test cases, reducing complexity.

In [Bucchiarone 2014] model-checking techniques are used to validate the software architecture model conformance with respect to selected properties, while testing techniques are used to validate the implementation conformance to the software architecture model. The specification, consisting of a topology definition and state diagrams, is translated to the Promela formalism where the SPIN model checker is applied.

The Fokus!MBT tool [Wendland et al 2013] is a multi-paradigm test modeling environment which gives users the freedom to choose among programmatic and diagrammatic notations, as well as state-based and scenario-based styles, reflecting the different concerns in the process. The diverse model styles can be combined by model composition in order to achieve an integrated and collaborative model-based testing process. The approach is realized in the successor of Microsoft MBT tool Spec Explorer, and has a formal foundation in the framework of action machines.

Conclusion and Future Works

The MDT+ tool allows software 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 the Java language but also in two formal languages, which are 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 tool provides more reliable support by amalgamating different analysis techniques, which strengthens the software validation process. While model-checking finds bugs in high-level system designs, testing identifies bugs in implementation level code. Considering the strong complementarity between those two worlds, an integration of them offers promising advantages.

In comparison to the related works described before, the following advantages are emphasized:

-Dual analysis: MDT+ achieves both static and dynamic analysis.

-UML-Alloy connection: generally, the proposed tools associate UML/OCL with MDT or OCL with Alloy. In this case, MDT+ consistently integrates the three of them.

-Better Tools: MDT+ is built on top of stronger and newer tools (i.e., Acceleo, Papyrus and Mockito), in contrast to the tools used in the previous works (i.e., MOFScript and EasyMock).

-Complete process: generally, only one part of the software development process is automated. In this case, MDT+ provides a code ready for production which is verifiable, adaptable and usable for a wide range of users.

MDT+ was initiated in Ilan Rosenfeld´s thesis [Rosenfeld 2016] and to extend the proposed solution the following lines are being considered:

- The re-generation of automatically generated code preserving possible updates made for the developer will be provided. This is achieved by using special markers in the code text.
- Less abstract tests will be generated without using mocks.
- When finding an inconsistence in the source model, counterexamples in the natural/Java language will be generated, to improve the understandability for users with little knowledge in formal verification.

-The developer will be able to select other programming language for the generated code (additionally to Java).

References

M. Brambilla, J. Cabot and M. Wimmer (2012). "Model-Driven Software Engineering in Practice". Morgan&Claypool Publishers ISBN: 9781608458820.

T. Stahl and M. Voelter (2006). “Model-Driven Software Development”. Technology, Engineering, Management. 1st edition. Wiley..

D. Di Ruscio, R.F. Paige and A. Pierantonio Editors (2014). Science of Computer Programming. Special issue on Success Stories in Model Driven Engineering. Vol. 89, pp. 69-222. Elsevier.

OMG (2015). OMG success stories. [Online]. Available: http://www.omg.org/mda/products_success.htm, last access.

UML (2017). Unified Modeling LanguageTM (UML) [Online]. Available: http://www.omg.org/spec/UML/

OCL (2017). The Object Constraint Language [Online]. Available: https://www.omg.org/spec/OCL/2.4/

M. Utting and B. Legeard (2007). Practical Model Based Testing: A tools approach. Morgan Kaufmann Publishers Inc. USA ©.

EMP (2017). Eclipse Modeling Project. [Online]. Available: https://eclipse.org/modeling

EMF (2017). Eclipse Modeling Framework EMF: [Online]. Available: http://eclipse.org/modeling/emf/

Papyrus: (2017). Papyrus: [Online]. Available: http://eclipse.org/papyrus

Acceleo (2017). Acceleo: [Online]. Available: https://projects.eclipse.org/projects/modeling.Acceleo

JUnit (2017). JUnit: [Online]. Available: http://junit.org/junit4/

Mockito (2017). Mockito: [Online]. Available: http://site.mockito.org/

Jackson, D. (2006): Software Abstractions: Logic, Language, and Analysis. MIT Press.

Alloy (2017). Alloy tool: [Online]. Available: http://alloytools.org/ "

A. Cunha, A. Garis, D. Riesco (2015): Translating between Alloy specifications and UML class diagrams annotated with OCL. Software and System Modeling 14(1): 5-25 AlloyMDA: [Online]. Available: http://sourceforge.net/p/alloymda/wiki/Home/

S. Khalek, G. Yank, L. Zhang, D. Marinovt, S. Khurshid (2011). TestEra: A tool for testing Java Programs using Alloy specifications. 26th IEEE/ACM International Conference on Automated Software Engineering (ASE).

Hilken, F., Niemann, P., Wille, R., Gogolla, M (2014): Towards a base model for UML and OCL verification. In: Boulanger, F., Famelis, M., Ratiu, D. (eds.) MoDeVVa@MODELS. pp. 59–68.

Kuhlmann, M., Hamann, L., Gogolla, M. (2011): Extensive validation of OCL models by integrating SAT solving into USE. In: Bishop, J., Vallecillo, A. (eds.) TOOLS 2011. LNCS, vol. 6705, pp. 290–306. Springer, Heidelberg.

M. Nabuco, Ana C.R. Paiva (2014). Model-based test case generation for Web Applications. In Proceeding of the 14th Int. Conf. on Computational Science and Its Applications — ICCSA 2014 – Vol. 8584. Springer-Verlag. New York, NY, USA.

A. Bucchiarone, H. Muccini, P. Pelliccione, and P. Pierini (2014). Model-Checking plus Testing: from Software Architecture Analysis to Code Testing. In Proc. Int.Workshop on Integration of Testing Methodologies,ITM ’04. LNCS n.3236.

M. Wendland, Andreas Homann, Ina Schieferdecker (2013). Fokus!MBT – A multi-paradigmatic test modeling environment. in: ACME '13 Proceedings of the workshop on ACadeMics Tooling with Eclipse, Montpellier, France. ACM New York, NY, USA.

Ilan Rosenfeld. (2016). “Lenguajes formales y derivación automática de código de pruebas a partir de modelos de software con restricciones OCL”. Informatics thesis, UNLP. Argentina.