



Institute of Technologies and Information Systems

CASTILLA-LA MANCHA UNIVERSITY

PhD Thesis

MANDINGA: METHODOLOGY FOR AUTOMATION TESTING INTEGRATING FUNCTIONAL AND NON-FUNCTIONAL ASPECTS

PhD Student:

Federico Leonardo Toledo Rodríguez

Supervisors:

PhD. Macario Polo Usaola

PhD. Beatriz Pérez Lamancha

Ciudad Real, Spain, 2014

Dedicado a la persona que me dio la enseñanza más importante de mi vida, y no lo hizo con palabras sino predicando con el ejemplo, pero intentando llevarlo a papel esta enseñanza fue: *no importa lo adversa que parezca la situación, siempre con mucho esfuerzo y con absoluta bondad se puede salir adelante.* Gracias mamá, ¿qué sería de mi sin vos?

ACKNOWLEDGMENTS

It is impossible to mention all the people with whom I am grateful, but I will try.

First of all, I would like to thank my advisors, Macario and Beatriz, for their technical and non-technical advice during the realization the thesis, which would not exist without their supervision.

I would like to extend my thanks to my research workmates of these years: Pedro, Ricardo, Laura, Tomás, Beatriz, Ana, Andrea, César, Alberto, María, Alberto, Rubén and Jesús. We had great time working together.

I am also very grateful to the Alarcos Research Group, head Mario Piattini and all its members. Their support to carry out this thesis was very valuable too.

Also, I would like to thank the CNR, in Pisa, Italy, and especially Antonia and Francesca that have allowed me to develop my research stage with them, supporting and extending the horizons of my research. In this case I would like to extend my thanks to the colleagues I had there in this period: Breno, Guilliemo, Andrea, Eda and many more.

I would also like to express my gratitude to all the people of the *Centro de Ensayos de Software*. My knowledge and passion about testing grew up working with them.

I am also very grateful to the new friends I have made in Spain and Italy. They never let me feel homesick. Of course, I would like to thank my old friends I am wishing to meet again. I consider myself very lucky because of the people around me. They are the best on the world.

My most important thanks go to my family; they are my main motivation, inspiration and support.

Last but not least, my greatest thanks go to all *Abstracta* team, especially to my close friends Matías and Fabián, I am very happy to share the most important projects of my life with you.

“El Mandinga” is the name of the Devil in South American culture, when he appears as a normal human being. When strange things happen, *“es cosa é Mandinga”* means that something with no rational explanation or supernatural has just occurred.

Testing can be considered a destructive process. Often, developers see testers as devils who break their (already broken) software. There he is, *Mandinga*, trying to find a way to make the software commit a sin, an explanation for the weird behavior that software sometimes has.



CONTENT

Content	9
Abstract	13
CHAPTER 1. Introduction.....	17
1.1. Motivation	17
1.2. Goals	18
1.3. An Illustrative Example	19
1.4. Expected Contributions	27
1.5. Research Methods	29
1.5.1. Proof of Concept	29
1.5.2. Action-Research with Abstracta	30
1.6. Context	31
1.7. Structure of the Document.....	32
CHAPTER 2. State of the Art and State of Practice.....	37
2.1. Introduction.....	37
2.2. Models and Metamodels.....	38
2.2.1. Unified Modeling Language (UML)	39
2.2.1.1. Tools and Graphical Editors.....	39
2.2.1.2. Extension Mechanisms.....	40
2.2.2. Model Transformations	40
2.2.3. State of the Art in Information System Models.....	42
2.2.3.1. Functional Specification	42
2.2.3.2. Non-Functional Specification	44
2.2.4. State of the Art in Functional Testing Modeling.....	46
2.2.5. State of the Art in Performance Testing Modeling.....	50
2.2.5.1. Model Driven Approaches for Performance Testing	50
2.2.5.2. Current Version of UML-TP for Performance Testing.....	56
2.3. Processes, Methodologies and Approaches in the Current Practice.....	60
2.3.1. Reverse Engineering and Reverse Engineering of Databases	61
2.3.2. Implementation of Functional Tests	62
2.3.3. Implementation of Performance Tests	63
2.3.4. Measurement of Test Quality	65
2.4. Conclusion of the State of the Art and Practice	67
CHAPTER 3. MANDINGA: Methodology for Automation Testing Integrating Functional And Non-Functional Aspects.....	71
3.1. Introduction.....	71
3.2. Methodology	72
CHAPTER 4. Information System Model Construction	77
4.1. Introduction.....	77
4.2. Information System Metamodel.....	78
4.2.1. Data Model	78
4.2.2. Graphic User Interface Model.....	80
4.2.3. Business Rules.....	84
4.3. DBesTest Implementation	85

4.3.1.	Database Structure Extraction	86
4.3.2.	Information System Model Generation	90
4.4.	Conclusion	95
CHAPTER 5.	Automatic Generation of Functional Test Cases	99
5.1.	Introduction	99
5.2.	Test Model Generation	100
5.2.1.	Test Architecture	101
5.2.2.	Test Case Generation	103
5.2.2.1.	Coverage Approximations	105
5.2.2.1.1.	Coverage on Class Diagrams	106
5.2.2.1.2.	Coverage on State Machines	107
5.2.2.2.	Test Patterns	107
5.2.2.2.1.	One-table Patterns.....	108
5.2.2.2.2.	Two-table Patterns	112
5.2.2.2.3.	Three-table Patterns.....	117
5.2.3.	Test Data Model Generation	120
5.2.3.1.	Equivalence Class Partitioning.....	120
5.2.3.2.	Structured Test Data	122
5.2.3.3.	Invalid Data Generation	123
5.2.3.3.1.	Criteria	123
5.2.3.3.2.	Example	125
5.3.	Test Code Generation	126
5.3.1.1.	Test Behavior	126
5.3.1.2.	Platform Specific Execution.....	128
5.3.1.3.	Parameterization.....	130
5.3.1.4.	Test Data Generation	131
5.4.	Conclusion	132
CHAPTER 6.	Automatic Generation of Performance Tests	137
6.1.	Introduction	137
6.2.	Model-Based Test Cases Design Integrating Functional and Non-Functional Aspects	139
6.2.1.	Contributions to PMM	140
6.2.2.	Contributions to UML-TP	144
6.2.2.1.	Workload Information.....	144
6.2.2.2.	Non-Functional Validations	146
6.2.3.	Workload Generation and PMM Operators Coverage	148
6.2.3.1.	Test Model Generation Algorithm.....	148
6.2.3.2.	Example.....	154
6.3.	Executable Non-Functional Test Cases Generation	157
6.3.1.	Background and Motivation.....	158
6.3.2.	Automatic Generation of Executable Test Cases.....	159
6.4.	Conclusion	163
CHAPTER 7.	MANDINGA: Implementation, Automation and Application in the Industry	167
7.1.	Standard and Generalized Approach	167
7.1.1.	Dealing with Model-Driven Standard Tools	168
7.1.1.1.	UML and Modeling Tools	168
7.1.1.2.	Model Transformation Tools.....	170
7.1.2.	DBesTest	172

7.1.2.1.	Description	172
7.1.2.2.	Limitations	176
7.1.3.	Generation of Automated Performance Test Cases	177
7.1.3.1.	Description	177
7.1.3.2.	Limitations	177
7.1.4.	Generation of Non-functional Test Scenarios	177
7.1.4.1.	Description	178
7.1.4.2.	Limitations	178
7.1.5.	Conclusions About the Proof of Concept	178
7.2.	Transfer to Industry: GeneXus and GXtest	179
7.2.1.	Background: GeneXus and GXtest	179
7.2.2.	GXtest Generator	183
7.2.2.1.	Description of the Adaptation to GeneXus	184
7.2.2.2.	Experiences in industry	187
7.2.2.2.1.	Study Case: Bancard Paraguay	188
7.2.2.3.	Analysis of Errors Found	189
7.2.2.4.	Evaluation of the Case Studies	190
7.2.2.5.	Limitations	191
7.2.3.	GXtest for Performance Testing	191
7.2.3.1.	Description	192
7.2.3.2.	Experiences in industry	192
7.2.3.3.	Evaluation of the Case Studies	193
7.2.3.4.	Limitations	195
7.3.	Final Discussion	195
CHAPTER 8.	Conclusions and Future Research Lines	199
8.1.	Summary	199
8.2.	Contributions	200
8.3.	Analysis of the Goals Consecution	201
8.4.	Publications	203
8.5.	Future Work Lines	206
Annex 1.	Difficulties Faced with Model-Driven Approaches	211
A1.1.	Introduction and Motivation	211
A1.2.	Using UML with Model Transformations	212
A1.3.	Conclusion	219
Bibliography	221	

ABSTRACT

An Information System (IS) is a software system that allows the manipulation of structured data for a specific business goal, especially in a database. With the growth of the internet and web applications, and now with mobile applications, the use of these systems is embedded in our lives. As a result the importance of testing in the IS development process has been growing, looking for improvements in functional and non-functional aspects of quality.

In model-driven testing approaches different models are usually used, with two aims, some models for functional testing and some other models for non-functional testing. Our approach moves away from this traditional practice and breaks down the boundaries between functional and non-functional testing, incorporating both aspects into a comprehensive testing model which will later be translated into test code and will be useful for performing functional and non-functional validations.

Commonly, ISs consist of applications which deal with the information saved in relational databases, storing the data of different entities on the basis of particular business rules. Thus, there is a correspondence between the visual components (e.g. web forms), the data structures and the logic in the middle to accomplish the business rules. The basic operations to manipulate data structures are the CRUD operations (create, read, update, delete). For example, if values are updated in the user interface, this will produce the execution of an operation on an object in the middle layer, and then an update operation on the database.

Taking this into account, the data model can be used as a basis to generate test cases, verifying the way that application layers manage these structures.

This thesis proposes a methodology to automatically generate test cases from existing IS, paying special attention on CRUD operations. It takes a UML representation of the SUT which has been automatically obtained through reverse engineering from the IS database. A model-driven approach is then applied to obtain executable functional and non-functional test cases. In the middle, a set of model to model and model to text transformations are in charge of performing the main stages of MANDINGA. After reverse-engineering the IS, MANDINGA deals with different models to represent the aspects of the system which are interesting from the testing point of view:

- A Data Model to represent the database structure, including entities and relationships.
- A Graphic User Interface model to keep information about the elements with which the user interacts and the way they navigate on the IS.
- A Non-Functional Properties model to describe other aspects of system requirements such as dependability, performance, security, etc.
- A standard UML profile for Testing (UML Testing Profile) to represent the test model generated.
- Model-Implementation Mapping, in order to relate the model elements to the elements on the SUT, in order to be able to generate completely executable test cases.

Transformations between models are carried out with ATL, the *de facto* standard for model-to-model transformations. The final generation of executable test code is made with Aceleo scripts, the pragmatic implementation of the standard for model-to-text transformations.

One of the main contributions of this thesis is the return into the industry of the proposed methodology. The general framework was adapted for a specific model-driven environment called GeneXus, and its testing tool GXtest. The same scheme was developed and used within industry for this environment: test cases are generated from the data model, and then, with this test model, functional and non-functional validations can be performed automatically on IS developed with GeneXus. This set of tools have been used industrially in different real projects providing successful results, specially reducing costs in the preparation of automated test cases and performance testing.

*Be ready, heart, for parting, new endeavor,
be ready bravely and without remorse
To find new light that old ties cannot give.
In all beginnings dwells a magic force
for guarding us and helping us to live.*

– Herman Hesse

CHAPTER 1. INTRODUCTION

This chapter explains the subject matter and how this thesis attempts to address it. It covers the motivation and goals of this thesis as well as the expected contributions. Chapter 1 introduces the illustrative example to be used in the subsequent chapters and finally outlines the methodology and the context of the research.

1.1. MOTIVATION

The use of Information Systems (IS) has increased in recent years not only for leisure and personal use, but also for important tasks, work, company decision making, etc. As an IS provides information to its users, which is stored in databases represented as a set of entities and relationships, the quality of the system used to manage the data becomes a factor in its success, and therefore, a way to ensure that quality is necessary.

Hainaut et al. define *information systems* or *data-oriented applications* as the applications whose central component is the database (or a set of permanent files) [1], which is composed of entities and relationships.

A study by Grady et al. [2] showed that 10.6% of corrective maintenance requests presented by users proceed from a wrong manipulation of data structures, and 11.8% are due to errors in the management of events and data in the user interface. In our recent experience (several years working in different testing teams, specialized testing companies and testing consultancy services), the observation has been of a similar or slightly higher distribution of errors. “Wrong manipulation” entails, for example, the management of:

- Duplicated values when data is entered for a column with a unique restriction
- Strings longer than expected
- Invalid foreign keys

- Invalid format: letters instead of numbers, malformed or invalid dates

In fact, in systems that use databases intensively, the right manipulation of data structures is especially important not only for the need of preserving the integrity and consistency of data, but also for the strong influence of the data schema on the design of the remaining layers and components of the applications that use it. Ideally, the database schema can be seen as the “precursor” of the domain model of the applications which deal with it.

Although data management is a responsibility assigned to the selected database management system (usually a third-party system acquired from a vendor, which it can be assumed deals properly with the defined constraints of the data schema, such as foreign keys, check constraints, etc.), it is also important to test how the applications that use the database process this data.

Functionality is not the only important aspect of quality, however. There are many more properties, considered as “non-functional” (according to the different product quality characteristics specified in the *Software product Quality Requirements and Evaluation* standard - SQuaRE [3]), from which there is a special concern about “performance” and “availability” when talking about Information Systems, considering that they are accessed, in most of the cases, by multiple concurrent users.

This is especially interesting for platform migrations, adaptive or perfective maintenance. When the systems are released to users there are also risks concerning functional and non-functional properties. It is expected that the functionality, performance and availability of the system are adequate; otherwise the system is not going to be adopted by users. Therefore, a preventive workload simulation (to verify non-functional properties) is crucial to guarantee the success of any implementation project.

These quality factors, specifically for Information Systems using databases, are the main concerns of this thesis.

1.2. GOALS

The first goal is to automate the test case design and execution for applications that make use of databases, in the context of web environments, but with the possibility of extending the approach to other types of applications, such as mobile applications.

The second goal is then to reduce the costs associated with performance testing, improving the automation process with more flexibility.

In addition, both tests are typically performed separately, one after the other. First, functional testing is executed, then, performance testing. This makes sense because the automation is done with different kind of tools, and the required skills of the testers are different for each case.

In order to save costs related to execution time, and help testers to perform these tasks automatically, the **third goal is** to propose an integrated view, considering the verification of functional and non-functional properties. In our approach the objective is to have a framework and a process that automate both aspects in an integrated, unified way.

From the beginning the focus is to obtain practical and industrially applicable results.

For all cases, a black-box approach is going to be taken (which implies taking into consideration the inputs and outputs of the SUT without paying attention to its internal structure, thus considering it a black box [4]), so that it should be useful for testers with no access to the source code (to allow outsourcing of testing services for example).

1.3. AN ILLUSTRATIVE EXAMPLE

Consider, as a running example, the AjaxSample as the System Under Test (the SUT), which is available at <http://samples.genexus.com/ajaxsample>. It is an open source demonstration system, developed with GeneXus (a model-driven development tool developed by Artech): it manages invoices, products and clients through a web interface and a database to store the data. Figure 1 shows the web interface for creating invoices.

Imagine that the database was inherited from a previous version of the system that was implemented in an older technology, such as Visual Fox or RPG (green screen). The database was therefore given to the development team, and they developed a new set of layers on it: data access, domain logic and web presentation. These layers were designed to properly deal with the database schema.

One of the aspects of this new version of the system that should be tested is the interaction between the new layers and the database. So, taking the database metadata into account, it is possible to design test cases and test data in order to verify boundary situations, or common errors in the management of the database structures and column data types.

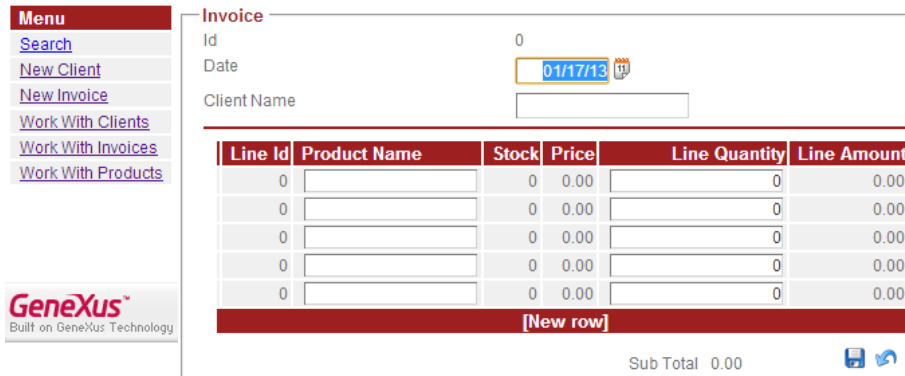


FIGURE 1 - AJAXSAMPLE WEB INTERFACE

Therefore, from the database schema, as the diagram in Figure 2 that corresponds to the AjaxSample, a tester could derive a set of test cases to be executed on the system interface, thus verifying whether the complete system (the three layers) can or cannot correctly manage the database structure.

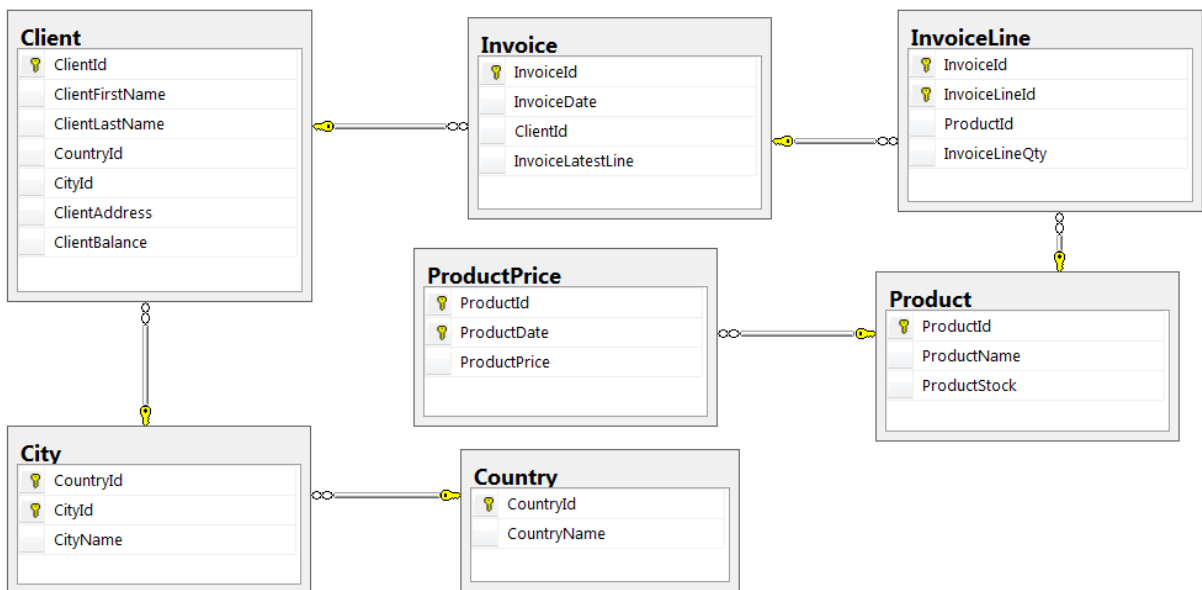


FIGURE 2 - AJAXSAMPLE DATABASE SCHEMA

Associated with this data model, there are a set of business rules, some of them implemented as database restrictions, and others managed in the logic of the application. Table 1 shows some examples.

TABLE 1 - BUSINESS RULES EXAMPLES

Rules
(ClientLastName, ClientFirstName) is unique
CityName is not null
A client cannot buy more products if the balance (ClientBalance) is less than "-2.000".
ProductPrice must be greater than 0
ProductStock must be greater than 0

In order to give an idea of the test cases that can be derived from the database schema, Table 2 shows some examples.

It is therefore possible to derive interesting situations from the data model from a testing point of view (e.g. the creation of invoices with no associated products or the introduction of strings with 31 characters in a field corresponding to a *varchar(30)* column, applying different test design techniques). Although the DBMS will probably manage these scenarios in a proper manner, the goal of the test cases is to check how the application layers deal with them: (1) even though the invoice with no product is not created, how will the application layer consider this special situation?; and (2) will the user interface show the complete string, or will the user know that the string has been truncated?

TABLE 2 - EXAMPLE TEST CASES

Description	Examples
Robustness problems	
Entering data into the web user interface that does not respect the database restrictions.	Duplicated values when you enter data for a column with a unique restriction, as for instance two clients with the same first name and last name. Null values in not-null columns, as for <i>CityName</i> .
Entering data, or executing actions, in the web user interface, that do not respect the database structure.	Invalid foreign keys, e.g. creating an invoice with a non-existing client. Deleting an instance of <i>Client</i> that is referenced from an instance of <i>Invoice</i> , because the foreign key is not-null.
Entering data, in the web user interface, that does not respect the database data types (out of range of the corresponding column's data type).	Strings longer than expected, as for instance in table <i>Client</i> , the column <i>ClientAddress</i> accepts up to 30 characters, but the user interface input allows the user to enter up to 50 (will the value then be truncated or will the error be controlled and properly shown in the user interface?) Invalid format: letters instead of numbers for the line quantity, malformed or invalid dates for the <i>InvoiceDate</i> (2013-55-55), etc. Check rules not respected.
Restrictions that cannot be accomplished.	It will not be possible to create an invoice when the client's table is empty.
Gotchas	
Column defined as unique, but not as NOT-NULL, it allows insert of different rows with NULL (so, you cannot distinguish between the created rows).	It is desirable to distinguish products by their name, but you can put null values for <i>ProductName</i> (as many as you want).
If null values are inserted in numeric columns (that do not have a not-null restriction) it could cause a problem if the column is involved in any kind of calculus.	For instance, <i>ProductStock</i> accepts null values, and what could happen when a new invoice is created, and there will be needed to see if there is enough stock.
Transactions for multiples tables	
If the creation of a composed entity is composed of the creation of instances in different tables, it is interesting to test with valid data for some tables, and invalid for others, to verify that the data is only stored when all the instances are correct.	When creating an <i>Invoice</i> , it is interesting to test with invalid data for one of the <i>Products</i> of the invoice lines, in order to see if the header of the invoice was stored and the rest of the elements were not, i.e., the data of the <i>Invoice</i> table is stored but the data for <i>InvoiceLine</i> is not because it is invalid.
Cascade	
Update or delete instances that have cascade effects.	If a product is deleted, are all the invoices lines associated deleted too? Or is it not permitted?
Relationship boundaries	
With operations of processing instances, there are also boundary situations considering the relationships: if the relationship is 0..1 to 0..n, then there are some situations to consider as border: 0:1, 0:N, etc.	Is it possible to create invoices without any product?
Business rules	
Boundaries within the rules	What happens trying with a <i>ClientBalance</i> under -2.000? Or with a negative <i>ProductPrice</i> ?

Current testing technologies allow testers to automate the execution of test cases in order to improve the required time in the regression testing activities. Those test cases

from Table 2 can therefore be automated to reduce execution costs. For this, and considering that AjaxSample has a web interface, it is possible to automate the test cases using different tools such as Selenium¹ or Watir² (examples of popular open source projects). Figure 3 shows a simple Selenium script that a tester prepared in order to automate the execution of a test case that inserts invalid data, creating a new product in AjaxSample.

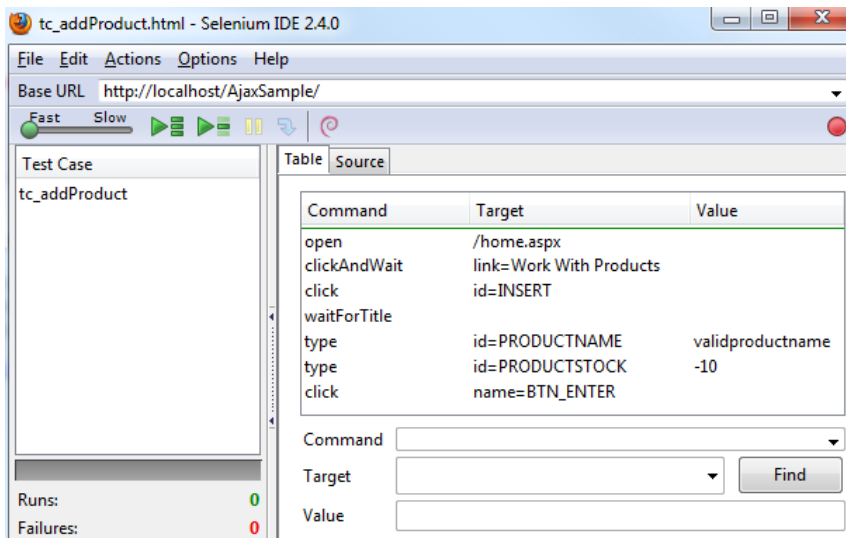


FIGURE 3 - SELENIUM TEST CASE

Basically, it is an ordered set of commands, which in this case starts opening the home page of the SUT, goes to the menu (*Work With Products*), in this page goes to the *Insert* option, and in this page inserts data to create a new product (*validproductname* for the product name in the input *PRODUCTNAME* and “-10” for the product stock in *PRODUCTSTOCK*) and finally clicks the button (*BTN_ENTER*) to complete the transaction (more detail about Selenium tool is presented in Chapter 2).

With an artifact like that it is possible to easily execute the regression tests, but typically this is exported and integrated with JUnit [5], in a test program as shown in Table 3.

¹ Selenium: <http://www.seleniumhq.org/>

² Watir: <http://watir.com/>

TABLE 3 - JUNIT AND SELENIUM INTEGRATION

```

public class JunitSelenium {
    private WebDriver driver;
    private String baseUrl;
    private boolean acceptNextAlert = true;
    private StringBuffer verificationErrors = new StringBuffer();

    @Before
    public void setUp() throws Exception {
        driver = new FirefoxDriver();
        baseUrl = "http://localhost/AjaxSample/";
        driver.manage().timeouts().implicitlyWait(30, TimeUnit.SECONDS);
    }

    @Test
    public void testJunitSelenium() throws Exception {
        driver.get(baseUrl + "/home.aspx");
        driver.findElement(By.LinkText("Work With Products")).click();
        driver.findElement(By.id("INSERT")).click();
        for (int second = 0; second < 60; second++) {
            if (second >= 60) fail("timeout");
            try { if ("".equals(driver.getTitle())) break; } catch (Exception e) {}
            Thread.sleep(1000);
        }

        driver.findElement(By.id("PRODUCTNAME")).clear();
        driver.findElement(By.id("PRODUCTNAME")).sendKeys("validproductname");
        driver.findElement(By.id("PRODUCTSTOCK")).clear();
        driver.findElement(By.id("PRODUCTSTOCK")).sendKeys("-10");
        driver.findElement(By.name("BTN_ENTER")).click();
    }

    @After
    public void tearDown() throws Exception {
        driver.quit();
        String verificationErrorString = verificationErrors.toString();
        if (!"".equals(verificationErrorString)) {
            fail(verificationErrorString);
        }
    }
}

```

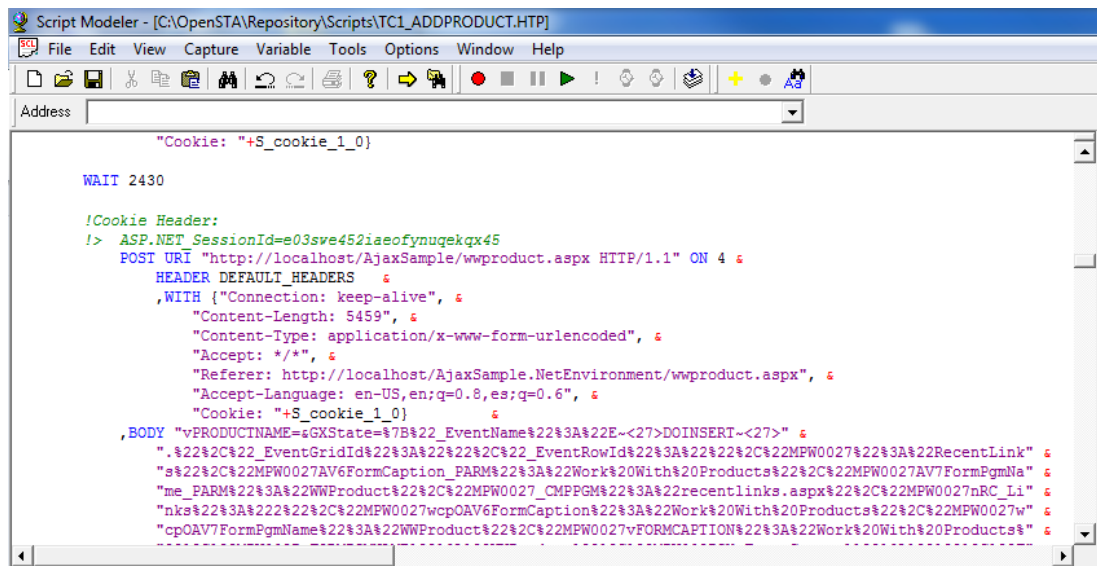
After the functional verification and its automation, and considering that the application is going to be used by multiple concurrent users (hundreds, maybe thousands), it is also necessary to perform a load simulation test in order to mitigate the risks related with the system and infrastructure performance when it is exposed to such workload. For this purpose, the tester uses tools that simulate the action of multiple users concurrently accessing the system, verifying the health status of the system under this artificial workload. In this case, tools such as OpenSTA³ and JMeter⁴ simulate *virtual users*

³ OpenSTA: <http://opensta.org/>

⁴ JMeter: <http://jmeter.apache.org/>

executing different test cases and measuring their response times. Figure 4 shows an example of a test case automated with OpenSTA so as to execute it as part of the workload presented in Figure 5.

The script corresponds to the POST http method for the insertion of a new product (a special kind of http message to the server invoking a request on the server sending a set of parameters).



```
Script Modeler - [C:\OpenSTA\Repository\Scripts\TC1_ADDPRODUCT.HTP]
File Edit View Capture Variable Tools Options Window Help
Address
"Cookie: "+S_cookie_1_0}

WAIT 2430

!Cookie Header:
!> ASP.NET_SessionId=e03sve452iaeofynugekqx45
POST URI "http://localhost/AjaxSample/wwproduct.aspx HTTP/1.1" ON 4 &
HEADER DEFAULT_HEADERS &
,WITH {"Connection: keep-alive", &
      "Content-Length: 5459", &
      "Content-Type: application/x-www-form-urlencoded", &
      "Accept: */*", &
      "Referer: http://localhost/AjaxSample.NetEnvironment/wwproduct.aspx", &
      "Accept-Language: en-US,en;q=0.8,es;q=0.6", &
      "Cookie: "+S_cookie_1_0} &
,BODY "vPRODUCTNAME=%GXState=%7B%22_EventName%22%3A%22E~<27>DOINSERT~<27>" &
      ".%22%2C%22_EventGridId%22%3A%22%22%2C%22_EventRowId%22%3A%22%22%2C%22MPW0027%22%3A%22RecentLink" &
      "%22%2C%22MPW0027AV6FormCaption_PARM%22%3A%22Work%20With%20Products%22%2C%22MPW0027AV7FormPgmNa" &
      "me_PARM%22%3A%22WWProduct%22%2C%22MPW0027_CMPPGM%22%3A%22recentlinks.aspx%22%2C%22MPW0027nRC_Li" &
      "nks%22%3A%22%22%22%2C%22MPW0027wcpOAV6FormCaption%22%3A%22Work%20With%20Products%22%2C%22MPW0027w" &
      "cpOAV7FormPgmName%22%3A%22WWProduct%22%2C%22MPW0027vFORMCAPTION%22%3A%22Work%20With%20Products%" &
```

FIGURE 4 - OPENSTA SCRIPT

This kind of tool works at a communication protocol level, thus the automated test case is an ordered set of commands executing http actions on the server. The workload is a compound of different scripts, indicating how many users are going to execute each one. In Figure 5 the load test was configured to execute 200 virtual users (VU) adding products, 100 VU adding new clients, and 15 VU buying products and therefore creating new invoices. Executing this workload the team can verify the response times and resource use among others.

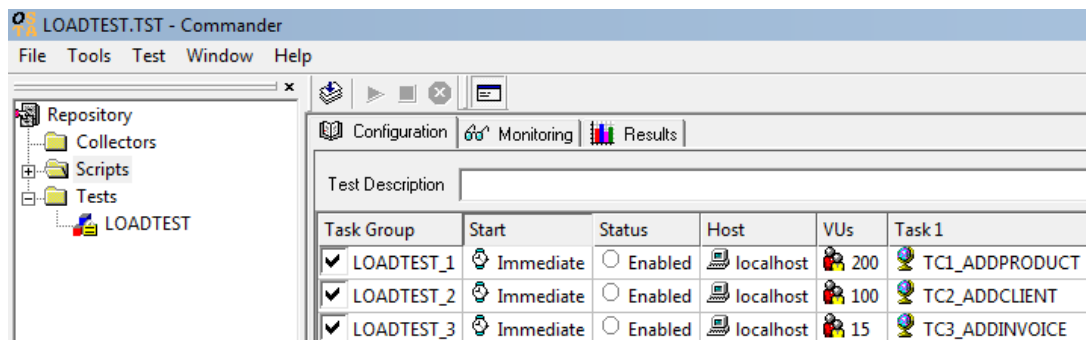


FIGURE 5 - OPENSTA TEST SCENARIO

Once those tasks are properly finished, the tester can consider that they have contributed to the functional and non-functional quality of the SUT.

In this traditional approach there are some drawbacks that could be improved:

- The tester requires knowledge of the database schema, and needs access to it.
- The tester requires knowledge in testing techniques.
- The tester requires knowledge in functional testing automation tools. Typically this is equivalent to a programming language.
- The tester requires knowledge in load simulation testing tools. Typically this includes not only knowing a programming or scripting language, but also knowledge about the communication protocol (at least HTTP in web systems).
- If the system is migrated to another platform, or if the test team decides to change the automation tool, etc., it will be necessary to rebuild all the test artifacts from the scratch.

Taking these concerns into consideration, a **model-based approach** could be used to tackle them. This kind of approaches increases the level of abstraction making the process more understandable and perdurable: i) understandable because it is easier to work with models than with code; ii) perdurable because if the models are independent of the platform, the test knowledge could be preserved even after platform changes. Later, the models can be transformed and refined to new test code.

If the tester provides information about the functional requirements and design, it is possible to design a test model covering only the functional quality aspects. If the tester provides non-functional requirements, it is possible to design test cases that will also cover them.

As Figure 6 shows, providing models that include functional and non-functional aspects makes it possible to build a test model to cover those aspects, verifying them against the

system. The final result should be a set of test artifacts capable of: (1) executing the desired tests against the SUT and (2) providing a verdict.

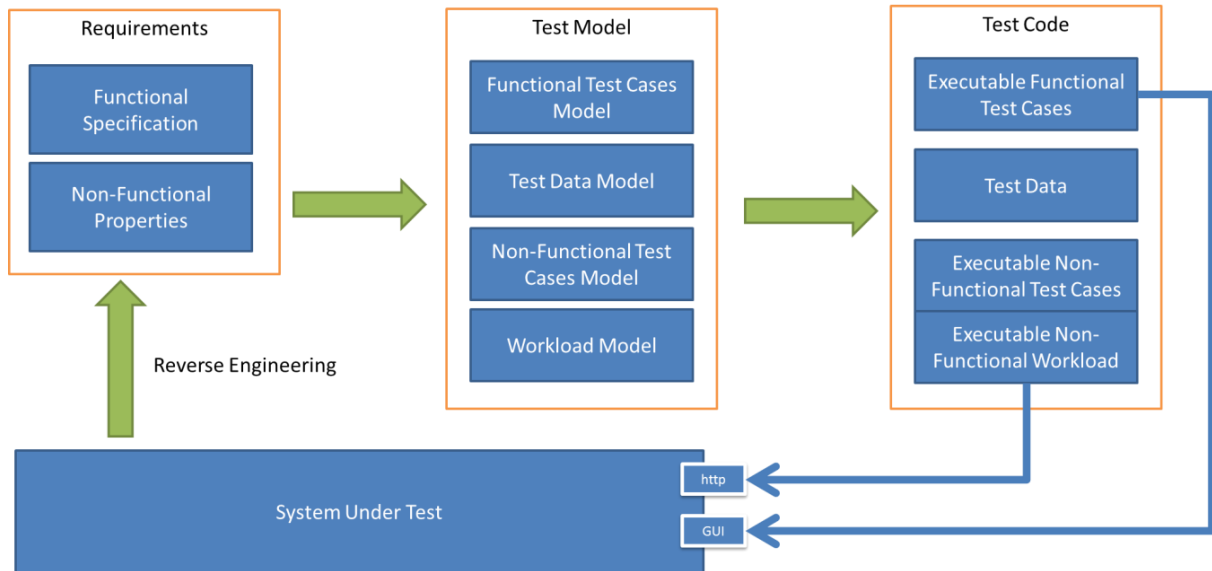


FIGURE 6 - GENERAL MODEL-BASED APPROACH

As can be seen in the figure, the functional specification and non-functional properties are part of the requirements, but also could be instantiated from the SUT using reverse engineering techniques, helping the user to at least have a first version of these models.

In addition, there are plenty of standards for model based approaches, mainly provided, organized and validated by the *Object Management Group* (OMG)⁵. This thus simplifies the construction of tools, making use of modeling tools which also complying to the standards.

1.4. EXPECTED CONTRIBUTIONS

As Dias Nieto et al. say [6], it is necessary to transmit more model-based testing practices to the industry, taking advantage of the development models for testing proposes, in order to augment the productivity of the testing team. There are many interesting proposals in the academy that are often not transferred to the companies. Our intention is therefore to institutionalize all our research contributing economic benefits and an improvement in the product quality to the industry.

⁵ OMG: <http://www.omg.org>

Since the beginning of the thesis, the focus was to improve the real practice of testing in the industry. The author of this thesis is a founder partner of Abstracta (www.abstracta.com.uy), a Uruguayan company dedicated to the provision of software testing services and the development of testing tools. As a result, an implicit requirement of this thesis was to use all research in the industry field, in order to improve effectiveness, performance and cost-benefit and increase the knowledge of the company in the testing field.

Considering the company's field of action, the expected contributions were:

- Reduce cost of functional testing for applications that make use of databases
- Reduce cost of performance testing for web applications that make use of databases

The main product of Abstracta is a tool for automated software testing called GXtest (gxtest.abstracta.com.uy). GXtest is specifically built for a model-driven development tool called GeneXus (www.genexus.com), which is developed by Artech (www.artech.com.uy). Even though the research of the PhD thesis is not limited to this environment, the idea of Abstracta is to progressively include all the promising results of the thesis into its product. At the moment, the strongest limitation of GXtest is that it is specific for GeneXus but, thanks to this, it can automate many more characteristics than if it were generic. The tool is therefore not going to incorporate all the research results directly: instead, they are adapted to the specific context of GeneXus and GXtest environment.

Actually, as will be presented as a conclusion of this thesis, there are some limitations and complications to the current state of model-driven approaches and standard technologies. These limitations were determinants of the decision about adaptation of the generic framework to the GeneXus environment. As it was determined that the current state of the technologies is not enough to build software, the test case generator for GeneXus was developed with C# and with a proprietary metamodel stored in the database.

One of the final goals of this research is also the implementation of a generic framework to automate functional and non-functional tests in information system contexts, based (when possible) on OMG standards, the validity of the proposed techniques is, in the first instance, tested with pilot studies which are controlled by the researcher in the laboratory, and then tailored to GXtest and included in different versions of the tool. This tailoring is required to allow the industrial partner to continue their usual business activity. During the research period, different versions of GXtest were released, including part of the generated knowledge, and this will be described in the conclusions (Chapter 8).

On the other hand, the main focus is on web platforms. In recent decades web-based applications that use databases have been one of the most widely used types of software, and they have become the backbone of many e-commerce and other businesses, motivating their precise validation. More than 90% of Abstracta's projects are involved with this kind of system, which is a very important reason for guiding the research in that direction.

1.5. RESEARCH METHODS

In this thesis, the different parts of the global methodology and the different components of the framework were developed using an iterative and incremental approach. In each iteration the methodology was improved and extended for the next iteration.

Seven iterations were conducted (see Table 4). The researchers were the author and the supervisors of this thesis (referenced in the table as *Alarcos Group*); other researchers contributed to some of the iterations, as for example Jesús Núñez, who contributed as a part of his *Proyecto de Fin de Carrera* (final degree project), *Consiglio Nazionale delle Ricerche* (CNR), from Pisa, Italy, where a short research stage was done, and Abstracta, from Montevideo, Uruguay, where two other short research stages took place. The different components, the results of the different iterations, will be introduced in the different chapters of this thesis. The research methods used were Proof of Concept (PoC) and Research-Action (RA), which will be explained in sections 1.5.1 and 1.5.2.

TABLE 4 - RESEARCH METHODS

Subject	Framework and methodology	Coverage criteria	DBesTest	UML-TP extension	PMM Coverage	GXtest Generator	GXtest for Performance
Researcher	Alarcos Group	Alarcos Group, Abstracta	Alarcos Group, Jesús Núñez	Alarcos Group, CNR	Alarcos Group, CNR	Alarcos Group, Abstracta	Alarcos Group, Abstracta
Method	PoC	PoC, AR	PoC	PoC	PoC	AR	AR
Researcher and Object	AjaxSample	AjaxSample	AjaxSample	Bookstore system	Bookstore system	2 industrial studies	5 industrial studies
Place	Ciudad Real, Spain	Ciudad Real, Spain and Montevideo, Uruguay	Ciudad Real, Spain	Pisa, Italy	Pisa, Italy	Montevideo, Uruguay	Montevideo, Uruguay
Year	2011	2011-2012	2011-2013	2013	2013	2011-2013	2012

1.5.1. PROOF OF CONCEPT

A proof of concept is an opportunity to demonstrate the possibilities of a proposal for a small application environment in a controlled manner in order to demonstrate its feasibility. It is an excellent risk mitigation strategy for any kind of initiative. It helps to

determine whether the software/approach/idea/tool, etc., is appropriated for use, to verify its technical feasibility, how easily it can be configured and to start to determine the benefits and limitations.

1.5.2. ACTION-RESEARCH WITH ABSTRACTA

Action-Research (AR) is a qualitative research method that brings theory and practice, and researchers and practitioners, together to solve a problem. AR is an iterative process oriented towards the progressive addition of new research knowledge entailing benefit for the research stakeholder [7], [8].

The roles involved in AR are:

- *Researcher*: person or group of people who actively carry out the research process
- *Researched object*: the problem to solve
- *Critical reference group*: group on which research is performed inasmuch as it has a problem that needs to be solved
- *Stakeholder*: anyone else who can benefit from the research but does not directly participate in it [9].

The author of this thesis belongs to Abstracta and is completely interested in applying the research results to the commercialized tools. In this sense, the thesis's author has undertaken the AR role of *Researcher*; Abstracta is the *Critical Reference Group*, received the research results, adapting them to its specific context, obtaining the researcher's assessment and returning feedback about the actual application of techniques. Also, in this context several stakeholders were identified:

- i) Abstracta team: the company provides tools and services related to software testing. The most common services are functional tests, automation and performance testing. Developing tools and methodologies to reduce costs and improve the quality of the services is directly beneficial for the company.
- ii) Artech: the company who develops GeneXus. Their interest in GXtest arises mainly because it gives their customers the ability to increase the quality of the applications they develop, so that their customers will be more satisfied with the GeneXus environment.

- iii) GeneXus Community: there are more than 100.000 developers comprising the community of users of this tool⁶, who are going to benefit from the improvements on the tools and techniques, useful for them when ensuring quality of the products they develop.

AR is an iterative process with four stages. The first is *planning*, in which researchers define research question and goals. The second is *action*, in which researchers conduct fieldwork according to their goals and research questions in real environments. The third stage is *observation*, in which researchers collect the data obtained in the action stage. Finally, the last stage is *reflection*, in which researchers analyze the results observed as a result of their actions. The reflection stage leads to the generation of new knowledge, with which the research customers are provided, and which is used to redefine goals in the next iteration.

1.6. CONTEXT

This thesis was developed in the Alarcos Research Group of Castilla-La Mancha University, in Ciudad Real, Spain, from February 2011 to December 2013. The researcher has a grant from the *Agencia Nacional de Investigación e Innovación*⁷ (ANII; BE_POS_2010_1_2360), from the Uruguayan government, and was involved in a variety of research projects, as shown in Table 5 and Figure 7.

⁶ According to the information published by Artech in 2012 there are only 85 thousands developers (see www.genexus.com/files/genexus-facts-sheet?es), and it was updated during a presentation at the last GeneXus International Meeting in Montevideo (www.genexus.com/encuentro2013).

⁷ ANII: www.anii.org.uy

TABLE 5 - RESEARCH PROJECTS

GEODAS	
Project Title	GEODAS (GEstión para el Desarrollo globAl de Software mediante Ingeniería de Negocio y Entornos Avanzados de Colaboración)
Financed by	Ministerio de Economía y Competitividad (TIN2012-37493-CO3-01)
Participants	Universidad de Castilla-La Mancha (Spain), Universidad de Murcia (Spain), Universidad de Alicante (Spain)
Duration	From: 01/01/2013 to: 31/12/2015
Main researcher	Mario Piattini Velthuis
PEGASO/MAGO	
Project Title	PEGASO/MAGO (coordinated projects, PEGASO: Procesos para la mEjora del desarrollo GlobAl del Software /MAGO: Mejora Avanzada de procesos software GLObales)
Financed by	Ministerio de Ciencia e Innovación (TIN2009-13718-CO2-01)
Participants	Universidad de Castilla-La Mancha (Spain), Universidad de Murcia (Spain)
Duration	From: 01/01/2010 to: 31/12/2013
Main researcher	Mario Piattini Velthuis
DIMITRI	
Project Title	DIMITRI (Desarrollo e Implantación de Metodologías y Tecnologías de Testing)
Financed by	Ministerio De Ciencia e Innovación (TRACE) (TRA2009_0131)
Participants	KYBELE CONSULTING (Spain)
Duration	From: 01/03/2010 to: 29/02/2012
Main researcher	Macario Polo Usaola

The three research stages shown in Figure 7 are:

- October 2012: Abstracta, Montevideo, Uruguay.
- April 2013: Abstracta, Montevideo, Uruguay.
- July-October 2013: Consiglio Nazionale delle Ricerche, Pisa, Italy.

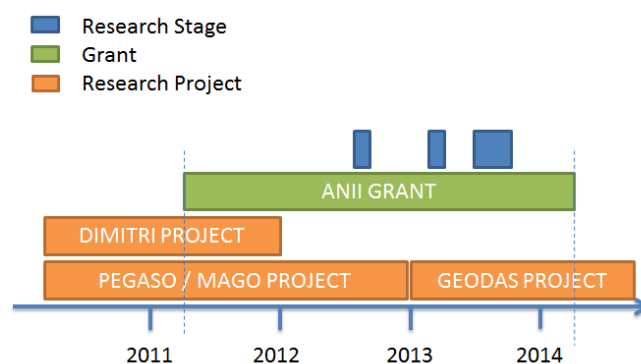


FIGURE 7 – GRANT, RESEARCH STAGES AND PROJECT PARTICIPATION

1.7. STRUCTURE OF THE DOCUMENT

The document is organized in eight chapters, as Figure 8 depicts.

Chapter 2: State of the Art and State of Practice presents the state of the art in the areas related to the research topic, as well as the state of the art of the practice of the corresponding activities.

Chapter 3: General Methodology introduces the general framework proposed as a solution in this thesis.

Chapter 4: Information System Model Construction explains the first part of the proposal corresponding to the reverse engineering applied with the objective of generating a first version of the Information System Model.

Chapter 5: Automatic Generation of Functional Test Cases presents the approach to generating the functional test cases and representing them in the test model.

Chapter 6: Automatic Generation of Performance Tests completes the presentation of the proposal including the non-functional aspects that were taken into account.

Chapter 7: Current State of the Implementation and Automation describes the implementation of the prototypes and the tools developed in Abstracta which empirically validate the proposal.

Chapter 8: Conclusions and Future Research Lines summarizes the main contributions of this thesis and proposes possible future research lines.

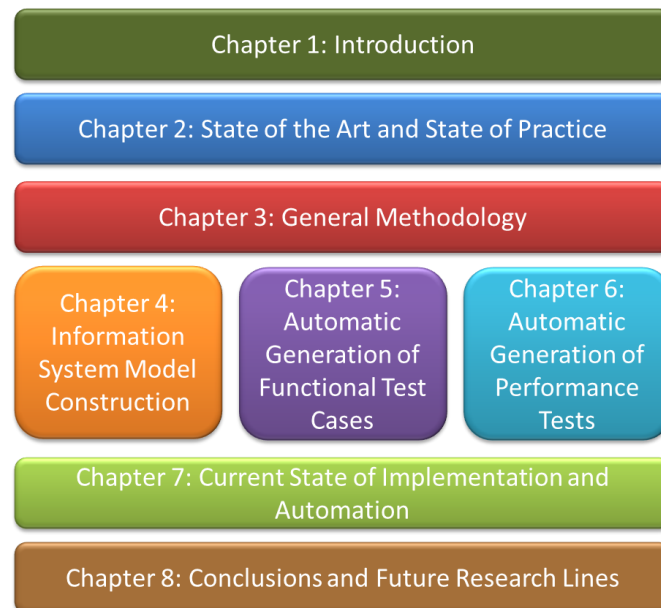


FIGURE 8 - THESIS STRUCTURE

Utopía

Ella está en el horizonte.

Me acerco dos pasos, ella se aleja dos pasos más.

Camino diez pasos y el horizonte se corre diez pasos más allá.

Por mucho que yo camine nunca la voy a alcanzar.

¿Para qué sirve la utopía?

Sirve para eso: para caminar.

— Eduardo Galeano

“Utopia lies at the horizon.

When I draw nearer by two steps, it retreats two steps.

If I proceed ten steps forward, it swiftly slips ten steps ahead.

No matter how far I go, I can never reach it.

What, then, is the purpose of Utopia?

This is its purpose: It is to cause us to advance.”

— Eduardo Galeano

CHAPTER 2. STATE OF THE ART AND STATE OF PRACTICE

This chapter introduces the fundamental concepts of this thesis, and the state of the art and practice for the different artifacts and methods involved.

2.1. INTRODUCTION

From the example introduced in Chapter 1 some potentially automatable points were identified and they are presented in Figure 9.

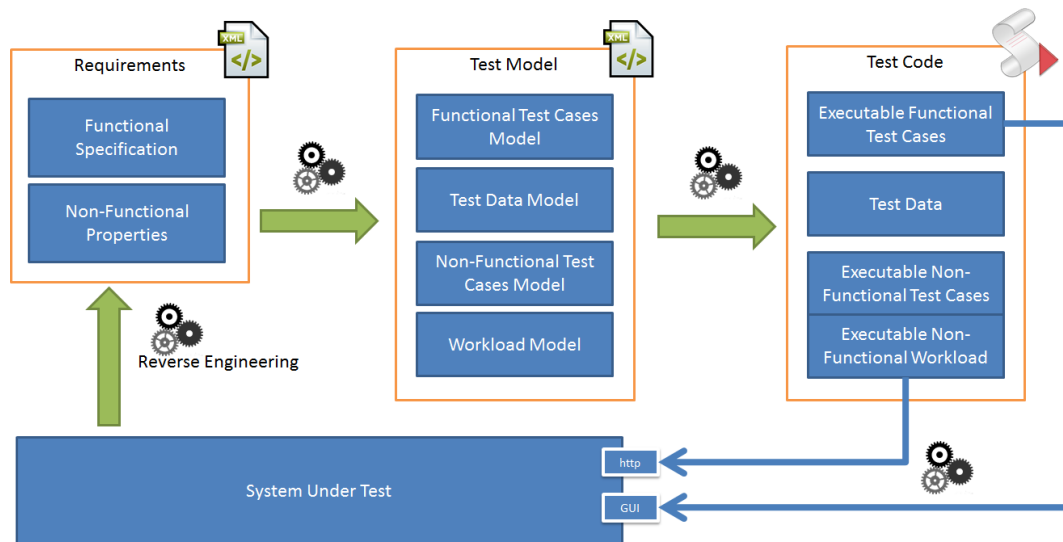


FIGURE 9 - GENERAL PROCESS

These points are:

- Reverse engineering process (data model and system model)
- Test model generation
- Functional test case execution

- Test code generation
- Performance test case generation
- Performance test case execution

This section is structured according to the different artifacts and the different automation mechanisms of the proposal, making reference to its correspondence to Figure 9, showing the same figure in order to let the reader clearly visualize the part of the whole process to which it corresponds.

2.2. MODELS AND METAMODELS

This section presents the different possibilities for the representation of the artifacts involved, from the Information System models to the representation of tests, both functional and non-functional. Basic related concepts are presented, as well as model transformations and extension mechanisms of the metamodels.

Before the OMG standardization efforts related to Model-Driven Architecture (MDA) [10], researchers were prolific in proposing their own metamodels and different mechanisms for model transformation, code generation and model generation from code (reverse engineering). Even though some of the techniques proposed were very powerful, an important drawback was the impossibility of sharing models between different tools. In software engineering, standardization enables the definition of a common language and the possibility of interchanging information between different platforms. This was, for example, one of the primary goals of web services: the possibility of invoking remote operations, independently of the technology used in the implementation of clients and servers. The definition of a standard structure for the byte strings that should travel from one to another machine was thus required, so defining the SOAP protocol. In the case of the Model-Driven paradigm, the OMG proposed the MDA, which was defined based on other OMG standards such as the UML [11] (explained later in this chapter), the XML Metadata Interchange (XMI) [12] and the Meta Object Facility (MOF) [13].

The three primary goals of MDA are portability, interoperability and reusability through architectural separation of concerns. MDA provides an approach for: (i) specifying a system independently of the platform that supports it, (ii) specifying platforms, (iii) choosing a particular platform for the system, and (iv) transforming the system specification into one for a particular platform. For this, MDA specifies three models for a system [14]:

- The Computation Independent Model (CIM) focuses on the environment of the system and on its requirements. Details about its structure and processing are hidden or undetermined.
- The Platform Independent Model (PIM), which is a formal specification of the structure and function of a system that abstracts away technical details. A platform-independent view of a system shows the parts of the complete specification that do not change from one platform to another.
- The Platform Specific Model (PSM) completes the PIM with the details required for obtaining the modeled system for a specific platform.

Setting aside the very high abstraction level of the CIM, in an MDA context, the software engineer may draw the structure and behavior of a system using UML diagrams in the form of a PIM: with no details about the final platform where the application will run. Here there is enough information to derive test cases for any platform, and then, adding the PSM information, those test cases could be used to generate executable test code on the specific platform.

2.2.1. UNIFIED MODELING LANGUAGE (UML)

The use of standards is essential to allow the adoption of new technologies by industry. UML is the most widespread modeling language to represent high-level views of software systems, used in the software development process. The objective of UML is to provide the different members of a development team with tools for the analysis, design, implementation and testing of software systems. Thus, it is a common language between analysts, designers, developers and testers.

2.2.1.1. TOOLS AND GRAPHICAL EDITORS

There are several tools that allow us to model with UML and its satellite technologies, both commercial and open. One of the most important projects is the Eclipse Modeling Project (<http://www.eclipse.org/modeling/>) which extends the popular and powerful Eclipse IDE. Eclipse Modeling allows UML and other kind of models to be dealt with. It is even possible to define your metamodels and build a tool, based on Eclipse, to manipulate them.

With this set of tools it is also possible to create and manage models programmatically which is really important when trying to extract information from different sources, without the necessity of manually writing the XML.

As a drawback, it is important to mention that in practice UML is excessive and too complex, and that there are different implementations of the standard that usually

make it impossible to operate between the different tools. In Chapter 7, there is a more detailed analysis of these issues within the implementation details of the solution.

2.2.1.2. EXTENSION MECHANISMS

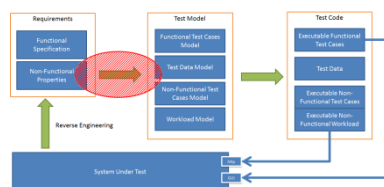
The most common and practical way to extend the expressiveness of UML is through the use of *UML Profiles*. The UML Profile mechanism includes the ability to tailor the UML metamodel by defining domain specific languages by means of *stereotypes*, *tagged values* definitions, and *constraints* which are applied to specific model elements. This mechanism gives more semantic strength to the UML standard elements. For example, in a class diagram, it is possible to represent different *kinds* of classes, according to the stereotype added to them, or by adding special values to any element with tagged values.

Recently, Eclipse has a new project to store the most common UML Profiles, accessible here: <http://www.eclipse.org/proposals/modeling.uml2profiles/>. For example, there are (or will be) profiles for data modeling, testing (the UML Testing Profile), service oriented architecture (SoaML), business processes (BPMN), etc. The main purpose is to provide a central repository within the Eclipse Modeling project to discover and install UML profile implementations, promoting interoperability. Regrettably, this is not yet implemented, and there has been no activity registered since July 2012.

2.2.2. MODEL TRANSFORMATIONS

Model-based Testing (MBT) provides techniques for the automatic generation of test cases by using models. Model-driven Testing (MDT) is an MBT approach, where the test cases are automatically generated using models extracted from software artifacts through model transformations. MBT plays an important role in the software validation process [15], which contributes to test automation by pushing the application of model based design techniques to software testing. It involves the development of models that describe test cases, test data and the test execution environment. It also includes the application of automated facilities for generating executable test cases from these models. A key element in MBT is the modeling language used for defining a test model from the informal system requirements or the design models. For testing purposes, it is important to have a test model that is easy to verify, modify and manipulate without losing all the information needed to generate test cases. Miller et al. [6] present a systematic literature review of MBT. A result of this review is that it was seen that by using models developed from the analysis of the abstract behavior of the SUT, MBT has been traditionally used for generating functional test cases, but has missed addressing non-functional requirements. This thesis especially considers UML models.

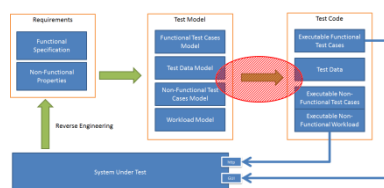
Standard UML models can be serialized and represented in XMI [12], an XML-based language that allows the representation of any UML diagram. Any programmer may write a set of routines in their favorite programming language for reading the XMI code (maybe making use of a DOM or SAX library) and, so, make any type of manipulation with the diagrams (for example, to generate a PSM adding information to a PIM). However, it would be tedious to write such a program and, would probably only be written for a single functionality. Fortunately, the OMG has also considered this kind of task and has developed *Query/View/Transformation* (QVT) [16] and the *MOF Model to Text* (MOFM2T) [17], which are two standard languages for: (1) transforming models into other models; and (2) transforming models into text.



Using QVT, it is possible to throw queries against models, and of course, to perform model transformations within models. One of the most common operations in QVT is pattern-matching. For any operation, QVT expresses models as search patterns. Many implementations of the standard can be used, such as MediniQVT [18] or ATL [19].

QVT uses the following definitions:

- *Queries* are expressions that are evaluated in a model. They result in one or more instances of types being defined in the source model (i.e., “give me all the states in the state machine”), or defined by the query language.
- *Views* are models derived completely from other base models. A view cannot be modified separately from the model from which it is derived, and changes to the base model cause corresponding changes to the view. Typically, the metamodel of the view is not the same as the metamodel of the source (i.e., a class diagram is translated into a relational database diagram, or vice versa). Views are generated via transformations.
- A *Transformation* generates a target model from a source model. A model transformation may also have several source models and several target models: through queries, the transformation obtains the elements of the source model that must be transformed, and produces views which correspond to that targeted.



In addition, MOFM2T provides a model-to-text transformation language [17]. Its goal is to define a language to facilitate the generation of code or documentation from models. There are several tools following this approach, such as MOFScript [20] or

Acceleo [21], which are pragmatic implementations of the standard.

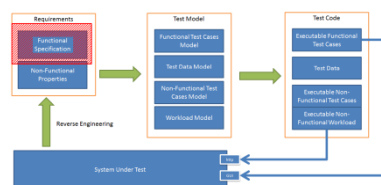
The model to text transformation mechanism uses text templates that use placeholders to represent the interesting data that must be transformed into text. Essentially, these placeholders are expressions specified over metamodel entities. A query language is the primary mechanism for selecting and extracting the values from models. These values are then converted into text fragments using an expression language augmented with a string manipulation library. Templates can be composed to address complex transformation requirements. Moreover, the language allows the structuring of large transformations into modules.

More detailed information about the different tools for both model-to-model and model-to-text, is presented in Chapter 7 within the implementation details about the solution. Some of the limitations and difficulties of using these technologies are depicted, especially in reference to the supporting tools. For instance, MediniQVT, which is the official implementation of the standard, is no longer maintained, nor is MOFScript. In the case of model-to-model transformations, ATL became the standard *de facto*, and MOFScript has been effectively substituted by Acceleo. Even though these tools and approaches are very promising, the current state and maturity of the tools (they are hard to debug, there are many errors and not much support) makes the adoption of the technology very difficult.

2.2.3. STATE OF THE ART IN INFORMATION SYSTEM MODELS

One of the main goals of this thesis is to generate executable test cases. Thus, an analysis determined which elements were necessary for inclusion in the system model. The intention was to keep the model as simple as possible, taking only the necessary information for the automatic generation of executable test cases. This section is divided into two parts, one for functional specification and one for non-functional, because they are typically considered separately.

2.2.3.1. FUNCTIONAL SPECIFICATION



UML is a general purpose language, so there was a especial interest to find any extension, including specific concepts for the kind of applications and components that this thesis addresses, such as the database structure and the graphic user interface. By

undertaking a first analysis of the test case generation necessities, it was determined that the functional specification should include: the data model (as the basis of the test cases design), the graphic user interface model (structure and navigation, in order to

have information to generate executable test cases stimulating the SUT at a graphic user interface level) and the business rules (to know the expected behavior of the SUT). For each view of the system a review of the state of the art was performed.

Systems Modeling Language (SysML) [22]–[24] seemed to be a good option for the functional IS specification, being the OMG standard modeling language for specifying, analyzing, designing and verifying systems. It is a UML Profile, but uses only a subset of UML capabilities. As it does not provide anything special for representing data models or graphic user interface models and includes other aspects that are not going to be taken into account (such as hardware, software, information, personnel, procedures, and facilities), it is clear that it does not fit with our modeling requirements.

Data Metamodels: Regrettably, there is no standard for data modeling. Researchers were prolific in proposing their own metamodels to use as an input for MDE techniques. Many UML vendors and users use UML tools for data modeling and ended up defining their own UML profiles. In December 2005 the OMG issued a *Request for Proposals* (RFP) [25].

According to the experiments presented in [26] and [27], UML class diagrams have the same expressiveness as Entity-Relational diagrams for data modeling, and they are better for verification activities.

After having analyzed several models ([28], [29], [30] and [31]) it was decided to use the one proposed by IBM, the UML Data Modeling Profile (UDMP) [32]. This profile is one of those presented to the aforementioned Data Modeling RFP. Something very important for our work is that the proposals are not vastly different: all consider the main aspects of a database structure, and they represent most of the concepts in a similar way, and with very similar names. This is probably because database research has existed for many years, and it is a relatively standardized world.

The UDMP is a UML extension to support the modeling of relational databases with UML, including extensions to represent schemas, tables, views, columns, keys, triggers and more. Chapter 4 explains it in a more detailed manner.

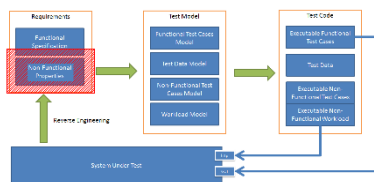
GUI Metamodels: as the ultimate utility of the Information System Model is to generate executable test cases, it was determined that the GUI metamodel should at least be able to represent: the structure (pages and page elements such as inputs and buttons) and navigation. This information is required to interact with the SUT at a GUI level.

Most of systems for GUI metamodels have been provided via several methodologies for application development. Some of these metamodels have been analyzed, such as *Relational Management Data Model* (RMDM) [33], the *Object Oriented Hypermedia* (OOH) [34] [35], the *Web Modeling Language* (WEBML) [36] and the UML Profile

presented in [37]. Each one is defined for a different focus, they are more complex than necessary, and none of them is considered standard or well adopted. Taking this into account, a simpler metamodel is proposed, which is presented in Chapter 4.

Business Rule Metamodels: Even though there some other proposals were analyzed, none were as tightly integrated to UML as the Object Constraint Language (OCL) [38]. For instance, Halpin [39] proposes the use of Object Role Modeling (ORM) to expand the expressiveness of UML with business rules. Other authors ([40]–[42]) used Alloy [43] to express rules for the models with the possibility of using a constraint solver with them. This rule language is declarative and based on Z notation [44].

2.2.3.2. NON-FUNCTIONAL SPECIFICATION



Several proposals exist for modeling scheduling, performance, time and other non-functional aspects:

- UML-SPT: *Schedulability, Performance and Time Profile* was the first OMG standard for this aim, modeled as a UML Profile.
- MARTE: *Modeling and Analysis of Real-Time and Embedded Systems* [45] is an evolution of UML-SPT, which is now the standard. MARTE [45] extends UML by providing a rich framework of concepts and constructs to model the non-functional properties of real-time and embedded systems and defines annotations to augment models with information required to perform quantitative predictions and analysis of time-related aspects, such as schedulability and performance.
- UML4SOA-NFP [46], which is a UML4SOA Profile taking into account non-functional properties, is used together with the UML MARTE Profile to represent performance, security and dependability properties for service-oriented systems.

The main limitation of such profiles is that they do not provide support for testing.

Another proposal to specify non-functional properties is the *Property Meta Model* (PMM) [47]–[49]. It is not based on UML, but was specially considered since this aspect of the proposal was integrated in this thesis in the research stage in CNR, Pisa, Italy, where they developed this metamodel.

PMM is a generic, comprehensive and flexible metamodel for defining non-functional properties spanning dependability, performance and security. It is made available to the

community for adoption, validation, and possibly for extension⁸. The metamodel is implemented as an Ecore model and comes with an associated editor realized as an Eclipse Plugin.

PMM allows for specifying metrics and provides a machine computable specification language (included into PMM, but also usable in isolation) that allows for defining complex events models involved into non-functional properties. The proposed language improves over existing complex event specification languages (such as GEM and Drools Fusion) by adding new features not included in the existing languages.

Figure 10 sketches the key concepts and their relationships in this metamodel: *Property*, *MetricsTemplate*, *Metrics*, *EventSet*, and *EventType*.

The definition of properties and metrics is independent from the application domain. The defined properties represent quantitative and qualitative properties that a generic software system or its components may expose (descriptive properties), or must provide (prescriptive properties). The concepts and terms belonging to an application domain are linked to PMM via the *EventType* and *Action* entities, which model a generic observed event or an atomic action, respectively, of the application domain in which the system will be used.

Thus, there is a distinction between a generic metrics formula (represented by a *MetricsTemplate*) and concrete metrics (i.e., the *Metrics*) that is instantiated in a specific application domain by means of the *EventSet* and *EventType* concepts. This approach makes the metamodel more generic, since the same template can be used (instantiated) in different scenarios.

⁸ A release of the Property Metamodel and the associated editor is available at <http://labse.isti.cnr.it/tools/pmm>.

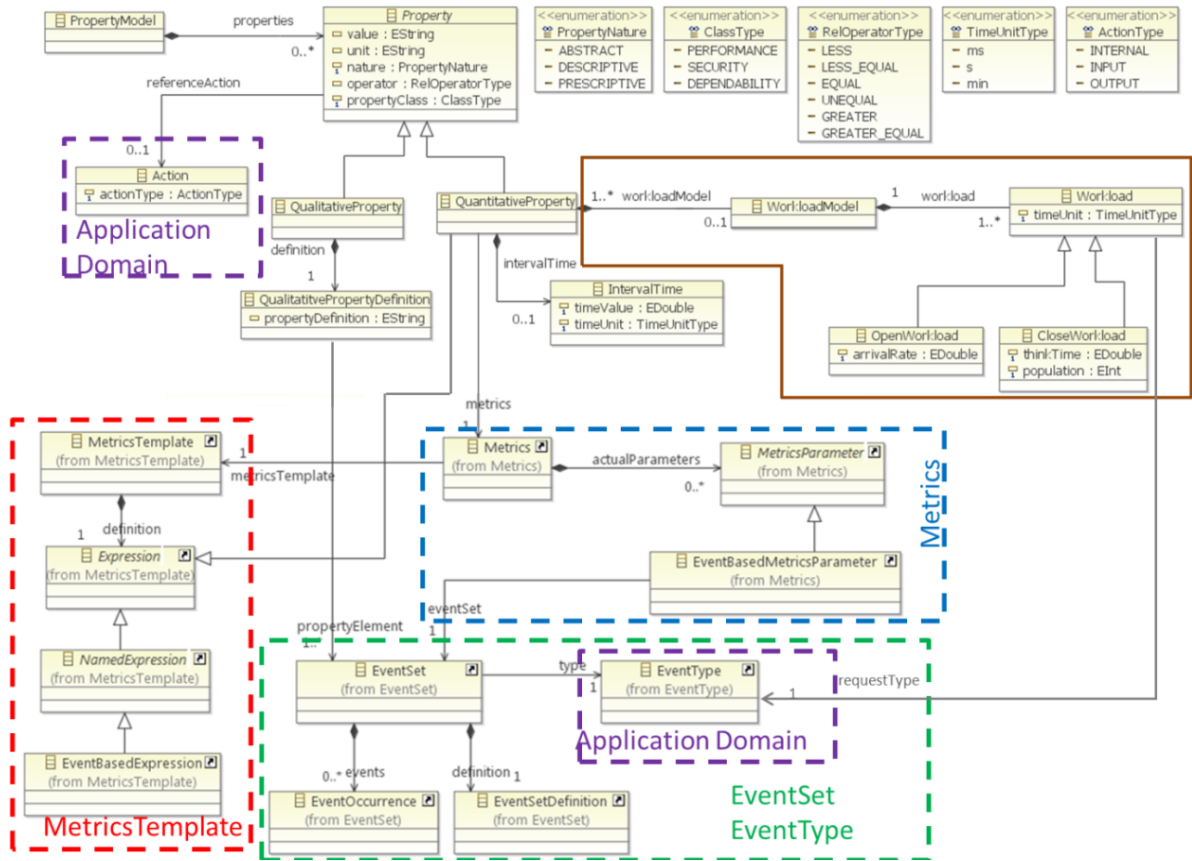
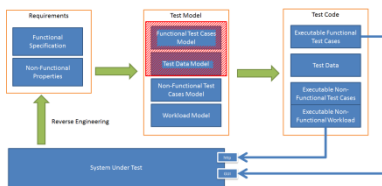


FIGURE 10 - PMM METAMODEL

2.2.4. STATE OF THE ART IN FUNCTIONAL TESTING MODELING



Software testing is a process designed to make sure computer code does what it was designed to do and that it does not do anything unintended [4].

In general, testing activities involve planning, test case generation, test environment set-up, execution, test result evaluation, test reporting, and defect tracking [50]. The main concepts related to test case generation are: the test case, test procedure and test oracle.

A *test case* contains a set of inputs, execution conditions and expected results which are developed for testing a particular system’s objectives (such as exercising a given path or verifying the compliance of a specific requirement) [51]. A *test procedure* contains instructions for the set-up, execution and evaluation of results for a given test case [51]. The result obtained from the test case is compared with the expected result through an oracle. A *test oracle* is any agent (human or mechanical), which decides whether a

program behaved correctly in a given test, and accordingly assigns a verdict of “pass” or “fail” [50].

There are few proposals for test specifications, and the most important and widespread are *UML Testing Profile* (UML-TP) and the *Testing and Test Control Notation version 3* (TTCN-3).

The **UML-TP** [52], [53] is the OMG standard for test modeling, implemented as a UML Profile. Its utility has been proved via application in different MDT contexts [54]. Figure 11 shows an excerpt of the UML-TP metamodel for test architecture and test behavior concepts (taken from the specification [53]).

UML-TP is a lightweight extension of UML with specific concepts (stereotypes) for testing, grouped into: i) test architecture; ii) test data; iii) test behavior; and iv) test time. This extension fills the gap between system design and testing, allowing the users to have a unified model for both aspects of the development of a software product.

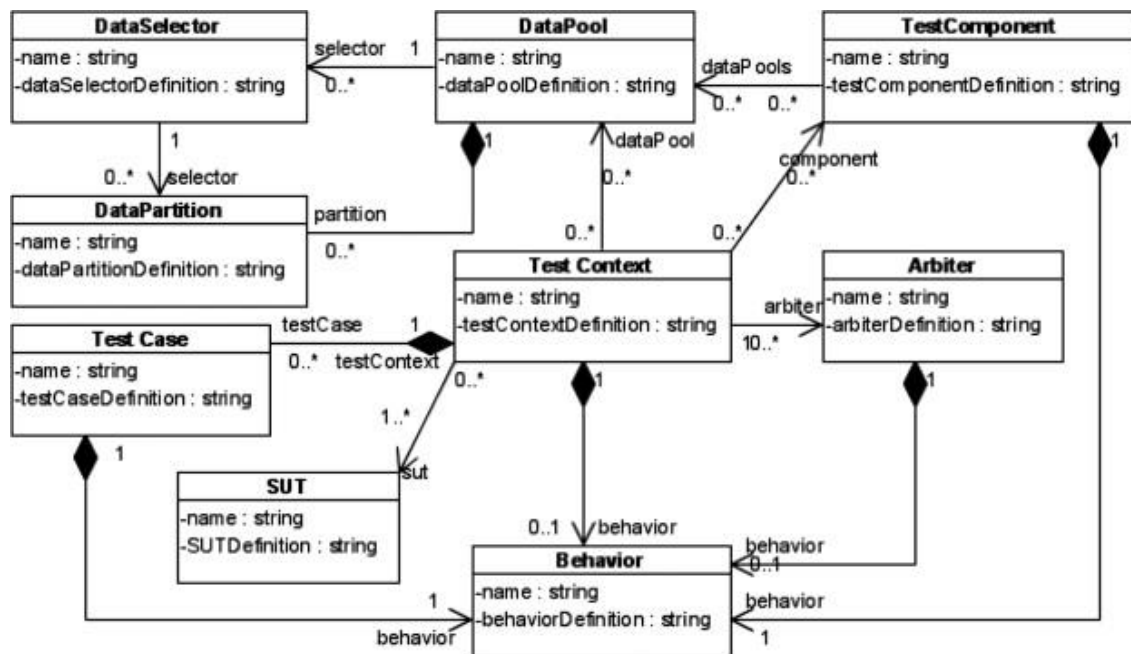


FIGURE 11 - UML-TP METAMODEL

The test architecture provides all the elements that are needed to define the test cases. Specifically, it includes the set of concepts to specify the structural aspects of the test. Some of these are: i) the *Test Context*, which groups the *Test Cases*; and ii) the *Test Components*, which are responsible of the communication with the *SUT*.

The main constructor is the *Test Case*, whose behavior can be described by sequence diagrams, state machines or activity diagrams. In UML-TP, the *Test Case* is an operation

of a *Test Context* that specifies how a set of *Test Components* cooperates with the *SUT* to achieve the *Test Objective*, and to provide a *Verdict*. The behavior can be enriched with *Timer* restrictions. Finally, another important aspect of the test specification is the test data. It is possible to model different *Data Partitions* that are obtained from a *Datapool* through *Data Selector* methods. It is also possible to enrich these definitions with the use of *Wildcards* and *Coding Rules*.

Generally, a UML-TP model is presented through different diagrams:

- There is a UML package diagram representing the test architecture, and showing how the test package uses a test data package and includes the *SUT* model in order to allow the test elements to access the different elements under test.
- A class diagram could be used to show the structure of the test package, showing how the *Test Context* is related to the different *Test Components*, *Datapools*, and *SUT* components that are going to be exercised in the test. This class diagram should also model the *Test Cases* as methods of the different *Test Contexts*, in this way representing the complete test suite.
- There could be also a composite structure diagram of the *Test Context* class to show its *Test Configuration*, describing the relationships between the *SUT* and the *Test Components* for this *Test Context*.
- Finally, each *Test Case* behavior is presented through any kind of UML behavior diagram, such as a *State Machine Diagram*, *Sequence Diagram* or *Activity Diagram*.

UML-TP provides a mechanism of “*default behaviors*”, for example those for the *Arbiter* and for the *Test Scheduler*. If the modeler/tester wants a different behavior for them, it is also necessary to provide an extra behavior diagram, mentioned in the standard as “*user-defined behavior*”.

The **TTCN-3** [55], [56] is a strongly typed test scripting language for testing reactive systems. It was developed and standardized by the *European Telecommunications Standards Institute* (ETSI⁹) and the *International Telecommunication Union* (ITU¹⁰).

Although primarily used in telecommunications, TTCN-3 is well accepted by the industry and its use has spread into new domains including automotive, railway and financial

⁹ ETSI: <http://www.etsi.org>

¹⁰ ITU: <http://www.itu.int>

applications. Figure 12 shows an excerpt from an example script taken from the tutorial, to specify the test of a coffee machine.

```

function CoffeeMachineFunction() runs on CoffeeMachineComponentType
{
    const integer Price := 50;
    var integer Amount, Cents;
    Amount := 0;
    while (true) {
        InputPort.receive(integer:?) -> value Cents;
        Amount := Amount+Cents;
        while (Amount >= Price) {
            OutputPort.send(charstring:"coffee");
            Amount := Amount-Price;
        }
    }
}

function CoffeeDrinkerFunction() runs on CoffeeDrinkerComponentType
{
    var integer Count;
    OutputPort.send(100);
    Count := 0;
    timer t;
    t.start(5.0);
    alt {
        [] InputPort.receive(charstring:"coffee") {
            Count := Count+1;
            repeat;
        }
        [] t.timeout {
        }
    }
    log("Received " & int2str(Count) & " cup of coffee.");
    if (Count == 2) {
        setverdict(pass);
    }
    else {
        setverdict(fail);
    }
}

testcase TwoCoffeesPlease () runs on EmptyComponentType
{
    var CoffeeMachineComponentType CoffeeMachine;
    var CoffeeDrinkerComponentType CoffeeDrinker;
    CoffeeMachine := CoffeeMachineComponentType.create;
    CoffeeDrinker := CoffeeDrinkerComponentType.create;
    connect(CoffeeDrinker:OutputPort, CoffeeMachine:InputPort);
    connect(CoffeeDrinker:InputPort, CoffeeMachine:OutputPort);
    CoffeeMachine.start( CoffeeMachineFunction() );
    CoffeeDrinker.start( CoffeeDrinkerFunction() );
    timer t; t.start(6.0); t.timeout;
    CoffeeMachine.stop;
}

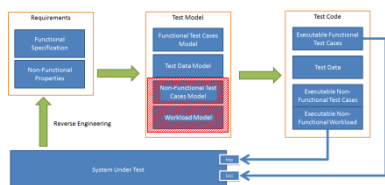
```

FIGURE 12 - TTCN-3 EXAMPLE SCRIPT

If the tester provides an adaptation layer for connecting the test script commands with the specific execution method on the SUT, then the test script can be compiled and executed.

There are some proposals to translate UML-TP models into TTCN-3 scripts ([52], [57], [58]).

2.2.5. STATE OF THE ART IN PERFORMANCE TESTING MODELING



A Performance/Load Test is defined as a research technique for determining or validating the response times, dependability, scalability and/or stability of a system under test. The tester simulates the interaction of multiple concurrent users running against the SUT

as in the production environment (noted as *scenario* or *workload*). Test results allow an analysis of how the SUT and the infrastructure behave in the presence of this load [59].

Following our experience and a careful analysis of the elements required to model a performance test, our conclusion was that it is necessary to model the workload (with all its components), test cases and test data, and the acceptance criteria for each one based on the non-functional requirements of the SUT.

UML-TP is mainly used to model functional testing whereas its application for specifying non-functional test cases is limited since it lacks facilities to address specific non-functional concepts.

This section is divided into two subsections: a study of different model-driven approaches for performance testing is presented, and then the current support of UML-TP for modeling this kind of test case, which is necessary to understand why this thesis proposes an extension.

Both aspects of this topic were developed in the research stage in the CNR, Pisa, Italy (July-October 2013).

2.2.5.1. MODEL DRIVEN APPROACHES FOR PERFORMANCE TESTING

To be aware of existing model driven approaches for performance testing, a systematic survey was performed, following the guidelines proposed by Kitchenham [60]. From this automatic search 411 papers were obtained, reduced to 25 after reading the title, keywords and abstract, and finally to 24 papers after reading the full text.

Table 6 shows the classification framework adopted in this study. The extracted data was classified according to different dimensions explained in the following and reported in the rows of the table.

Type of Non-Functional Property (NFP): Various NFPs are addressed by the different approaches. Our search string specifically pinpointed performance and dependability properties, however, the extracted works also relate to other non-functional properties such as security, usability, etc.

Instrument: This refers to the formalism used for expressing the proposal. Some authors propose UML extensions (UML profile) whilst others design their own metamodel.

Expressiveness: This represents which kinds of elements are modeled in the proposal (test cases, test environment, properties, etc.).

Testing type: This analyzes whether the proposal gives support only for non-functional testing or if it is also compatible with functional testing.

Purpose: The approaches aim to support different activities such as testing, performance prediction (known as Software Performance Engineering, SPE), benchmarking, etc. It also explains whether the approach addresses a particular technological context such as Service-Oriented Architecture (SOA), Web Systems, Embedded Software, etc.

Automation: This refers to the support provided by the approaches in terms of tools or automation facilities such as model based transformations or similar.

TABLE 6 - CLASSIFICATION OF APPROACHES FOR PERFORMANCE TESTING

Name & Ref.	Type of NFP	Instrument	Expressiveness	Testing type	Purpose	Automation
UML-TP [53], [52] *	Security, Usability, Performance, Dependability	UML Profile	Test cases and SUT, structure and behavior	Functional and Non-Functional	Testing	General UML tools. Mapping between UML-TP and JUnit and TTCN3 for execution
MDAbench [61], [62]	Performance, Resource Usage	UML and tailored UML-TP	Workload, test cases, architecture of the SUT (behavior and structure)	Non-Functional	Benchmark of system's architectures	Tool to generate the simulated architecture, the load test and monitoring collectors (with JMX)
Argo/MTE [63]*	Performance	UML	Architecture of the SUT (behavior and structure), load test parameters	Non-Functional	Design Benchmark (Client/Server)	ArgoUML for modeling, generating the simulated architecture, the load test and metrics collection
Puppet [64]	Performance	UML, WS-Agreement	Behavior and SLA of the required components	Functional and Non-Functional	Testing with stubs for SOA	Tool for generation of test beds for modeled components
WAGON [65]	Performance, Resource Usage	Own metamodel	Workload and user behavior	Non-Functional	Benchmark Web Servers	Simulated traffic
UpperT Tool [66]*	Performance	UML-SPT	Performance requirements	Non-Functional	Performance testing Web Applications	Petri Net Model, JMeter test scripts with a results interpreter
PLeTs Tool [67] *	Security, Performance	Own UML Profile	Workload	Functional and Non-Functional	Performance testing Web Applications	Generates Load Runner [68] scripts.

Name & Ref.	Type of NFP	Instrument	Expressiveness	Testing type	Purpose	Automation
Performance under stressed nets [69]	Performance	UML Profiles	SUT, users activity	Non-Functional	Performance testing distributed systems	Generates load test under data traffic stress. Does not provide a tool
SWAT[70]*	Performance	Own metamodel	Workload, interactions with data dependencies	Non-Functional	Performance testing Web Applications	Generates test scripts and data for httperf [71]
MBPeT [72], [73]*[74] *	Performance, Resource Usage	Probabilistic timed automata	Workload by user profiles (probabilistic models)	Non-Functional	Performance testing Web Applications	Execution tool
WST (Web Service Testing tool) [75]	Performance, Availability	Own metamodels	Workload, scheduling, user behavior, etc.	Non-Functional	Performance testing SOA	Execution tool
Models in performance testing [76] *	Performance	Own metamodels	Requirements, workload, SUT, measurements	Non-Functional	Performance testing Web Applications	IBM Rational modeling tools, proprietary tools
SPT for prediction [77]*	Performance	SPT- UML	Workload, SUT, performance requirements	Non-Functional	Performance prediction	General UML tools, their model transformation and analysis tools
SPE-ED [78]	Performance, Resource Usage	Own metamodel	SUT, resources, devices utilization	Non-Functional	Performance prediction	Tool for the whole methodology
PIMF [79]– [81]	Performance	Own metamodel	Workload, SUT, resources	Non-Functional	File format	None
Survey on Performance Prediction [82] [83]*	Performance	Finite state automata, sequence charts, etc.	Workload, response time, resource usage	Non-Functional	Performance prediction	Tools that partially support performance prediction in the software life cycle
Survey on search-based approaches [84] [85]*	Performance, Resource Usage, Security, Usability, Safety	None	None	Functional, Non-Functional	Test data generation by metaheuristic search techniques	None

As can be seen, most of the related works in the current state of the art focus on performance prediction (which is well studied in the survey of Balsamo [82] and Koziolok [83]), benchmarking (e.g. [61], [63]) and generation of test scripts from non-functional requirements (e.g. [66]). There is also a systematic review of search-based techniques and genetic algorithms applied for test input generation, to test non-functional properties [84].

The complete survey is published in a technical report in the digital library from CNR [86].

In the first column of Table 6 the most relevant approaches for our research topic were marked with a “*”, and they are further explained in the following in more detail, and grouped according to four main research directions:

- Benchmark generation
- Performance test generation
- Models to predict performance and Software Performance Engineering (SPE)
- Search-based testing for non-functional properties

Benchmark Generation: There are a large number of code generation techniques that can be used in benchmark suite generation. The aim of MDABench [61] is to automate the generation of a complete benchmark suite from a UML-based design description, along with a load testing suite modeled in the UML-TP.

The output of the proposed approach is a deployable benchmark suite, including the core benchmark application, load testing suite, utilities to collect performance data and configuration files for external monitoring and profiling framework. Specifically, a PIM design is annotated with three types of profiles: a platform specific profile, a performance profile and a tailored UML-TP. A benchmark suite is then generated using all the profiles for different platforms.

The load testing suite is modeled in the UML-TP that the authors have tailored to represent a workload including some tagged values such as the number of processes the load generator should start, the number of threads that each process spawns, the maximum length of time in milliseconds that each process should run, the time interval between starting or stopping new processes, etc.

The main result of the approach is thus the automated generation of the application under test including a complete test harness with the load to execute and some facilities to automatically collect monitoring information. However, as opposed to our work, the effort of the authors is more focused on automated benchmark generation than on test modeling, hence they are merely concerned with representing load tests using the tailored profile in order to produce a configuration file containing all the tagged values and deriving a default implementation of the model including both test logic and test data.

Argo/MTE [63] proposes a performance test-bed generator for industrial usage. The test bed is modeled, indicating what interactions between client and server must be simulated. The model is used to generate the code of a performance test bed able to run the specified performance tests. In comparison to our proposal, this approach does not focus on load test modeling, but aims to solve some challenges of performance test bed generation including: extending an open-source CASE tool, namely ArgoUML, to provide

UML-like architecture modeling and XMI derived model representation capabilities; restructuring and enhancement of the employed XSLT-based code generators; efficient use of tools and databases for performance test management; and result visualization.

Performance test generation: A recent research direction in performance testing is the model-based generation of test beds to assess whether the application meets its performance requirements. In particular, de Olivera et al. [66] take as input a performance specification using the SPT-UML, and derive a modified Stochastic Petri Net from which a realistic test scenario in JMeter [87] is generated. It also has a result interpreter to give a verdict. The main difference with our proposal is that this approach tries to generate the test code directly from the requirement specification without having the test model which is the objective of our work.

Puppet [67] is another solution presented in this context. Their authors propose to include five stereotypes in the system UML models (use cases and activities) to express performance information that will be used for generating test scripts for a commercial tool called LoadRunner [68]. Specifically, these stereotypes include: *PApopulation* representing the number of users and the host where the application is executed; *PAProb* representing the probability of execution for each existing activity; *PATime* representing the expected time to perform a given use case; *PAThinktime*, which is the time between two different user actions; and *PAparameters* representing the input data to be provided to the application when running the test scripts. The main limitation of this approach is that it addresses the issues of a specific performance testing tool, LoadRunner. The information about the proposed stereotypes is not enough to generate scripts for different tools.

SWAT [70] attempts to model the workload for performance test generation. It provides a tool for generation of a synthetic workload characterized by sessions of interdependent requests. From request logs for a system under test, the approach automatically creates a synthetic workload that has specific characteristics and maintains the correct inter-request dependencies. The main difference in this approach with respect to our proposal is that it pays great attention to script generation and the definition of how to take data (from an external file or from the previous response), rather than to the workload specification.

MBPeT [72], [74] is another performance tool which aims to evaluate the performance of a system, and monitor different key performance indicators (KPI) such as the response time, the mean time between failures, the throughput, etc. The tool accepts as input a set of models expressed as probabilistic timed automata, the target number of virtual users, and the duration of the test session, and will provide a test report

describing the measured KPIs. The main contribution of the paper is that the load applied to the system is generated in real time from the models.

Different metamodels related to performance testing were presented by Pozin and Galakhov [76]. These metamodels are useful for ensuring the adequacy of the results of performance testing and its parts, namely the statement of the problem, how to collect the initial data and analysis of the experimental results. Specifically, their authors propose four metamodels: a metamodel of requirements characterizing the type of the system under test and the non-functional requirements; a metamodel of the system describing the structure of the system and the configuration of the resources; a metamodel of the load describing the number and types of service requests and their distribution; and a metamodel of measurements indicating the quantities to be collected, the method of their collection and the criteria for assessing the results. The use of these metamodels in planning a new load testing experiment ensures that the resulting models are complete and integral. These models make it possible to automate the configuration of automated testing tools for the parameters of a specific load experiment.

Models to predict performance and Software Performance Engineering: An orthogonal research direction to our work is represented by model based software performance prediction. Balsamo et al. [82] present an extensive survey on methodological approaches for integrating performance prediction in the early phases of the software life cycle. These approaches are classified according to relevant dimensions: software specification, performance model, evaluation method, and level of automated support for performance prediction. This survey gives some indications of the software system specification and performance modeling. For software specification, most of the approaches analyzed use standard practice software artifacts such as UML diagrams, whereas the Queuing Network Model and its extensions are candidates for performance models since they represent an abstract and black box notation allowing easier model comprehension, especially in a component based software development process. However, Smith et al. [79] proposed a performance model interchange format (PMIF) as a common representation of system performance modeling data. The idea is to establish a standard format to interchange models among different performance prediction tools that use a Queuing Network Model paradigm, in order to be able to use the same model for different analyses. Finally, [77] presents a performance engineering methodology that addresses the early stages of the development process. It is based on UML sequence diagrams annotated with performance information using the Profile for Schedulability, Performance and Time. These UML diagrams are then automatically translated into the stochastic process algebra FSP and are analyzed using existing tools

to study both the behavioral and performance properties of the SUT, detecting bottlenecks.

These approaches to model based software performance prediction are distant from our proposal since they are conceived for internal analysis of the performance of the system and not for testing. Indeed, the main purpose of these works is not the test case modeling as in our proposal, but the representation of the internal structure of the system and the simulation of its behavior with model analysis, to predict performance results.

Search-based testing for non-functional properties: Search-based software testing deals with the application of metaheuristic search techniques to generate software tests. In recent years these techniques have been also applied to testing non-functional properties. McMinn [85] provides a comprehensive survey of the application of metaheuristics in white-box, black-box and gray-box testing. This survey also addresses non-functional testing, evidencing the application of metaheuristic search techniques for checking the best-case and worst case execution times of real-time systems, detecting input situations that break memory or storage requirements or cause an automatic detection of memory leaks. An extension of this survey [84] focuses on types of nonfunctional testing, targeted using metaheuristic search techniques. The results of this survey show that metaheuristic search techniques have been applied for testing of the execution time, quality of service, security, usability and safety. These techniques are totally different from those in our proposal since they are mainly based on genetic algorithms, grammatical evolution, genetic programming and swarm intelligence methods.

According to Table 6 and to the previous discussion evidence, even though there are many interesting proposals addressing non-functional aspects of software development and quality verification, almost none of them present a metamodel for functional and non-functional test cases representation. The only one that does seems to be UML-TP that is also the standard proposed by OMG.

2.2.5.2. CURRENT VERSION OF UML-TP FOR PERFORMANCE TESTING

This subsection shows the main limitations of the current UML-TP standard version for modeling performance and dependability test cases. Specifically, when experimenting with UML-TP for trying to model some real load testing scenarios, it was observed that:

- i) it does not provide support for modeling the workload concept;
- ii) it does not include the most common validation facilities for performance and dependability, mainly based on the average or percentage of the response times values or the number of pass and fail verdicts.

In load testing it is important to model the workload that is usually required from any load simulation tool. The workload defines how many operations can be concurrently executed on the SUT in a given interval time. As part of testing design the tester should, for instance, define how the different test cases can be executed concurrently into a load testing scenario, what data they use, the delay between different test executions and how the test is going to reach the simulated load goal, namely the ramp-up of each test case.

As already noted, there is no workload concept defined in the standard. However, using the UML-TP concepts provided in the standard and according to the UML-TP examples in [52], [53], it is possible to derive a partial and complex way to model the workload. The example in Figure 13 illustrates how a simple workload example composed of one test case (*testcase_1*) executed by 150 users during a certain time, can be modeled.

In this example the workload is considered as a test case in the Test Context. The Test Component concurrently executes the different stimulus on the SUT for a certain time. As shown in the figure, the Test Context includes a test case *testcase_1*, and another special test case *testcase_workload* which is in charge of the concurrent execution. The behavior of this special test case (*testcase_workload*) is modeled with a sequence diagram where the Test Component executes the *testcase_1* inside a loop. To set the total test execution time a timer (*timer1*) is added. The number of users participating in the simulation for the test case is specified in a generic parameter of the Sequence Diagram (*maxUsers*), but it is not clear how to set this parameter. In addition, for the distribution of the executions (think times between executions), it is necessary to store the values in a Datapool, and give them to the timer (*timer2*) which establishes a pause in the loop after the execution of the test case. The main problem of this workload representation is that it is very complex and incomplete since it does not allow specification of different and concurrent test cases (it is not clear how to represent more than one test case in that way) or other important concepts, such as the ramp-up of the test case.

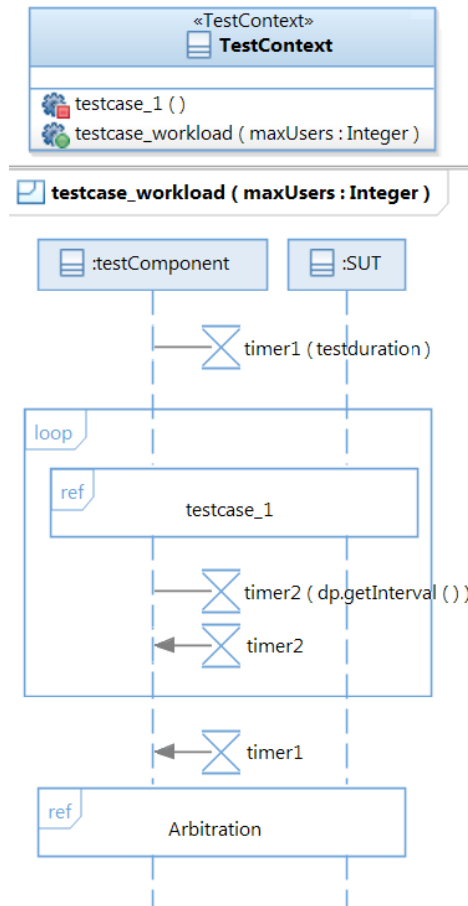


FIGURE 13 - TEST WORKLOAD WITH UML-TP

Other examples presented in the standard specification of UML-TP [53] to represent the workload include two concepts that are the “background load” (useful for generating a certain stress on the system) and the “foreground load” (which includes the test cases that the user is interested in measuring). These examples show that the test cases are executed in parallel, but there is no way to see clearly how the workload is defined. Specifically, the limitations of the examples provided are: i) it is not evidenced which load should be executed against the SUT in order to verify the non-functional properties; ii) the number of users executing the “background load” is not represented; iii) the number of executions, the delay between executions (presented as a datapool) and the ramp-up for each test case of the workload are not clearly defined.

Another attempt to represent a workload with UML-TP is presented in [61]. In this work the authors use the UML-TP in a nonconventional way in order to model a load test by specifying in the datapool the percentage of users executing each test case. This workload representation is also limited since the semantic of the workload

representation is not in the model, but in the way they interpret the content of a generic datapool.

Another important aspect that, in our opinion, is not well-covered by the UML-TP standard, is related to validation, namely how to define the verdict when a non-functional property (performance or dependability) related to a set of test cases needs to be verified. In these situations, the arbiter should be capable of expressing global validations in a simple way, taking into account, for instance, the average of the different response times of all the executions of a test case, or considering the verdicts of all test case executions, in order to compute the percentage of the passed test cases.

The aforementioned book [52] and the UML-TP standard [53] present some examples in which the global arbiter gives the verdict according to the percentage of “pass” test cases. However, it is necessary to explain how to calculate this percentage in order to set the verdict; Figure 14 shows a possible representation of a user defined behavior for an arbiter (according to the examples of the standard) asking for some percentage of “pass” responses. Since this kind of validation is much more common in performance and dependability testing, it is important to have a simple way to represent it.

On the other hand, and still on the lines of non-functional validations, the current UML-TP standard allows us to determine the minimum and maximum accepted response time values, specifying that all the response times should be within a certain range of values. These time restrictions are not enough to represent the most typical validations that are usually performed in a load simulation test, namely the average or percentage of the response times under a certain boundary.

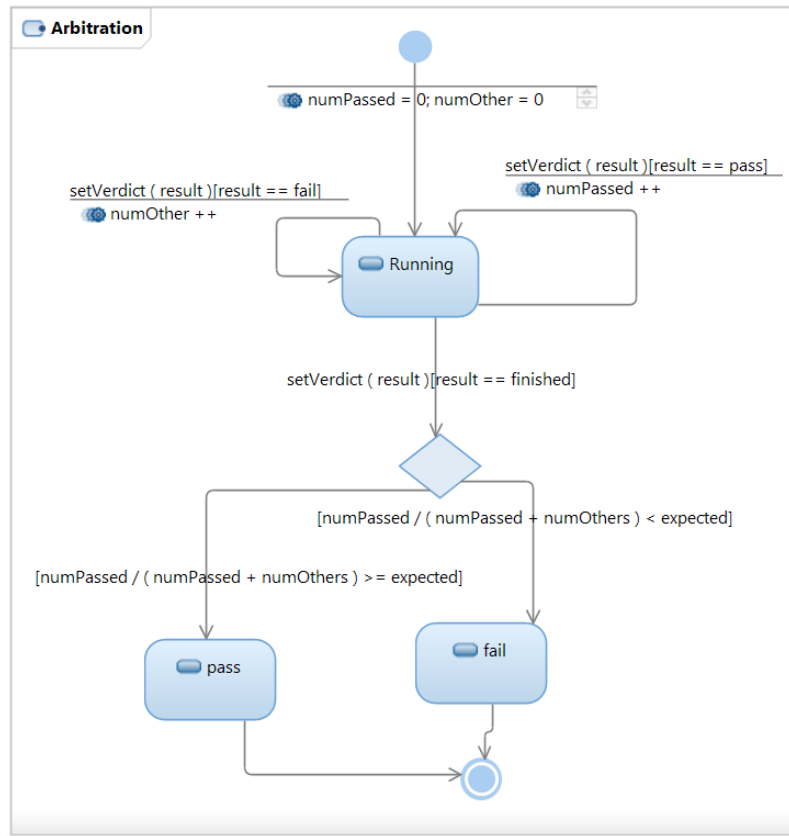


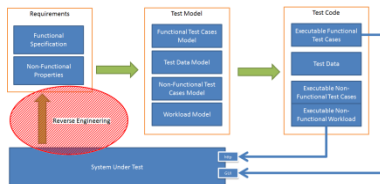
FIGURE 14 - USER DEFINED BEHAVIOR FOR AN ARBITER

For both cases (performance and dependability validations) a limitation is given by the default behavior of the arbiter that has an only-gets-worse policy. This means that if one test case reports a failure, the whole test suite fails. It is desirable to take into account all the test executions and provide a global verdict for the test suite according to a different policy.

2.3. PROCESSES, METHODOLOGIES AND APPROACHES IN THE CURRENT PRACTICE

This section focuses on how the different things are done in current practice (by companies, in the industry area) related to the activities involved. It therefore includes an introduction to reverse engineering, functional and performance testing, and to conclude, some ways to determine the quality of the test cases.

2.3.1. REVERSE ENGINEERING AND REVERSE ENGINEERING OF DATABASES



Following reverse engineering techniques, from the database schema, different aspects of the SUT could be inferred or supposed, which are essential for the test case design, such as the data model, some of the business rules and even the graphical user interface.

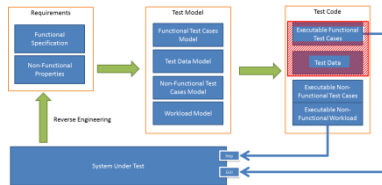
Generally, reverse engineering has been used for software modernization purposes, as Davis and Aiken [88] or Canfora and di Penta [89] showed. This consists of three main stages [90]: (i) reverse engineering, which analyzes elements of existing systems and their relationships and obtains an abstract representation of the system; (ii) the restructuring stage, which changes some internal features of the abstract representation of the system (maintainability, understandability, performance, etc.) whilst the external behavior is preserved; and finally (iii) the forward engineering stage, which obtains the targeted, modernized system by decreasing the degree of abstraction.

To our knowledge, there are few proposals purporting to take advantage of reverse engineering to aid the test design tasks.

Memon et al. [91] proposed the application of reverse engineering of a graphic user interface (for Java and Windows applications) in order to obtain an event-flow model that could then be used to generate test cases. Bellentini et al. [92] presented WebUML to generate test cases using UML Class Diagrams (to describe the components) and UML State Charts (to describe navigation and behavior), which were extracted with reverse engineering techniques from the SUT (specifically for web systems). Ricca et al. [93] also use UML models which are the result of the reverse engineering process of web systems. They apply white box testing techniques to verify the different flows of the SUT. Kung et al. [94] take into account the structural and navigational aspects of the system to extend traditional data flow testing techniques. Di Lucca [95] developed a complete framework to apply reverse engineering to web systems, representing it with a model that they use to generate and execute test cases.

None of these consider the database for the design of test cases. As pointed out in Chapter 1, the database schema is considered a good candidate element for a test-generation process, since the structure and behavior of the remaining application layers have a very strong influence on it.

2.3.2. IMPLEMENTATION OF FUNCTIONAL TESTS



The test discipline considered in our research is based on the Unified Process (UP) [96]. Despite being presented in the UP context, it represents the core testing activities described by a large number of works; in particular our terminology is adherent to the

one defined in UML-TP [97]. Typically, the first activity is the test planning. Its purpose is to plan the testing efforts by describing a testing strategy, estimating the requirements and scheduling the testing effort. The test cases are then identified and described, including test data, test procedure and test oracles. Afterwards, the automation of the test procedures takes place, which is an optional and well adopted strategy due to the cost savings in the execution activity. Once the test cases are automated in a test execution tool, they are executed according to the test plan and the defects are reported to the appropriate person.

In functional testing the specification is used to obtain the test requirements and test data, without any knowledge of the implementation [98]. It is necessary to use a high quality specification associated with the customer's requirements in order to correctly apply the functional test criteria.

There are mainly four different levels for automation: scripted testing, record and playback, data-driven and model based testing [99]. In this thesis all are considered, and they are explained below, as are some tools that can be used to follow them.

Over the last few years, the agile development community has implemented various frameworks to automate the software testing process, commonly known as **xUnit**; these are based on the principle of comparing the obtained with the expected output, and have quickly reached popularity: in fact, many development environments have 'plugins' and 'wizards' to facilitate xUnit testing (e.g. Eclipse, Microsoft Visual Studio, etc.) [100]. The basic idea of xUnit is to have a separate test class, containing test methods that exercise the services offered by the class under test. The most popular are perhaps JUnit [5] for Java and NUnit [101] for Microsoft .NET. These tools follow the **scripted testing** approach.

Going one step further in testing automation, are the **record and playback** tools. The most well-known open source tool for web environments following this strategy is Selenium [102]. This kind of tool also presents a scripted testing approach, but they have a special facility to create the test cases, by "recording" them. Through this mechanism the user simply manually executes a test case while the tool captures all the user actions

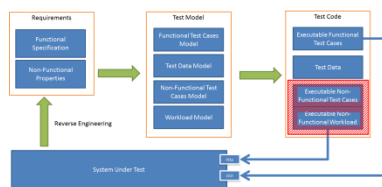
on the Graphic User Interface. These actions are stored into a script so as to reproduce them later.

The scripting approach can be improved following **data-driven testing** (DDT) [99], [103] in which the test procedure is parameterized so it can be executed with different test data.

All these approaches always execute the same test path, the one that was written or recorded into the test script. **Model based testing** (MBT) is an approach in which test cases are designed on the basis of a model of the test object [99], generally a state machine. The model can be read by a tool that handles the creation and execution of test cases, exploring different paths of the model according to different coverage criteria. This technique is therefore useful to explore new test paths in addition to the one initially specified by the tester. There are two main approaches: online MBT and offline MBT. The former implies the execution of the test case while the model is being examined. The second implies the examination of the model in order to generate the test cases that are executed *a posteriori*. There are different tools supporting both types of MBT. For example, and only considering open source tools, CTweb¹¹ supports offline MBT, while Graph Walker¹² and ModelJUnit¹³, support both.

Another kind of tool and strategy used in software testing for test data design is called **combinatorial testing**. As it is impossible to test all data combinations, the approach is to look for intelligent strategies for combining data from the different variables of the test case, minimizing the number of combinations and maximizing the possibility of finding errors. For this, there are combinatorial approaches such as the one presented in this survey [104]. Alarcos Research Group has developed its own combinatorial testing tool called CTweb [105], which is being adopted progressively by the industry.

2.3.3. IMPLEMENTATION OF PERFORMANCE TESTS



It is often necessary (and recommended) to simulate the expected workload in order to verify performance and dependability properties in a system. This kind of simulation is known as a Load or Performance Test.

Different load testing methodologies adopted by the

¹¹ CTweb: <http://ctweb.abstracta.com.uy/>

¹² Graph Walker: <http://graphwalker.org/>

¹³ ModelJUnit: <http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/>

industry, such as [59], [106], were analyzed, including activities for the definition and design of the test, for the automation of the test cases using an appropriated load simulation tool, and for the execution of the load scenarios, simulating the load, verifying the responses of the SUT and their response times, and the behavior and performance of the infrastructure.

The main issues to analyze in the test definition and design phase are: the test scenarios (workload), the test cases that run in the scenarios, the environment in which the tests will run, the data needed in the tests, and the acceptance criteria.

A scenario specifies the expected use of the SUT, showing how the users will interact with the SUT, or in other words, the workload to be simulated in the tests. Determining the scenarios is one of the most complex and critical parts of this phase of a performance testing project. If the workload is not properly defined, the tests are going to be verifying something wrong, obtaining only useless results. Generally one scenario represents one specific time or interval on a day; for example, it is possible to define a “diurnal scenario” as the total user interaction with the SUT from 13.00 to 14.00 hours.

It is then necessary to determine which test cases are included in the scenario, which data they use and how many users and with which frequency they use this operation. As a part of the test design, the tester should then define the way the test is going to reach the simulated load goal, that is, the ramp-up of each test case.

Finally, it is crucial to define the acceptance criteria based on the non-functional requirements. In that way the tester defines the expected response time (performance) and fault tolerance (dependability) for each test case. Typically, these kind of validations are defined on the average of all the response times, or saying that a certain percentage of executions would answer correctly, because it is expected that some values fall outside the acceptable value boundaries, so the restriction cannot be imposed on all the test cases. This means that if one test case responds in more than the expected time it does not imply that the test suite failed, because this could be an “outlier”.

Once the test is designed, the testers should use appropriate tools to automate the execution of the test cases and combine them in a proper test scenario, simulating the workload and verifying the defined acceptance criteria. There are specific tools to do this, called *load generators* or *load testing tools*, simulating concurrent users accessing the system. Two of the most popular *open source* load generators are OpenSTA (opensta.org) and JMeter (jmeter.apache.org).

Unlike the functional test automation, even though the *record and playback* approach is also common in the workload scripts, these tools do not record at a graphic user interface level. Instead, they do it at the communication protocol level. This happens

because a functional test script reproduces the user actions on a real browser, whilst load generators try to “save” resources by doing the simulation at a protocol level: for the HTTP protocol, for example, the tool will launch multiple processes that simply send and receive the corresponding byte arrays through a network connection. Since the goal of these tests is to check the behavior of the server, neither the user interface nor any other kind of graphic elements are required.

The workload script contains a sequence of commands that manage HTTP requests and responses according to the protocol. This script is much more complex than the equivalent functional test script.

2.3.4. MEASUREMENT OF TEST QUALITY

According to Offut et al., a *testing criterion* is a rule or collection of rules that impose requirements on a set of test cases. Test engineers measure the extent to which a criterion is satisfied in terms of *coverage*, which is the percentage of requirements that are satisfied [107]. The adequacy criterion can then be used to automatically generate test cases to satisfy the selected criterion.

The coverage criteria are used: (1) to know which areas of the system the test cases have exercised; (2) to find unexplored building blocks; (3) to create new test cases to exercise those unexplored building blocks; (4) in some situations, achieving a predefined coverage without finding new errors that could be used as a stop testing criteria [108].

Some coverage criteria that are useful for IS with databases are presented, and they will also be useful to explain and justify our proposal.

Coverage Criteria for Test Data

Equivalence partitioning and boundary values: The input or the output data are divided into disjoint sets, and it is assumed that the system under test will have the same behavior as any element of the set, so, any value of the set is a representative of the entire set. The values in the borders of the equivalence classes are also used [4].

It is a black-box approach, so, when testing an IS the definition of the equivalence classes is based only on the knowledge of the business and the requirements.

Combinatorial testing: Combinatorial testing is concerned with the construction of test cases using algorithms that combine the “interesting values” (i.e., the test data) identified for operation parameters. Grindal et al. [104] present a compilation of several combination strategies, whose goal is to obtain complete test cases that fulfill some coverage criterion and, thus, to quantitatively know the degree of use of the test data. Some of most well-known algorithms they cite are *All combinations* (which builds all the

possible test cases from all the test data), *AETG* [109] and *IPO* [110] (which obtain pairwise coverage: that is they visit all value pairs of any two parameters) and other deterministic and non-deterministic (genetic algorithms, for example) strategies.

This technique is useful for testing an information system, once test data has been selected for each input, in order to combine them into test cases.

Coverage Criteria for Class Diagrams: Andrews et al. [111] propose different coverage criteria for testing UML diagrams. For class diagrams, they propose the following:

- *Association-end multiplicity*: the test set must include the creation of each representative pair of multiplicities in the associations that appear in the model. Thus, if there is an association whose multiplicity is, in one of the extremes, $p..n$, the association should be instantiated with p elements (minimum value), n elements (maximum value) and with one or more values from the range $(p+1, n-1)$.
- *Generalization*: the test set must cover every generalization relation of the model.
- *Class attribute*: the test set must instantiate representative data sets for the different attributes of each class.

Coverage Criteria for State Machines and Activity Diagrams: UML State Machines and Activity Diagrams can be used to express the behavior of part of a system. To validate this behavior the test set should reach certain coverage criteria. Andrews et al. [111] have proposed several coverage criteria for activity diagrams, such as:

- *Condition Coverage*: the test set must cause each condition in each decision to evaluate both *True* and *False*.
- *All Messages Paths*: the test set must cause each possible message path to be taken at least once.

For state machines, Offut et al. [107] have defined the following criteria:

- *All States*: the test set must visit each state in the diagram.
- *All Transitions*: the test set must cause every transition to be taken.
- *Transition-pair*: for each state, the test set must cover each pair of adjacent transitions (considering the ingoing and outgoing transitions in every state).
- *Complete sequence*: the test set must include those paths in the state machine that the tester considers interesting, and that are not covered by the previous criteria.

Coverage Criteria for CRUD: The data instances of a system have a life cycle starting when they are created and finishing when they are deleted; they also go through a set of updates in between. Koomen et al. [103] have defined a coverage criterion for this life cycle: test cases could be defined with a regular expression starting with a *C*, followed by an *R*, followed by every operation that performs a *U* (with an *R* after that to validate the result) and finally a *D* and another *R* (to validate the deletion). Then, representing with U_i each operation that updates data (over different attributes for example), the criterion could be represented with the following regular expression: $C \cdot R \cdot (U_i \cdot R_i)^* \cdot D \cdot R$.

Testing database applications

Regarding test case generation for systems with databases, Tuya et al. [112] define a coverage criteria based on SQL queries, applying a criterion based on *Modified Condition/Decision Coverage* but adapted to the conditions in FROM, WHERE and JOIN clauses. In other approaches [113][114] the coverage criteria are extended to include the embedded SQL code, generating database instances that, from the testing point of view, cover the scenarios which are considered interesting. Arasu et al. [115] propose specifying in some way the expected results of each SQL included in the test, and then they generate test data to satisfy this specification. The proposal from Chays et al. [116], called AGENDA, takes as input the database schema and categorizes test data given by the user, thereby generating test cases and initial database states, and validating the outputs and the final database state after the test case execution. By means of a constraint solver, Neufeld et al. [117] generate database states according to the integrity constraints of the relational schema.

To the best of our knowledge, there are many proposals for test case generation, but none focuses on automated test model generation using model transformations based on the database structure, applicable for external teams (with a black box approach), using standards, and considering functional and non-functional properties at the same time.

2.4. CONCLUSION OF THE STATE OF THE ART AND PRACTICE

This chapter has briefly introduced the basic concepts of software testing, including functional and non-functional testing. It has presented a more detailed explanation of Model Driven Engineering. The notions of model transformation were provided, describing in more detail the standard metamodels and model transformations used in this thesis.

Finally, the main research topics related to this thesis have been explained more comprehensively: model-driven testing and performance testing. The current state of

the art in these topics has been summarized and the open issues in research have been identified. These open issues give rise to the principle motivation of this work, which is the provision of a model-driven testing approach based on OMG standards to define test models and model transformations considering both functional and non-functional aspects of an Information System using databases. The approach is described as a methodology to obtain automated test cases from models. This methodology is presented in Chapter 3.

Quand tu veux construire un bateau, ne commence pas par rassembler du bois, couper des planches et distribuer du travail, mais reveille au sein des hommes le desir de la mer grande et large.

– Antoine de Saint-Exupery

If you want to build a ship, don't drum up people to collect wood and don't assign them tasks and work, but rather teach them to long for the endless immensity of the sea.

– Antoine de Saint-Exupery

CHAPTER 3. MANDINGA: METHODOLOGY FOR AUTOMATION TESTING INTEGRATING FUNCTIONAL AND NON- FUNCTIONAL ASPECTS

This chapter explains the complete methodology for test case generation. It includes functional and non-functional aspects in an integrated way, and summarizes which instruments are used in each part of the methodology.

3.1. INTRODUCTION

In the traditional view, non-functional testing is performed after completing functional testing, with the aim of covering non-functional aspects such as performance, dependability and security. Usually different models are used for the two aims in the model-driven approaches. Our approach moves away from this traditional practice and breaks down the boundaries between functional and non-functional testing, incorporating functional and non-functional aspects into a comprehensive testing model which will later be translated into the test code and will be useful to perform functional and non-functional validations.

From our experience in dozens of projects, providing both functional and performance test services, the same situation has been observed many times: in the first stage the testers prepare the functional test specification, automate those test cases, and execute them; later, they prepare the non-functional test specification, automate those test cases, and execute them. Two different specifications, two different groups of test scripts, and most of the time, the non-functional test cases are a subset of the functional test cases, perhaps with some specific elements to measure performance, such as

timers. Moreover, functional issues are often found during the execution of non-functional tests, or vice versa. Last but not least, many customers claim not to have the time (or budget) to execute both test sets, therefore they opt to leave the performance test to another stage, assuming all the associated risks. The question is: why is an integrated approach not taken into consideration, designing, modeling and executing the functional and non-functional tests together?

This section presents MANDINGA, depicted in Figure 15, a methodology to generate automated test cases for functional and non-functional verification with one single and integrated model.

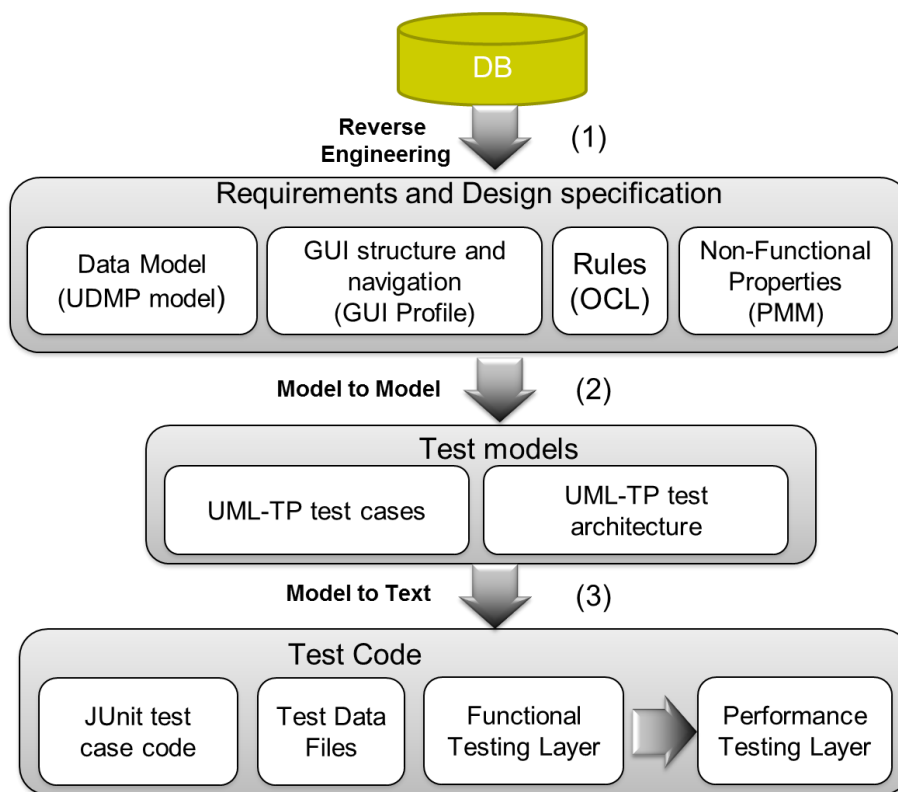


FIGURE 15 - MANDINGA METHODOLOGY

3.2. METHODOLOGY

The main characteristic of the methodology is that testing artifacts are obtained through model transformation from models specifying the SUT, specifically the database structure. For this reason, the metamodels and models used become particularly important, as do the tools involved.

The methodology was defined to achieve the following goals:

- **UML notation:** the proposal uses standards as much as possible. Thus, the UML notation is the metamodel of most of the models involved to represent functional specification of the SUT and testing models. Testing models are also represented using the UML-TP that is the standard notation for testing.
- **Model-driven test case generation:** Test cases are automatically generated from the specification models and evolve with the product up to the test code generation. System behavior is mainly represented using UML sequence diagrams that are first transformed into test models and later into test code.
- **Functional testing level:** the methodology generates test cases at the functional testing level, i.e., the system is considered as a black box and the stimuli between the system and the exterior are tested.
- **Non-Functional testing:** functional test cases are also used to generate a workload simulation, with the aim of verifying non-functional properties such as performance and availability.
- **Extensible test patterns:** test cases are generated based on test patterns; for certain substructures in the system's models, some special test scenarios are taken into consideration.
- **Standardized artifact handling:** The framework is based mainly on OMG standards. The standards used are UML, UML-TP as metamodels, and QVT and MOFM2T as standardized transformation languages.
- **UML-TP extension:** The UML-TP was extended to improve the description of non-functional tests.
- **Tools for model edition and visualization:** Existing UML modeling tools have improved editors to generate all the UML models. The selected modeling tool must allow description of UML models and application of UML profiles; it also should be able to export and import XMI files following the UML metamodel that represents these models.
- **Tools for model transformation:** Two kinds of tools are required to execute the defined transformations: one for model-to-model and another for model-to-text transformations.
- **Tools for automating testing tasks:** some well-known tools for test execution are used to exemplify and demonstrate the possibility of test execution.

The methodology has several steps and involves several actors. Most of the steps fit into different standards mainly from the OMG, especially UML, in order to use general UML modeling tools. The most important steps/phases are:

Information System Model Specification: taking a physical database as input, a reverse engineering process builds a class diagram that is the *Information System Model (ISM)*

artifact shown in the figure, and represents the possible conceptual model used when the database was built. This diagram is UML-compliant and describes different views of the system, such as the database and the user interface. The tester may review, adapt and complete the model with any UML-standard tool. In parallel, non-functional properties are specified in a PMM model. See Chapter 4.

Functional Test Cases Generation: Afterwards, an automated process searches interesting test situations within the ISM: given a set of entities, an interesting test situation corresponds with the occurrence of a generic pattern defined as an ATL rule. These rules are independently defined by the tester and are launched against the ISM similar to a regular expression which is launched against a plain text. Each time a substructure of the ISM maps onto a pattern, a set of test cases will be added to the test model. These test cases are designed taking into account one or more coverage criteria. Furthermore, the tester may enrich the test model with new test cases (described as UML sequence diagrams).

Later, functional test cases in the test model are translated into test code. The generated test cases interact with the SUT through a set of wrappers comprising an adaptation layer. These wrappers are generated with a semi-automatic mechanism, or can be provided by the user, in order to have test cases that are completely executable. See Chapter 5.

Performance Test Cases Generation: After this, functional test cases are used to generate performance test cases (for workload simulation). The definition of the workload is taken from PMM models, considering the coverage criterion defined on the PMM operands. See Chapter 6.

The final result is a set of executable test cases to verify the functional and non-functional properties of the SUT. See Chapter 7, in order to see how the whole framework was implemented in a prototype, with the execution of some examples, and then used into the industry, using the generated knowledge in real projects.

It is important to emphasize that this process can be performed in parts or modules (applying *divide & conquer*), applying different risk-based or prioritization criteria. For example, if the database has fifteen tables, perhaps only five are considered more risky, therefore the tester can start work by focusing on this subset of tables, generating test cases for them, and then perform another iteration considering the remaining entities of the SUT.

The following chapters explain each part of the methodology.

"Without specification, there are no bugs — only surprises"

Brian Kernighan

CHAPTER 4. INFORMATION SYSTEM MODEL CONSTRUCTION

This chapter explains the metamodel defined to represent the functional specification of the Information Systems under test, and how it is instantiated. This chapter also introduces DBesTest, our framework for the automation of the proposed methodology. An application to an example is explained to conclude the chapter.

4.1. INTRODUCTION

As shown in Chapter 2, there are different options for modeling functional and non-functional aspects of an Information System. This chapter presents the metamodel used in our proposal for the model-driven testing approach used to generate a completely executable testing framework.

One well-known problem with the Model-Driven Testing approach is the need for two specifications: one that developers prepare with the system code and another that testers must design to validate the first. This methodology proposes to help with the construction of the system model, and with the generation of test cases and test code – specifically for web applications using databases – by automatically initializing the system model from the data schema. The tester can then add extra information to the model to specify the expected behavior to test; this will allow the generation of Selenium test scripts at a lower cost.

The methodology is supported by a tool which was implemented as an Eclipse Plugin, called DBesTest. It includes the required metamodels and it is composed of a set of ATL and Aceleo scripts in order to execute the model-to-model and mode-to-text transformations respectively. It also integrates a reverse engineering tool, and adapts the resulting models to be compatible with the selected UML editor (this should not be necessary, but the different tool providers do not respect the UML standard, thus, demanding extra effort to interoperate between tools).

4.2. INFORMATION SYSTEM METAMODEL

The Information System Model (ISM) is composed of different views of the system, all proceeding from the database schema. As already mentioned, the ISM is composed of the Data Model, the Graphic User Interface Model (structure and navigation) and the Business Rules (see Figure 16).

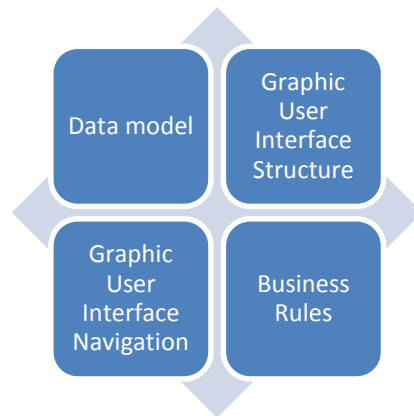


FIGURE 16 - INFORMATION SYSTEM MODEL COMPONENTS

There are different database reverse-engineering approaches (as mentioned in Chapter 2). Our proposal uses the approach presented by Polo et al. [30], who developed the *Relational Web* tool.

The following subsections focus on each part of the ISM.

4.2.1. DATA MODEL

The Data Model is represented using UML Data Modeling Profile (**UDMP**) [32], which is a UML model extension developed by IBM to design databases using UML, with the expressive power of an entity-relationship model. It defines concepts at a physical level and architecture (*Node*, *Tablespace*, *Database*, etc.), and those concepts required for the database design (*Table*, *Column*, etc.). Several proposals use this profile to model database structure [118][119][120].

Table 7 shows the different elements of the UDMP, making a correspondence with the database elements, and indicating which type of UML element is extended to represent them.

TABLE 7 – ELEMENTS IN IBM’S UML DATA MODELING PROFILE

Database element	Description	UML Data Modeling Profile	Extended UML Element
Database	The system for data storage and controlled access to stored data.	«Database» stereotype.	UML Component
Schema	The schema is the biggest unit that can be worked with at any given time.	«Schema» stereotype.	UML Package
Table	A set of records of the same structure, also called rows.	«Table» stereotype.	UML Class
View	A view is a virtual table, referencing columns of other tables.	«View» stereotype.	UML Class
		«Derived» stereotype.	UML Relationship
Column	Set of data values of a particular simple type.	No stereotype needed. Property of a “Table” or “View” stereotyped class.	UML Class Property
Primary key	Primary keys uniquely identify a row in a table.	«PK» stereotype.	UML Class Property UML Operation
Foreign Key	Foreign keys access data in other related tables.	«FK» stereotype.	UML Class Property UML Operation
		«Identifying» stereotype. «Non-Identifying» stereotype.	UML Relationship
Nullable	Indicates whether the column is mandated to contain data or not.	«Nullable» stereotype.	UML Operation
Index	Physical data structure that enables faster data access.	«Index» stereotype.	UML Operation
Unique	Constraint that defines a column or set of columns as containing unique data.	«Unique» stereotype.	UML Operation
Stored Procedure	It is a subroutine, stored in the database, available to applications that access it.	«SP» stereotype.	UML Operation

The table and view columns are represented in UDMP as attributes of the corresponding class.

All the attributes that belong to a primary key, foreign key, index or unique restriction have a corresponding stereotype, but it is also necessary to add an operation (with the same stereotype) to show that a primary key, for example, is composed of two attributes. This operation has all the attributes involved in the restriction as parameters.

Figure 17 shows the representation of the data model for AjaxSample. It is possible to see different foreign keys and primary keys defined in each table, with stereotypes applied for that in the columns (attributes of the classes) and in special operations to represent them. The operation *PK_City* indicates that there is a primary key in the *City* table comprised of two attributes (the parameters of these operations), and all the attributes corresponding to the parameters and the operation are stereotyped with the “PK” stereotype. It is easy to see that this model corresponds with the database schema presented in Chapter 1, where some small changes were made by the user in order to

simplify the model, removing columns that were not useful for the test case generation (for example *InvoiceLatestLine* in the *Invoice* table).

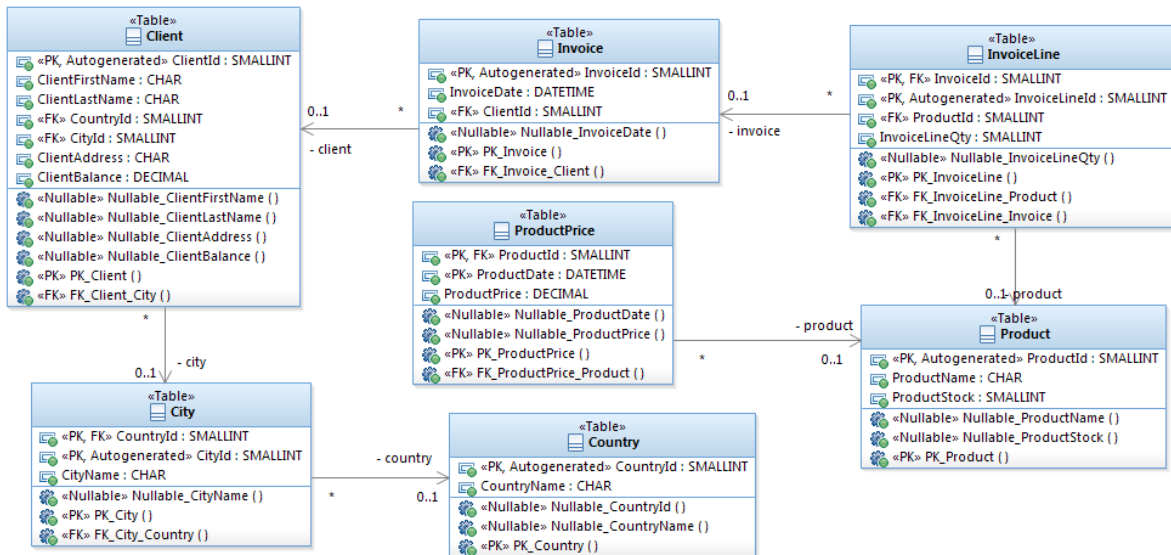


FIGURE 17 - ENTITY REPRESENTED WITH UDMIP

It was necessary to develop our own implementation of the metamodel in Eclipse as a UML Profile, in order to use it with the UML SDK [121] and with model transformation languages. The profile was extended with some aspects that were considered necessary for our purpose. For example, it was necessary to define primitive types according with database types (BIGINT, BINARY, BIT, CHAR, DATETIME, DECIMAL, FLOAT, IMAGE, INT, MONEY, NCHAR, NTEXT, NUMERIC, NVARCHAR, REAL, SMALLDATETIME, SMALLINT, SMALLMONEY, TEXT, TIMESTAMP, TINYINT, UNIQUEIDENTIFIER, VARBINARY and VARCHAR). Another stereotype was also defined to represent those attributes that in the database are defined as auto-generated, because those attributes are not inserted from the user interface.

4.2.2. GRAPHIC USER INTERFACE MODEL

As the Graphic User Interface metamodels analyzed were considered more complex than required (as explained in Chapter 2), it was decided to develop a new and simple metamodel: Graphic User Interface Profile (GUIMP) as a UML Profile. This profile thus includes the stereotypes necessary to our purpose of executable test case generation (interacting with the graphic user interface).

Information systems usually provide the user with the necessary interface to perform the basic operations to manipulate data, which includes CRUD operations (create, read, update, delete), and typically also a listing of the different instances of the entities. On

the other hand, and considering that the focus is to obtain information to generate executable test cases, it is necessary to have information about how to access to the different functionalities of the system, representing the execution flows and the elements with which to interact. There are therefore two main views, **structural** and **navigational**, for the Graphic User Interface model.

The structural part of the Graphic User Interface model stores information about the different elements that the user interacts with (pages, buttons, combo boxes, etc.) and will be useful when generating executable test cases commands on those elements, simulating the user actions.

For example, Selenium can simulate user actions with specific commands. The different commands interact with the different elements of web pages as inputs, checkboxes, combo boxes or buttons. For this, Table 8 shows the main stereotypes considered for the GUIMP and their corresponding Selenium commands to interact with them, simply in order to show that this categorization is useful for the following step, to generate test scripts. The metamodel was developed in Eclipse, and also as a UML Profile.

TABLE 8 - GUIMP STEREOTYPES AND THE CORRESPONDING SELENIUM COMMANDS

Stereotype	Element	Selenium Command
<<Page>>	Web pages	Open
<<Input>>	Input in forms	Type
<<Label>>	Label	Assert Text
<<Combobox>>	Combo boxes	Select
<<Check>>	Checkbox	Check, Uncheck
<<Button>>	Buttons, Image buttons	Click

For the navigational part of the Graphic User Interface model is necessary to generate different scenarios, covering the flows of the lifecycle of each entity. They are represented with behavioral UML diagrams, as sequence diagrams or state machines. For this, it is also necessary to distinguish the actions performed, such as creation, update or delete, for which the profile also includes special stereotypes to identify them. With this information the test case generation algorithm will be able to determine the oracle: after the execution of a test case using valid input data, from a valid state, if it includes the creation of certain entity, there must be a new instance of that entity with the data corresponding to the input data. Therefore, the GUIMP also provides the following stereotypes to apply to the behavior diagrams: “create”, “update”, “read”, “delete”. There are also some additional stereotypes to indicate the navigation required to access the application and the menu for each entity.

Figure 18 shows an example of the representation of a page (a class with the “Page” stereotype). In this case it is for the creation of new instances of *City*. In the figure it is

evident that for each page element (input, combo box, button, etc.) there is a corresponding attribute in the class.

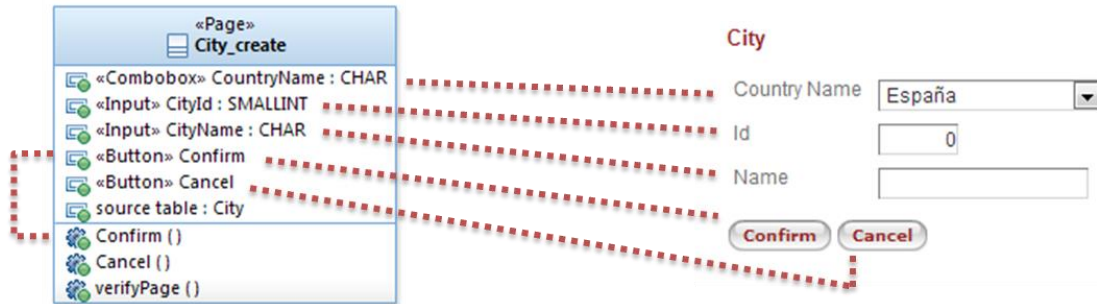


FIGURE 18 - CORRESPONDENCE BETWEEN STRUCTURE GUI MODEL AND GUI IMPLEMENTATION

For each button there is also a method (with the same name) so as to model the corresponding event raised for it. This operation has as parameters all the elements of the form that the user should insert, so that, in this case the signature of the method is:

*Confirm (CHAR CountryName, SMALLINT CityId, CHAR CityName)*¹⁴

The operation *verifyPage* is an auxiliary method allowing the user to define error identification. By default the method is designed at least to verify that the current URL is the expected one. The method is going to be useful in the test cases to show that once the test navigates to a page it should verify that it reached that expected.

There is also an attribute to reference the corresponding table called *sourceTable*.

Figure 19 presents all the elements of the structure of the GUI for the *City* entity.

¹⁴ Rational Software Architect does not show the parameters of the operations in class diagrams

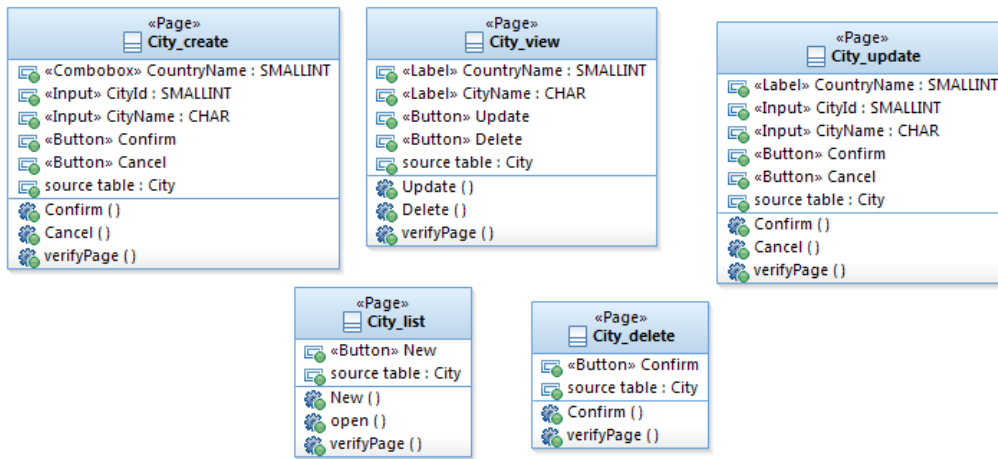


FIGURE 19 - GUI STRUCTURE MODEL FOR CITY ENTITY

The structure of the GUI also has a corresponding navigational representation (presented as UML behavior diagrams). To create a new instance of the entity *City* in AjaxSample it is necessary to follow the steps presented in the sequence diagram of Figure 20. The user first goes to the home page, then accesses the corresponding menu, that takes them to a page with a list of all the cities, where there is a button to *create* a new city. If the user clicks this button the system presents the page for the creation of cities, and after the user has completed the form and clicked the confirmation button the new city is inserted into the database.

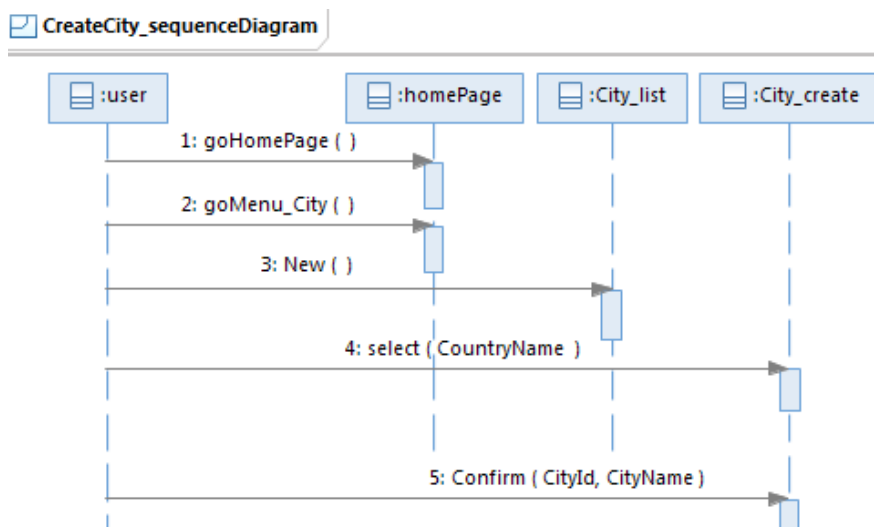


FIGURE 20 - NAVIGATION EXAMPLE FOR CREATION

Figure 21 shows the different steps of the GUI of AjaxSample so as to visualize the correspondence between the system and the model.



FIGURE 21 - CREATION OF INSTANCES OF CITY ENTITY

Each lifeline of the sequence diagram corresponds to a class in the GUI structure model. The methods invoked by the user are the ones offered for these classes, as for example the aforementioned *confirm* in the class *city_create*, for the creation of cities, with the corresponding parameters according to the inputs in the form.

In order to promote model reutilization, and to distinguish the different user intentions, the different parts of the sequence diagram are modeled separately. Thus, there will be a sequence diagram to represent access to the system (it is not the case, but in this diagram it will typically be necessary to ingress with credentials, that is, user and password), and then another for the creation of instances.

Therefore, a sequence diagram representing how to create an instance will be stereotyped with the “create” stereotype (as already mentioned, there are also stereotypes for “update”, “delete”, etc.).

4.2.3. BUSINESS RULES

If test case generation is based only on the information provided in the data model and the graphic user interface, the expected result of the test case cannot be determined. This is why the business rules are also considered in the Information System Model, and also to generate test cases and test data considering the boundary situations for those rules.

The best option for business rules is OCL [38], because of its integration with UML models.

The rules can be simple (those that apply to one attribute or column) or compound (defining restrictions on several attributes at the same time). For example, Figure 22 shows an example defining the value of the *ProductStock* column (amount of products in stock) cannot be less than 0.

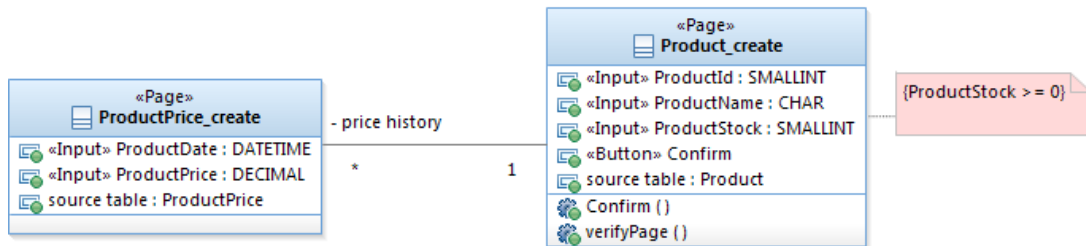


FIGURE 22 - OCL BUSINESS RULES EXAMPLE

This information is fundamental for determining valid and invalid data combinations, and also to test boundary situations. From here, it is evident that there should be a test using the value “0” or a negative value, with the expected result that the user receives a notification of the error, and the database does not change.

There are different engines and parsers to work with this kind of rules such as:

- OCL Library (an Eclipse Plugin <http://www.cs.kent.ac.uk/projects/ocl>)
- Eclipse OCL (the official implementation from the Model Development Tools Project in Eclipse <http://www.eclipse.org/modeling/mdt/?project=ocl>)

4.3. DBESTEST IMPLEMENTATION



DBesTest is the tool developed for supporting the methodology. DBesTest acts as an intermediary between the tester and external UML tools: (1) it reverse engineers the database and produces the ISM; (2) once the tester considers it is ready, it processes the model and translates it into a test model via a set of ATL transformation rules; (3) when the tester has validated the test model, it generates the test code by

means of the execution of Aceleo scripts.

This section presents the use of DBesTest in order to reverse engineer the database and generate the ISM. For this process, the tool follows the actions presented in Figure 23, which will be explained in this section.

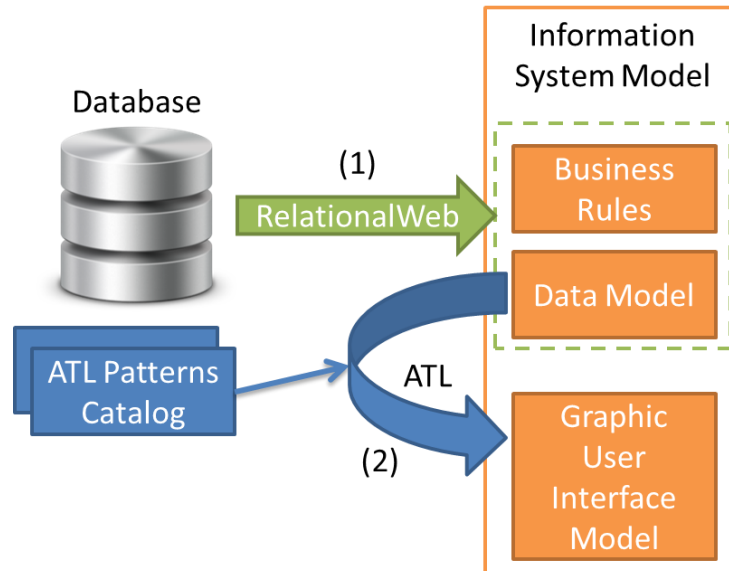


FIGURE 23 - INFORMATION SYSTEM MODEL GENERATION

In the first step DBesTest extracts the data model from the database schema, then, it generates the rest of the ISM based on the data model. Typically, there is a correspondence in Information Systems between the visual components (e.g. web forms), the data structures (generally in relational database) and the logic in between that is used to accomplish the business rules. Applying reverse engineering to a database, it is possible to derive the expected behavior of the system [30], considering certain patterns. This could be true for certain classes of applications, but in some cases there may be multiple forms that correspond to a single data structure, and a user needs to add navigation instructions to access these forms.

The section below presents both parts of the Information System construction, the database structure extraction to generate the Data Model, and the transformations applied on this model in order to complete the ISM. The result of the execution of this step is a single UML-compliant file which contains the four models of the information system. The tester may modify it with a third-party UML tool.

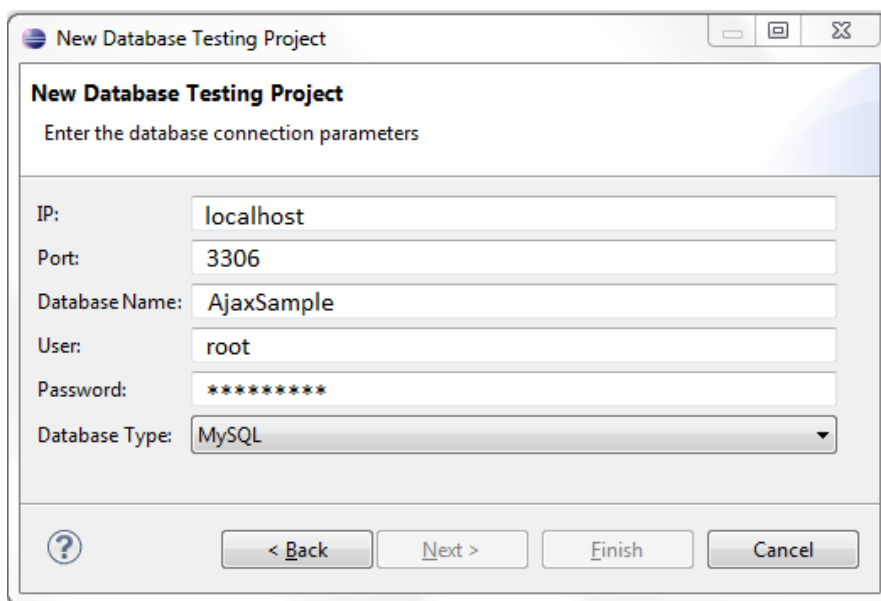
4.3.1. DATABASE STRUCTURE EXTRACTION

Relational Web[30] (a reengineering tool developed in the ALARCOS Research Group) has been adapted for loading the data model, in order to make it UML-compliant (since it uses its own metamodels).

This component is capable of coping with different database management systems (DBMS), such as MySQL, SQL Server, Oracle and Access, and it is easy to extend to others. It is simply necessary to extend two functionalities: the connection with the

database (the connection parameters or the connection string format may vary for each DBMS), and the reading of the metadata (tables, columns, data types, foreign keys, primary keys, indexes, etc.).

The user must provide connection information to DBesTest, which includes the IP of the machine where the DBMS is allocated, the port to connect it, which DBMS it uses (MySQL, SQL Server, etc.), user and password with adequate privileges, and the name of the database (see Figure 24).



The screenshot shows a dialog box titled "New Database Testing Project" with the subtitle "Enter the database connection parameters". The dialog contains several input fields and a dropdown menu:

- IP: localhost
- Port: 3306
- Database Name: AjaxSample
- User: root
- Password: *****
- Database Type: MySQL (dropdown menu)

At the bottom of the dialog, there is a help icon (question mark) and four buttons: "< Back", "Next >", "Finish", and "Cancel".

FIGURE 24 - CONNECTION CONFIGURATION

DBesTest then reads all the table names in order to allow the user select the tables to work with (see Figure 25).

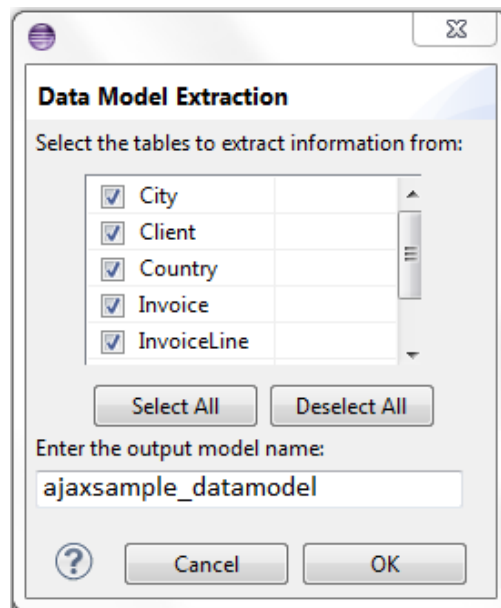


FIGURE 25 - SELECTION OF TABLE TO WORK WITH

The result of the execution of this step is a UML file (such as the one presented in Figure 26) which is compliant with the UDMP metamodel. This UML file represents the database conceptual model after applying reverse engineering to the physical schema. It includes the entities and their relationships, attributes, and constraints.

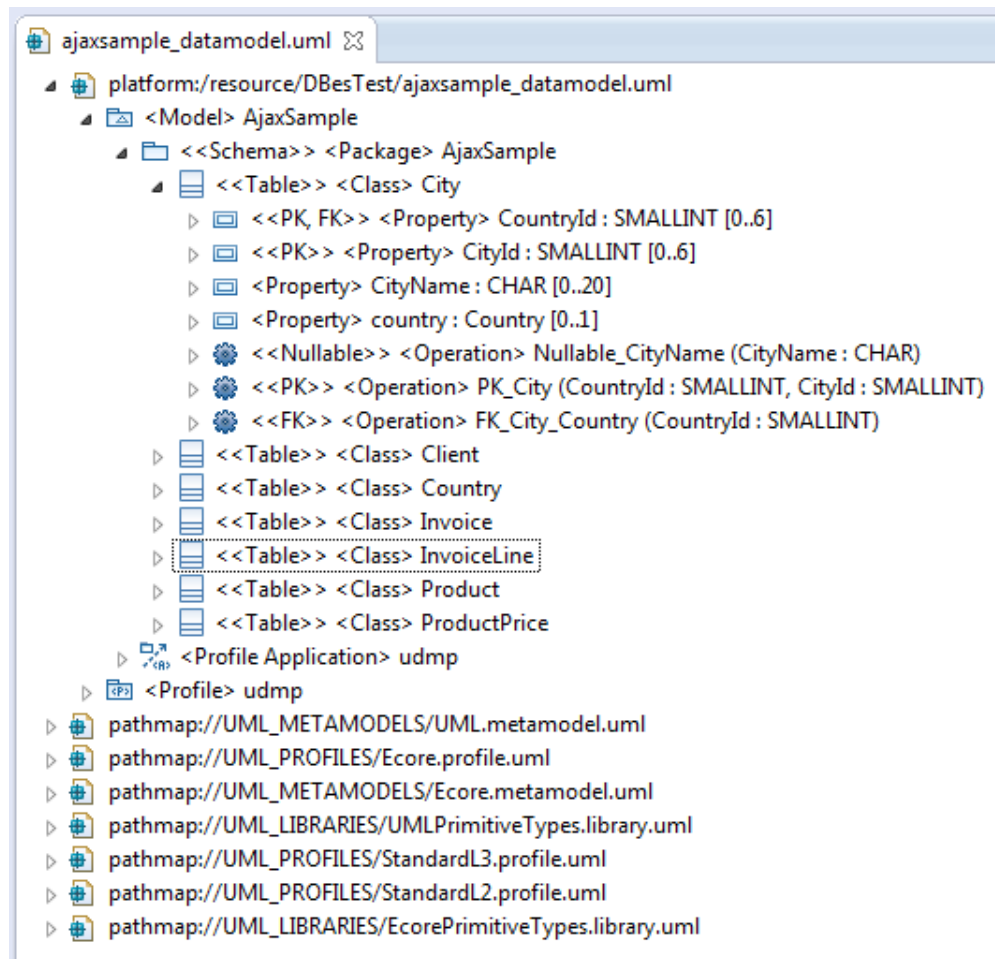


FIGURE 26 - AJAXSAMPLE DATA MODEL VISUALIZED IN ECLIPSE UML MODEL EDITOR

Basically, *Relational Web* transforms:

- One Table to a UML Class (except for certain cases as relations N to N)
- A *Foreign Key* to a Class relation with the corresponding characteristics (e.g. multiplicity).

Afterwards, the user can add more information about the business rules considered by the SUT in order to be able to generate richer and more accurate test cases. The business rules determine the equivalence classes for the test data.

These models are later verified and adjusted by the tester, mainly removing or tailoring the user interface view. The more information the user adds, the more accurate the generated test cases will be. In order to manage this model in a graphical way, it is necessary to initialize a Class Diagram including all the generated elements.

This model can be graphically represented as a class diagram, resulting in that presented in Figure 17.

4.3.2. INFORMATION SYSTEM MODEL GENERATION

The Graphic User Interface is generated by considering that for each entity there should be certain group of pages to provide functionalities such as creation, update, read, delete and list. According to the data structure it is thus possible to instantiate a possible version of the user interface structure and of its expected navigational flow. The semantic of the user interface should be the same as the semantic of the database [122]. For this task, there is a semi-automatic process where the user has to indicate which pattern should be applied for each entity. For example, if two entities are related (with a foreign key, or even with a relationship table), there are different options to manage the instances of the lifecycle at a user interface level; for example:

- One page for the referencing entity, showing its attributes and the information referenced (e.g. in the first row of Table 9, there is an entity “Country” and another entity “City” referencing the former, and in the page to view the city it is possible to see the country which it belongs to).
- One page for the referenced entity, showing the list of instances that reference it (e.g. in the second row of Table 9, the entities “Product” and “ProductPrice” are in the same situation as “Country” and “City”, but in this case there is only one page to visualize the “Product” information showing the list of “ProductPrice” instances it has related).

To argue for the validity of this assumption, generation of a simple GUI and a model from a database schema has also been a part of the Ruby on Rails framework¹⁵, or the GeneXus code generation. These techniques are mostly referred to as "scaffolding": from a database specification these tools can build an application to manage the defined entities, using the schema as a scaffold.

¹⁵ Ruby on Rails: <http://rubyonrails.org/>

TABLE 9 - EXAMPLES OF GUI AND TABLE STRUCTURE FOR A 1-N RELATIONSHIP

Database substructure	GUI for the creation of these instances												
<pre> classDiagram class City { CountryId : SMALLINT CityId : SMALLINT CityName : CHAR } class Country { CountryId : SMALLINT CountryName : CHAR } City "*" -- "0..1" Country : - country </pre>	<p>City</p> <p>Country Name <input type="text"/></p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p> <hr/> <p>Country</p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p>												
<pre> classDiagram class ProductPrice { ProductId : SMALLINT ProductDate : DATETIME ProductPrice : DECIMAL } class Product { ProductId : SMALLINT ProductName : CHAR ProductStock : SMALLINT } ProductPrice "*" -- "0..1" Product : - product </pre>	<p>Product</p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p>Stock <input type="text" value="0"/></p> <hr/> <p>Price</p> <table border="1"> <thead> <tr> <th>Date</th> <th>Price</th> </tr> </thead> <tbody> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td colspan="2" style="text-align: center;"><input type="button" value="[New row]"/></td> </tr> </tbody> </table> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p>	Date	Price	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="button" value="[New row]"/>	
Date	Price												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="button" value="[New row]"/>													

As it is probably impossible to determine this information with pure automatic reverse engineering techniques, our approach is to follow a semi-automatic mechanism, assisting the user/tester with a pattern catalog. The user will therefore have to decide which pattern applies for each set of related entities. It could be necessary to manually adjust the model afterwards. It is important to note that the goal of this process is to facilitate the task of building the entire ISM from scratch; it is not the intention to create a precise model through a new reverse engineering technique, on the contrary, it is better to use already existent techniques.

Together with the GUI structure, the navigation diagrams are also instantiated. These diagrams are correlated to the structure, as explained in the previous section, indicating the user interaction for each CRUD operation.

DBesTest performs this task with a group of model-to-model transformation scripts implemented with ATL [19]. Figure 27 shows the different inputs and the output of the ATL transformation executed by DBesTest.

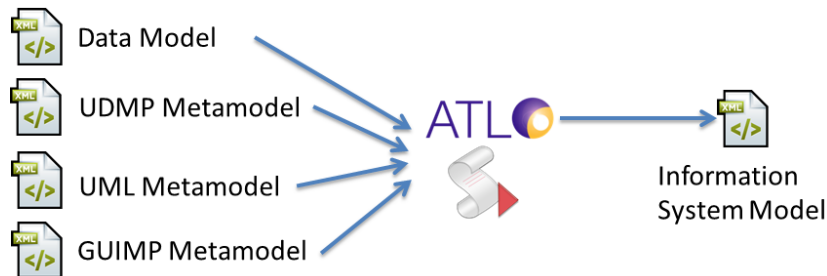


FIGURE 27 - ISM GENERATION PROCESS

Basically, the input of the transformation scripts was the data model, previously extracted from the database and adjusted by the tester, which is a UML model with the UDMP profile applied (that is why these metamodels are also part of the inputs). As the output is a UML model with the data model (with the UDMP) and the graphic user interface, it is also necessary to provide the transformation with the GUIMP metamodel as input.

The proposed algorithm is iterative and incremental. Firstly, for each table, it asks the user if there should be a group of pages to manage this entity alone. This is the case for “Country”, but there are examples where the user can say that there should not be any page to manage the instances (e.g. a log table), or that it should be considered as a part of a bigger pattern later.

Apart from the structure of the graphic user interface, the transformations also generate different sequence diagrams to show how the user navigates through those pages to create, update, delete, etc. For instance, Figure 28 shows one of the diagrams generated by ATL rules, which is a sequence diagram for the creation of the Country entity. This is the simplest pattern, one entity without foreign keys.

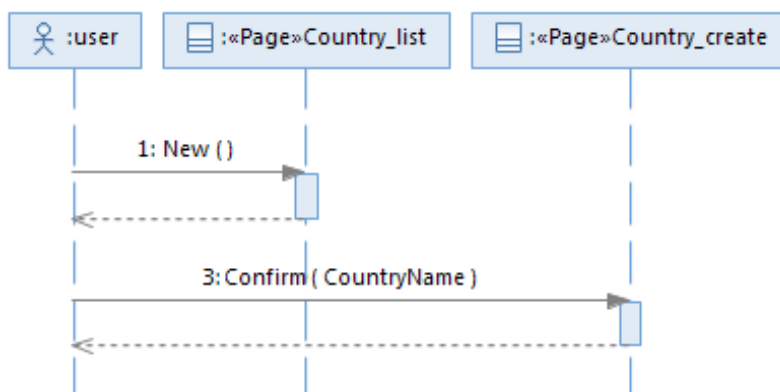


FIGURE 28 - CREATE COUNTRY SEQUENCE DIAGRAM

It then looks for groups of two tables related with an FK in a 1-N relationship. In this case the user has to select the pattern which best fits the graphic user interface structure.

This is the cases presented above, between *Country* and *City*, and *Product* and *ProductPrice* tables.

To see both cases of a 1-N relationship, Figure 29 shows the sequence diagram for the creation of Products, and Figure 30 for the creation of Cities.

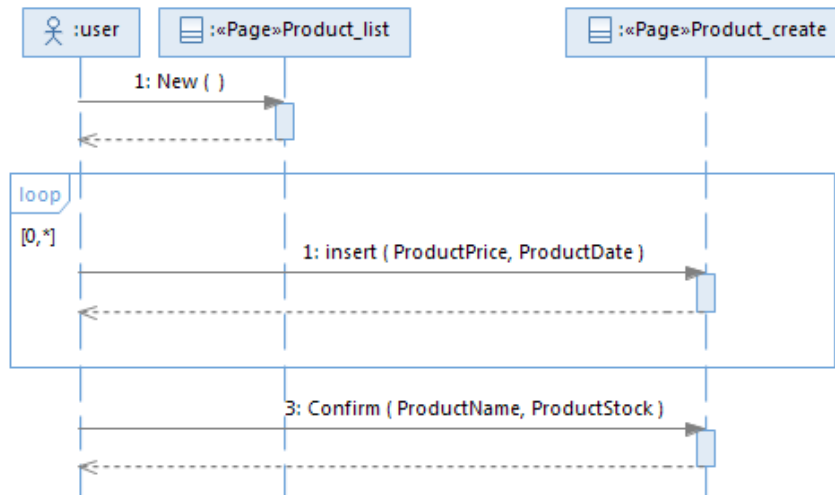


FIGURE 29 - CREATE PRODUCT SEQUENCE DIAGRAM

It is interesting to note that when the user has to select a value that already exists in the database (as the *CountryName* when creating a *City*) it is represented with an operation called *Select*, and when the user has to insert data (that it will be stored in the associated table, as in the case of *ProductPrice*) it is represented with an operation called *Insert*. This information will be very useful in the generation of test cases and test data. It is also important to see that for the case of *Product*, as it is possible to associate many lines of the pair price and date; the insertion of these values is in a loop block.

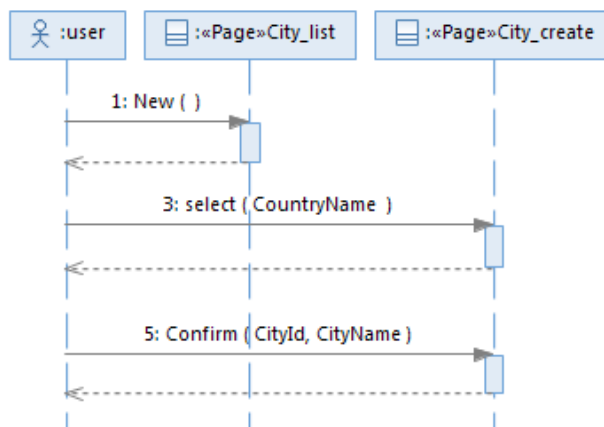


FIGURE 30 - CREATE CITY SEQUENCE DIAGRAM

After this, it looks for groups of three related tables, one participating as a relation-table. For example, for *Invoices* and *Products* there is an N-N relationship, implemented with the table *InvoiceLine*, which has a foreign key pointing to each referenced table, and with extra information, for instance, where *InvoiceLineQty* refers to the number of products selected in the invoice. Actually, the case of *Invoice*, involves four tables, because *Invoice* also references the *Client* table. The page for creation can be seen in Figure 31, and the representation of the user interaction and navigation appears in Figure 32.

Ajax Sample

English Español Português

Recents: [Work With Invoices](#) Invoice

Menu

- [Search](#)
- [New Client](#)
- [New Invoice](#)
- [Work With Cities](#)
- [Work With Clients](#)
- [Work With Countries](#)
- [Work With Invoices](#)
- [Work With Products](#)

Invoice

Id: 0

Date:

Description: Inv: 11/22/13

Client First Name:

Client Balance: 0.00

Client Address:

Invoice Line

Line Id	Product Name	Stock	Price	Line Quantity	Line Amount
0	<input type="text"/>	0	0.00	<input type="text" value="0"/>	0.00
0	<input type="text"/>	0	0.00	<input type="text" value="0"/>	0.00
0	<input type="text"/>	0	0.00	<input type="text" value="0"/>	0.00
0	<input type="text"/>	0	0.00	<input type="text" value="0"/>	0.00
0	<input type="text"/>	0	0.00	<input type="text" value="0"/>	0.00
[New row]					

Sub Total 0.00
Taxes 0.00
Total 0.00

Built on GeneXus Technology

FIGURE 31 - INVOICE CREATION WEB INTERFACE

In this example it is also interesting to see that it is easy to distinguish between the data that has to match with the database (existing clients and existing products) and that inserted by the user (date and number of products for each invoice line).

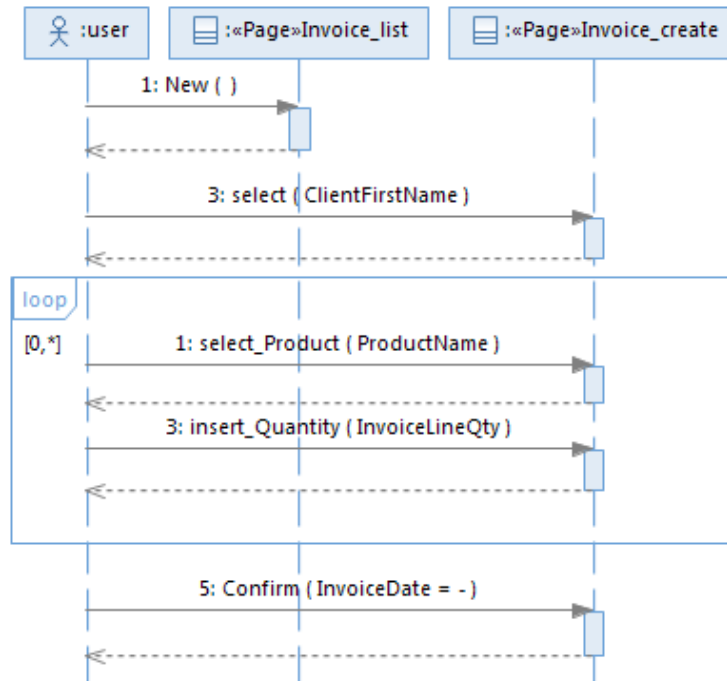


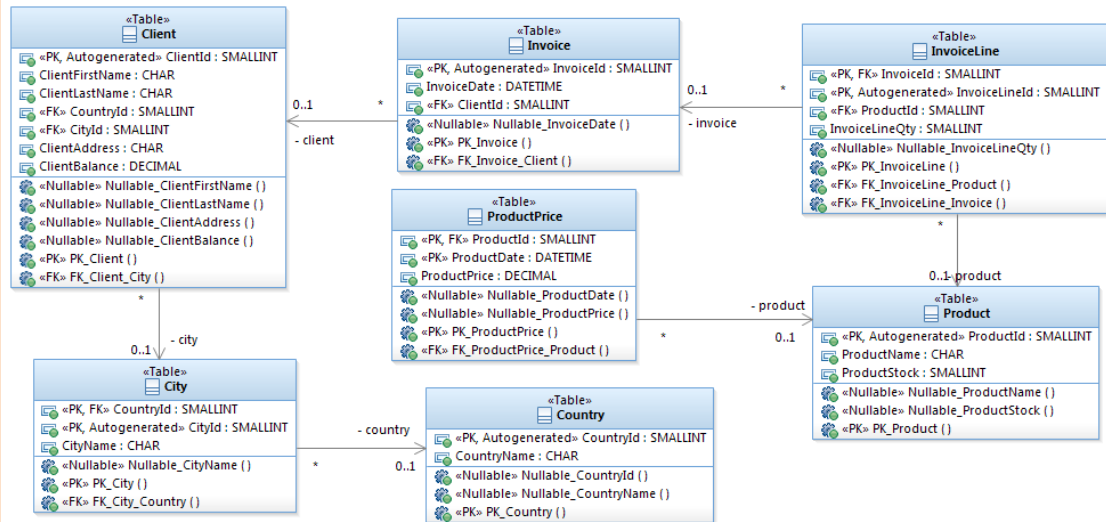
FIGURE 32 - CREATE INVOICE SEQUENCE DIAGRAM

Of course, more patterns could be considering involving more tables or different structures with the same number of tables, but the scope for our study will be those presented in this section.

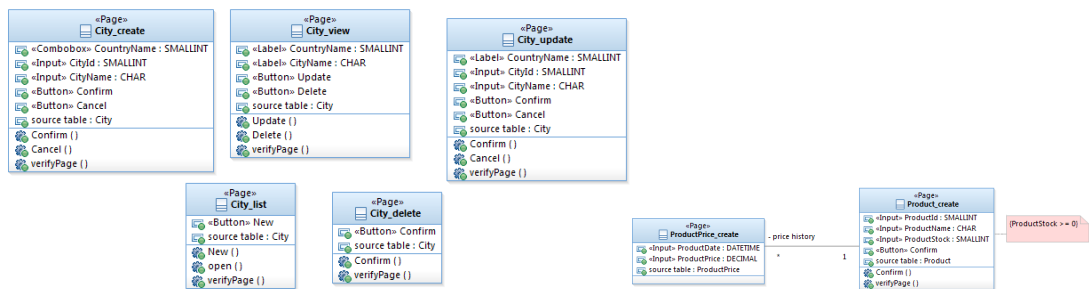
4.4. CONCLUSION

Figure 33 presents a summary of the results obtained after applying the methodology explained in this chapter. Basically, from the database schema and with the assistance of the user, DBesTest generates an Information System Model including a Data Model, a Graphic User Interface (structure and navigation) and Business Rules defined on that basis.

DATA MODEL



GUI MODEL STRUCTURE



GUI MODEL NAVIGATION

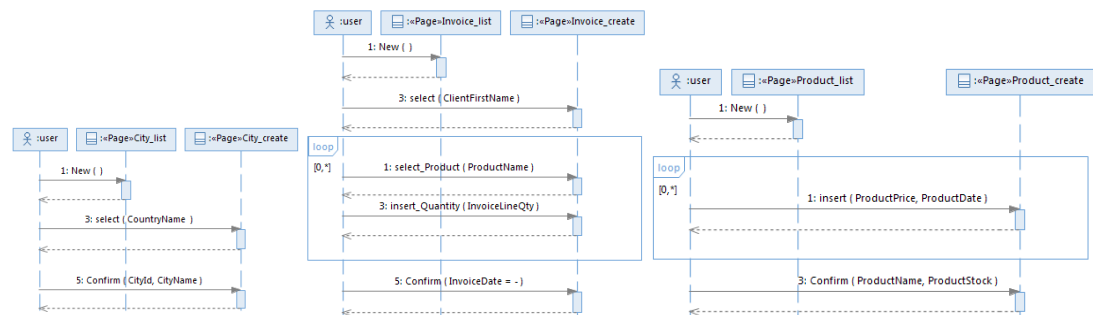


FIGURE 33 - SUMMARY OF THE CHAPTER

"Quality is value to some person."

Jerry Weinberg

"Quality is value to some person who matters."

Cem Kaner

CHAPTER 5. AUTOMATIC GENERATION OF FUNCTIONAL TEST CASES

This chapter evidences the utility of the Information System Model in order to generate, through model transformations, functional test cases represented in a UML-TP model. *MBT is expected to allow more adequate software testing because it is rooted in automated procedures, avoiding manual error prone activities* [123]. In this chapter the model-to-code transformation is explained, showing how the test code is obtained.

5.1. INTRODUCTION

This part of the MANDINGA methodology consists mainly of two phases:

- **Test Model Generation.** The Information System Model is processed using pattern-matching techniques to automatically generate the test model through model-transformations.
- **Test Code Generation.** The test models are transformed into test code, obtaining executable test cases.



In order to take an integrated approach, the model-to-model and model-to-text transformations were integrated in **DBesTest**, allowing the user to execute them in a very easy way in the Eclipse environment. The remainder of this chapter explains each part in detail, with an example using the AjaxSample application, showing how the transformations can be executed from the plugin. The graphic representations of the UML models were prepared with Rational Software Architect.

It is important to emphasize that these techniques do not offer all the possible error situations, only those related to the management of the database (the creation, reading,

updating and deletion of data, known as CRUD operations). Let us say that the patterns generate simple test cases but with low cost and high value.

Performing this kind of testing at the outset improves the productivity and morale of testers when they verify more complex use cases. If someone is trying to test a complex use case and becomes stuck with this kind of problems then they may lose focus and time. The focus of our approach is to start with CRUD operations, which are the core part of most of the SUT, allowing the tester to concentrate on more complex situations and avoiding them being blocked by simple problems, reducing interaction between testers and developers.

This is also based on what De Millo et al. [124] call the *coupling effect*, saying that “*test data that distinguishes all programs differing from a correct one by only simple errors is so sensitive that it also implicitly distinguishes more complex errors*”. As the reader will see, the framework starts by focusing on simple errors, and then applies more sophisticated testing techniques in order to dig into better testing coverage.

5.2. TEST MODEL GENERATION

This section presents the functional test model generation. Among other things, it defines the Test Architecture, the patterns to generate test cases and test data and a set of black-box coverage criteria for Information Systems considering the lifecycle of their entities and relationships. The criteria are defined by considering interesting situations from a testing point of view, and the result is represented in a UML model using the UML Testing Profile (UML-TP, also referred in some publications as UTP, as presented in Chapter 2).

It is important to emphasize that the main goal of the test generation strategy is not the testing of the database schema, instead, the goal is to test the applications that use it.

The transformation of the ISM into the test model is made with ATL transformations and it is presented in three main stages: one for the test architecture model, one for the group of test case behavior models (including the oracle) and finally for the test data model. Figure 34 shows the main inputs and outputs of the ATL transformations.

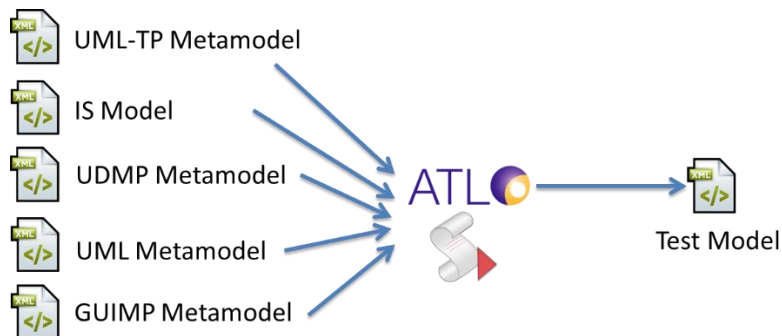


FIGURE 34 - INPUTS AND OUTPUTS OF THE TRANSFORMATION

As a result of this execution, a UML-TP compliant file is obtained, holding all the generated test cases and the remaining required elements. It is the input for the subsequent step, the test code generation.

5.2.1. TEST ARCHITECTURE

The test architecture offers the structural view of the UML-TP model, showing the fundamental organization of the test elements, how are they grouped, related and how they can communicate with each other. Typically, it is represented with a class diagram.

Figure 35 shows how the model-to-model transformation (in our case implemented with ATL code) processes all the entities in the ISM with the corresponding GUI elements and the navigation defined, and generates the Test Architecture in the Test Model.

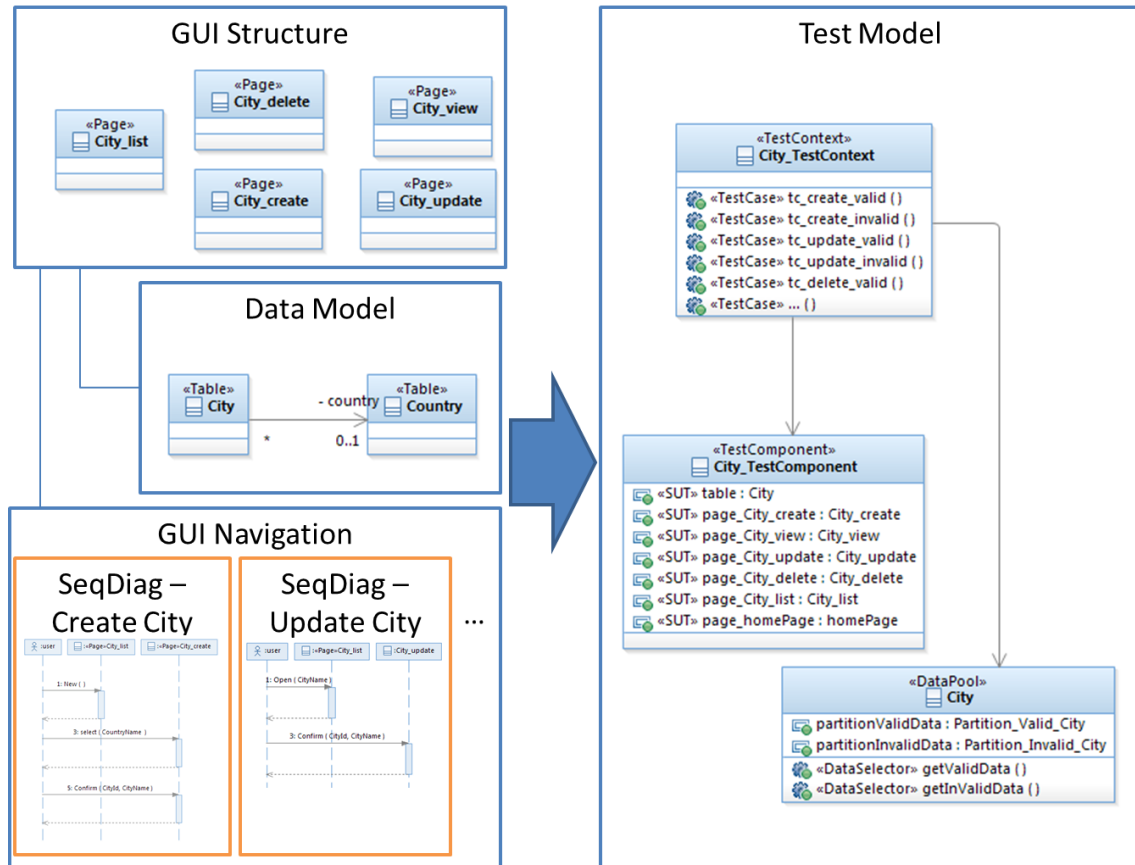


FIGURE 35 - ATL CODE FOR THE TEST ARCHITECTURE

For each pattern (including a substructure of the data model, its corresponding graphic user interface elements and its navigation for the CRUD operations) a set of test elements are created, and included in the test architecture.

Among other things the following components are created for each pattern:

- A *Test Context*, which contains the generated test cases as operations, defines the test suite.
- A *Datapool* for the test data (each *Datapool* has *Data Selectors* in order to provide *Data Partitions*, specific test data in different tests situations).
- A *Test Component* to interact with the SUT.
- *SUT* Classes: wrappers to pages to invoke actions in the test cases and validations at a graphic user interface level, and tables to invoke validations at a database level.

The test architecture imposed by the UML-TP defines a clear separation of responsibilities for the test case behavior, test data access and the interaction with SUT.

The separation between test behavior and test data allows the same test flow to be executed with different test data, which will be stored in a separate structure of the test model called *datapool*. This approach is known as *data-driven testing* [103], and the main advantage is that the tester can add easily new test cases simply by adding new rows to the datapool, indicating new interesting situations to cover with the data inputs.

The separation between the test flow and the component which executes the actions on the system allows the user to modify these aspects independently. This architecture fits well with one of our goals which is the possibility of defining test cases and the test data to be used/executed in different platforms (e.g. if the same functionality of the system can be accessed by the web user interface or by a mobile application). The test knowledge is thus defined once and used many times.

Once the static structure of the test model has been generated, the behavior of each test case and the test data must be also described in terms of the UML-TP language. When the test cases are generated, they are added to the corresponding Test Context, and while the test data is being designed, the test partitions and data selector operations are added to their corresponding datapools.

5.2.2. TEST CASE GENERATION

Each Test Case that is defined as an operation of a Test Context must have its corresponding activity diagram, sequence diagram or state machine diagram to represent the expected behavior. The behavior diagram is specified as the implementation method of the corresponding test case definition contained in the same Test Context.

In our case, the test cases behaviors are represented with sequence diagrams such as the one presented in Figure 36. According to the UML-TP specification, a test case requires three steps: (1) obtaining the test data, (2) executing the test case against the SUT and (3) obtaining the test case verdict.

Remember that (as explained in Chapter 4) when the user has to select a value that already exists in the database (such as the *CountryName* when creating a City in the example of Figure 36) it is represented by an operation called "Select", and when the user has to insert data (which will be stored in the associated table, as in the case of *ProductPrice*) it is represented by an operation called "Insert".

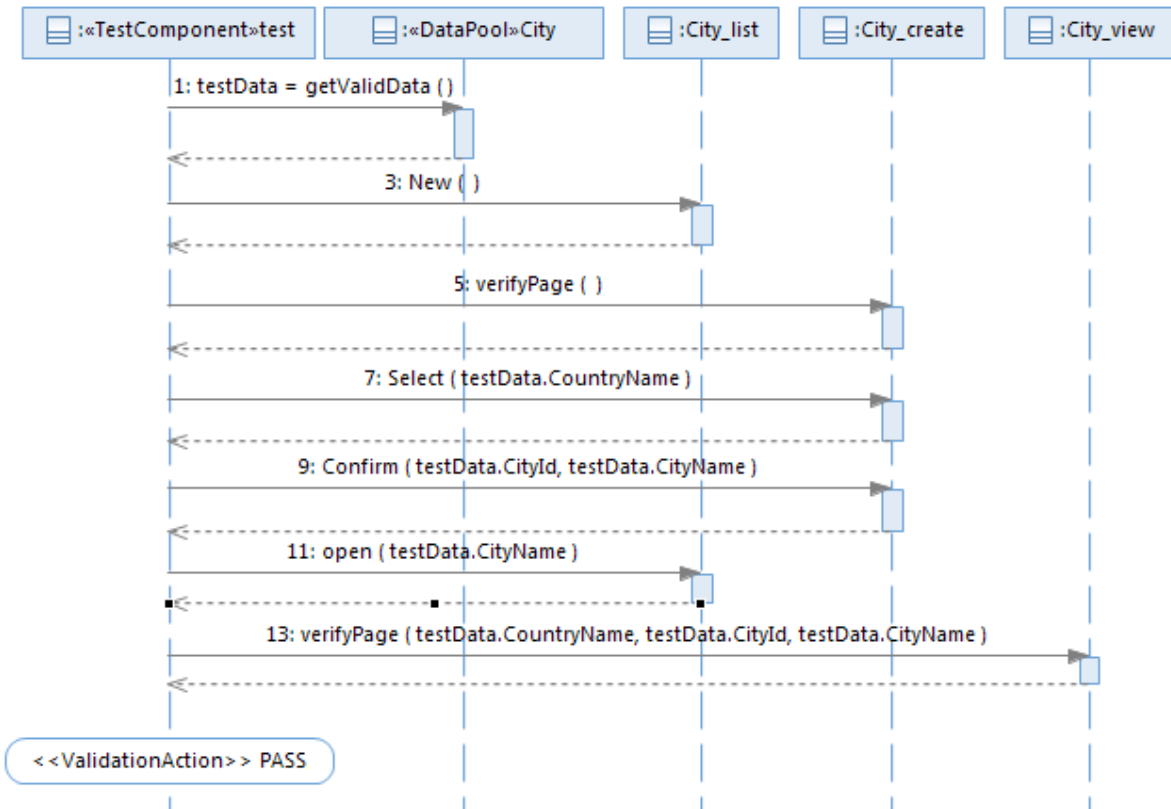


FIGURE 36 - SEQUENCE DIAGRAM REPRESENTING THE BEHAVIOR OF THE TEST CASE OF THE CREATION OF CITIES

The test data is obtained from the Datapool, invoking a Data Selector method, which returns a Data Partition with the test data. When the test executes operations on the SUT, it uses this data (see the parameters on the invocation to “Confirm” method). The test case then invokes a *Validation Action* telling the *Arbiter* that (if the flow of the execution reached this point) the *Verdict* of the execution of the test case is *PASS*.

Each validation is represented as an *UML invariant* with the stereotype *Validation Action* setting the verdict.

The test case asks for test data to the datapool instance, and then invokes actions into the different objects that interact with the SUT using this data. The different invocations are operations of the SUT elements, which are operators of the GUI model from the ISM, because these are the actions that a user would execute. Therefore, each lifeline with which the test component interacts corresponds to a class from the graphic user interface structure. The sequences of invocations are taken from the navigational specification.

After each transition from one page to another, the test case verifies that it is visiting the corresponding page according to the navigational model. This is the reason for

having the “verifyPage” operation on every page element. It will compare, at least, the current page with the corresponding URL (in the example, when visiting City_Create), and that the different elements of this page are shown with the correct data (when visiting City_View, it receives all the data as a parameter in order to verify whether this data corresponds to the actual data in the page).

The test cases in the test model do not have specific data, they are classified as *valid* or *invalid* (using *Data Partitions*), and in the following phase (when the model is transformed into test code), taking this categorization and the data types from the data model, they are instantiated with representative values.

The operations *create*, *update* and *delete* force a change in the database state only when they are executed with valid data, otherwise, the state should not change.

Test cases must define the test data to use, the test flow to execute, and with the same level of importance, they must provide a verdict of the execution, reporting whether the execution was passed or if a failure was found. The element responsible for giving a verdict is known as the oracle.

It is important to note that two kinds of test cases are generated:

- Tests for atomic operations: for specific functionalities such as creation, updating and deletion of the instances.
- Tests for the lifecycle of entities, based on the functional cycles, combining creation, reading, update and deletion of different entities.

Furthermore, there are two different considerations for the test case generation strategy. On the one hand, the test case generation is based on well-known coverage criteria (presented in Chapter 2), and on the other hand, with a pattern-matching strategy. Both are explained in the following subsections.

Another important aspect to highlight is that, even though different coverage criteria are reached by the automatic test case generation strategy, one of the most important things is that after this the tester can easily model new test cases according to their experience or desires.

5.2.2.1. COVERAGE APPROXIMATIONS

Coverage analysis is used: (1) to know the areas of the system that the test cases have exercised; (2) to find the unexplored building blocks; (3) to create new test cases to exercise those unexplored building blocks; (4) in some situations, so that achieving a predefined coverage without finding new errors could be used as a stop testing criteria [108].

Given that, in our case, test cases are generated from the Information System Model, which is a UML model containing class diagrams and sequence diagrams, there are some well-known coverage criteria that can be adapted for our situation. These coverage criteria induced us to define the test patterns then presented.

5.2.2.1.1. COVERAGE ON CLASS DIAGRAMS

Since the central component of the information system model is a class diagram corresponding to the system's data model the applicable coverage criteria are some of those proposed by Andrews et al. [111]:

- **Class Attribute (CA):** the test suite should make use representative values for each attribute in each class.
- **Association end Multiplicity (AEM):** the test suite should make use of every representative pair of multiplicities for the associations of the model.

These coverage criteria were designed to test UML specifications where an object oriented model defines the behavior of the system. In our case the criteria were applied for a data model instead of an object model, so some aspects were adjusted in order to make it applicable. For example, one of the *adjustments* was to take it into consideration that between two entities in the Data Model the relationship has certain limitations because it is implemented with foreign keys. For instance, if entity *A* has a foreign key to entity *B* then it is not possible that these relationship boundaries were 1..* in *A* (it will be explained in more detail in the section below) but in class diagrams these kind of associations are allowed.

Another *adjustment*, and perhaps the most important consideration, is that the operations under test are *create*, *read*, *update* and *delete* of each entity. This is important for determining the oracle, because the expected results of these operations are well-known (if a test executes a creation with valid data it should create a new instance with the corresponding data, etc.). There is another consideration related to the multiplicity of the associations: according to the definitions given in the foreign keys it is possible to have different kinds of association multiplicities, and for each one it is necessary to consider a special situation about the boundaries of the association end multiplicities.

To apply these criteria the framework will generate test cases to cover such situations for every substructure of the data model that matches any of the criteria, which means that for each class it will generate test cases according to *CA* criterion, and for each association will generate test cases according to *AEM* criterion.

5.2.2.1.2. COVERAGE ON STATE MACHINES

The lifecycle of the entities in our system can be modeled as a State Machine, which is equivalent to the regular expression: $C \cdot R \cdot [U \cdot R]^* \cdot D \cdot R$.

This representation is suitable for processing with the coverage criteria defined for state machines [14]: going over all the states, all the transitions or running all the input/output transition pairs for each state.

Different test cases will be generated for each CRUD operation. Test cases combining these atomic test cases are then generated in order to test different paths on the associated state machine.

5.2.2.2. TEST PATTERNS

Our framework includes a test pattern repository/catalog, including some patterns identified in order to try to reach the test coverage explained in the previous subsection.

Bertolino mentioned that one of the big challenges for testing research [125] is that it should be possible to systematize test pattern identification, materializing every new pattern identified through the experience, and allowing testers to reuse this knowledge in new situations. In that way, it would be possible to identify the most effective patterns to test our systems, generating a test catalog of well-proved test patterns.

With this idea in mind, the ISM is processed looking for model-pattern occurrences and automatically generating test cases for them, thus composing a test model.

All the patterns categorize test data as **valid** and **invalid** for each parameter of the GUI (user inputs), according to the data type obtained from the ISM, and from business rules defined on it. In this way it is possible to design representative test data for each attribute. This is explained in more detail in section 5.2.3, whilst in this section, only there will be references to “valid” and “invalid” Data Partitions.

For example, if a test invokes a *create* or *update* operation there are two different situations that determine the expected result:

- If using invalid data for the inputs then the test should check that the instance was not created or updated (all the attributes in the database keep their original value).
- If using valid values for all the inputs the test should check that the instance was created correctly with the corresponding values.

The patterns are expressed as Model-to-Model transformation rules (implemented with ATL) which explore the Information System model looking for occurrences of the defined

substructures. The target metamodel in the transformation is UML-TP: for each occurrence matched by the ATL rules, the transformation will generate different elements of the UML-TP. The main advantage of using a model-transformation rules approach for pattern-matching is that it is easier to add new patterns, for example once a new interesting situation for testing is designed through experimentation, or whenever a user considers it is important for their domain.

Below, the following subsections show an initial design for patterns with one, two and three tables, describing the different situations and the test cases that will be generated.

5.2.2.2.1. ONE-TABLE PATTERNS

The most basic pattern designs test cases based on one table. This pattern applies (generally) for those tables without a foreign key, which present GUI elements for the CRUD operations. Table 10 shows, as an example, the one-table pattern applied to the creation of the entity “Country”, Table 11 for the update and Table 12 for the deletion. Each example presents the behavior diagram (sequence diagram) which originates the generation of the test case, and then the sequence diagram for the generated test case using the valid data partition. It can be seen that the pattern generates a test case for create, update and delete, and it uses read to verify each action (note that the transformation considers the read operation as is presented in Figure 37).

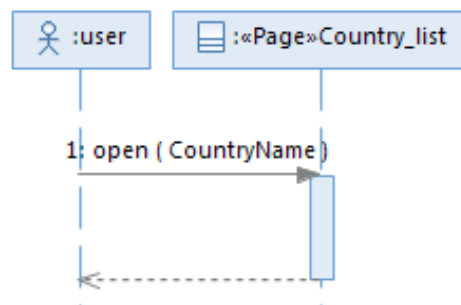


FIGURE 37 - READ OPERATION FOR COUNTRY

To clarify the intent of each test case, the examples presented in the tables also show the different screens of the application when the test case is executed.

TABLE 10 - ONE-TABLE PATTERN – TEST GENERATION TO CREATE COUNTRY

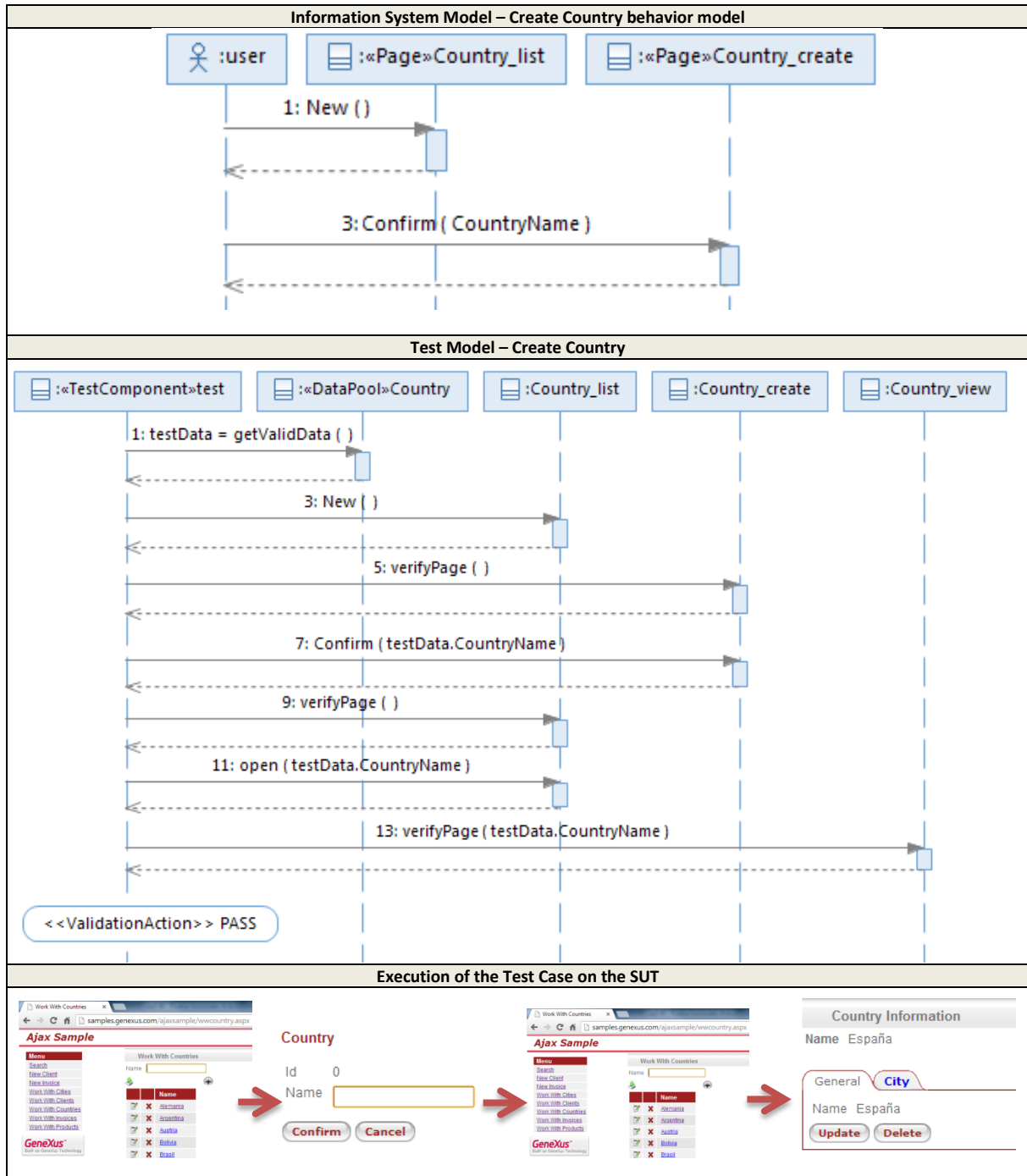


TABLE 11 - ONE-TABLE PATTERN – TEST GENERATION TO UPDATE COUNTRY

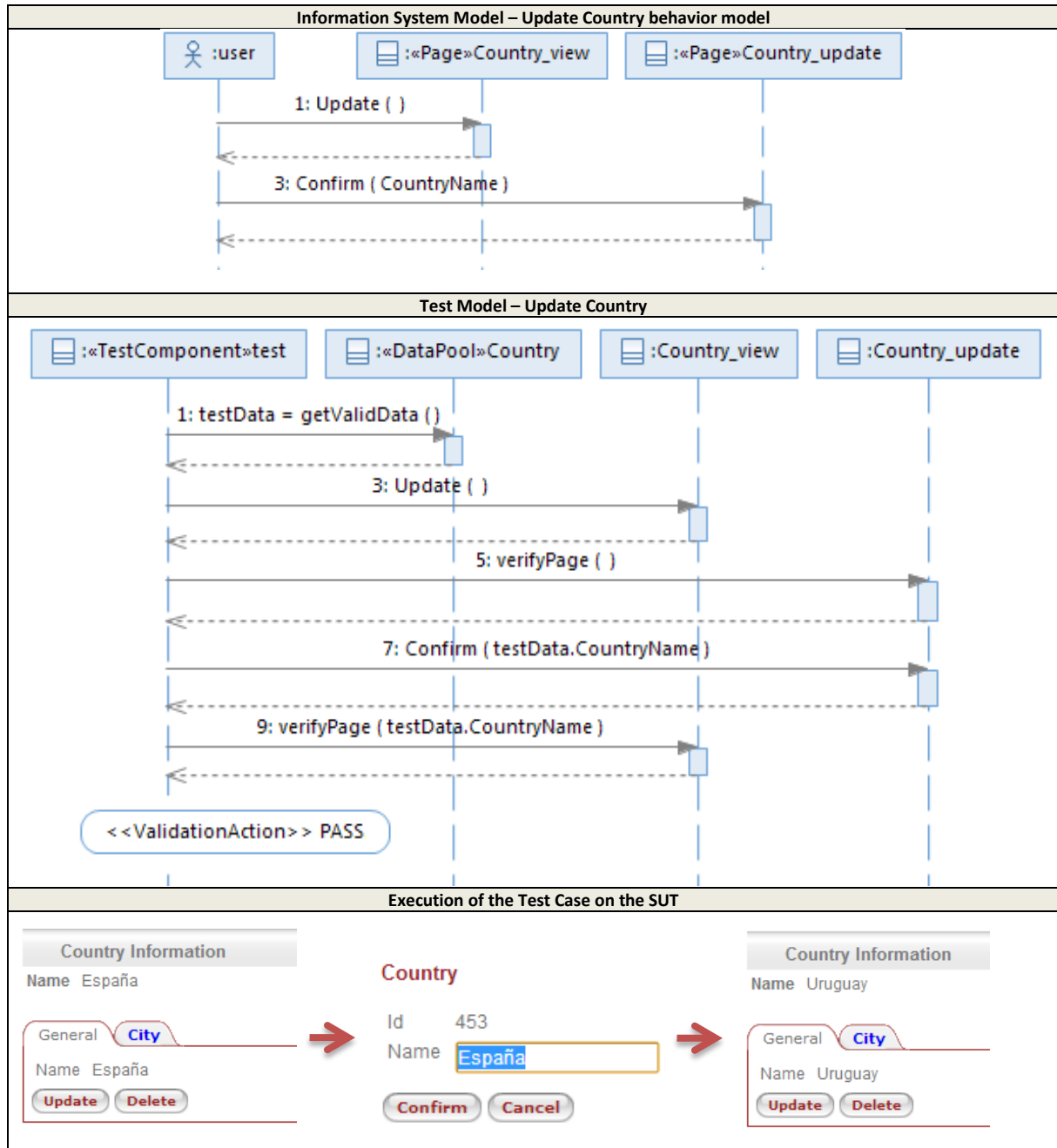
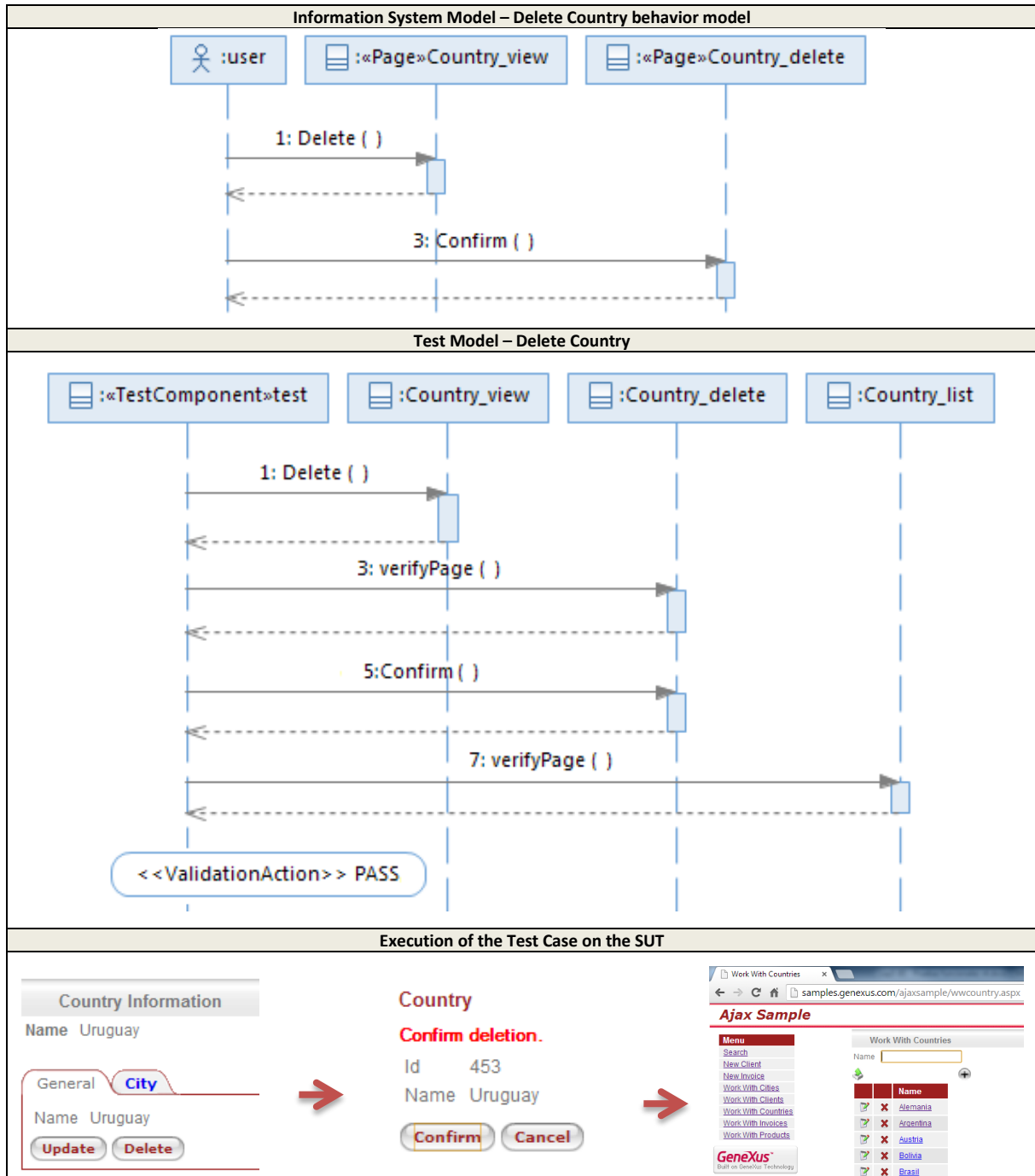


TABLE 12 - ONE-TABLE PATTERN – TEST GENERATION TO DELETE COUNTRY



Having the atomic test cases for the CRUD operations, the test generator also applies the CRUD coverage [103] to considerate the whole life cycle of an instance, which implies that the operation is tested in the sequences that can be obtained by expanding

the regular expression: $C \cdot R \cdot (U_i \cdot R)^* \cdot D \cdot R$, where the U_i represents each operation that updates a different attribute. Applying this to the example of the entity *Country* generates the following test sequence (see Figure 38):

- Create Country and Read Country
- Update Country (each attribute) and Read Country
- Delete Country and Read Country (which should not find it)

These test cases can be combined in different ways, covering different sequences.

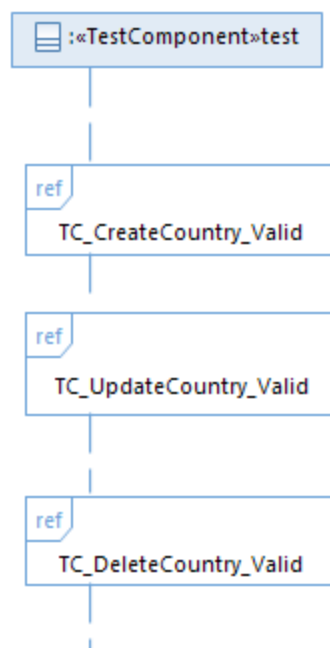


FIGURE 38 - TEST CYCLE FOR COUNTRY

Note that the final state of the database is the same one than the initial, what is convenient in order to have independent test cases: the execution order does not affect the expected result.

5.2.2.2.2. TWO-TABLE PATTERNS

This pattern is applied (generally) for two tables associated by a foreign key, which present GUI elements for the CRUD operations, as the example of *City–Country* and *Product–ProductPrice* already presented (see Table 13 where it is presented again).

TABLE 13 - EXAMPLES OF GUI STRUCTURES FOR THE SAME TWO-TABLE PATTERN

Database substructure	GUI for the creation of these instances												
	<p>City</p> <p>Country Name <input type="text"/></p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p> <hr/> <p>Country</p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p>												
	<p>Product</p> <p>Id <input type="text" value="0"/></p> <p>Name <input type="text"/></p> <p>Stock <input type="text" value="0"/></p> <p>Price</p> <table border="1"> <thead> <tr> <th>Date</th> <th>Price</th> </tr> </thead> <tbody> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td><input type="text" value="//"/></td> <td><input type="text" value="0.00"/></td> </tr> <tr> <td colspan="2" style="text-align: center;"><input type="button" value="[New row]"/></td> </tr> </tbody> </table> <p><input type="button" value="Confirm"/> <input type="button" value="Cancel"/></p>	Date	Price	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="text" value="//"/>	<input type="text" value="0.00"/>	<input type="button" value="[New row]"/>	
Date	Price												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="text" value="//"/>	<input type="text" value="0.00"/>												
<input type="button" value="[New row]"/>													

When there are two or more related tables the criterion most considered by our approach is *AEM* from Andrews et al. [111], and for this it is necessary to test associations between entities with representative multiplicities. To do so, the generated test cases consider to try each instance associated with 0, 1 and 2 instances of the other table. At this point, it is believed that associating two instances is good enough to test the multiplicity “*”.

The test set includes test cases to associate instances covering the different representative multiplicities. The association ends of two tables (a referencing and a referenced table) could have a multiplicity of 0..1 (if the foreign key allows nulls in the referenced table, or if the foreign key is unique in the referencing table), 1 (if the foreign key does not allow nulls in the referenced table) or 0..* (in the referencing table). Therefore, it is possible to have the combinations presented in Table 14.

TABLE 14 - COMBINATION OF MULTIPLICITIES IN A FOREIGN KEY

Referencing table	Referenced table
0..1	0..1
0..1	1
0..*	0..1
0..*	1

This restriction comes from considering the structures that can be implemented in a database schema with foreign keys. For example it is not possible to implement a 1:1 relation with foreign keys between two tables (it would be necessary to manage these association ends limits in the code of the business layer).

THE ABOVEMENTIONED EXAMPLES *CITY-COUNTRY* AND *PRODUCT-PRODUCTPRICE* CORRESPOND TO THE LAST SITUATION: 0..* → 1. IT IS NECESSARY TO CONSIDER BOTH, BECAUSE THEY SHOW TWO DIFFERENT EXAMPLES OF TWO RELATED TABLES, WITH DIFFERENT PRESENTATION LAYERS FOR THE SAME DATABASE SUBSTRUCTURE. TO EXEMPLIFY THE GENERATION OF TEST CASES FOR ONE OF THOSE SITUATIONS, TABLE 15 SHOWS THE GENERATED TEST CASES FOR THE CREATION OF A *CITY*, AND

Table 16 for the creation of a *Product*. It includes the behavior model in the ISM which is processed, the generated sequence diagram specifying the test case behavior, the data model in the test model architecture, and the screens of the execution of the corresponding test case specification.

The examples presented in the tables only show the creation, but there will also be test cases for update and delete, and using valid and invalid data.

TABLE 15 - TWO-TABLE PATTERN – TEST GENERATION TO CREATE CITY

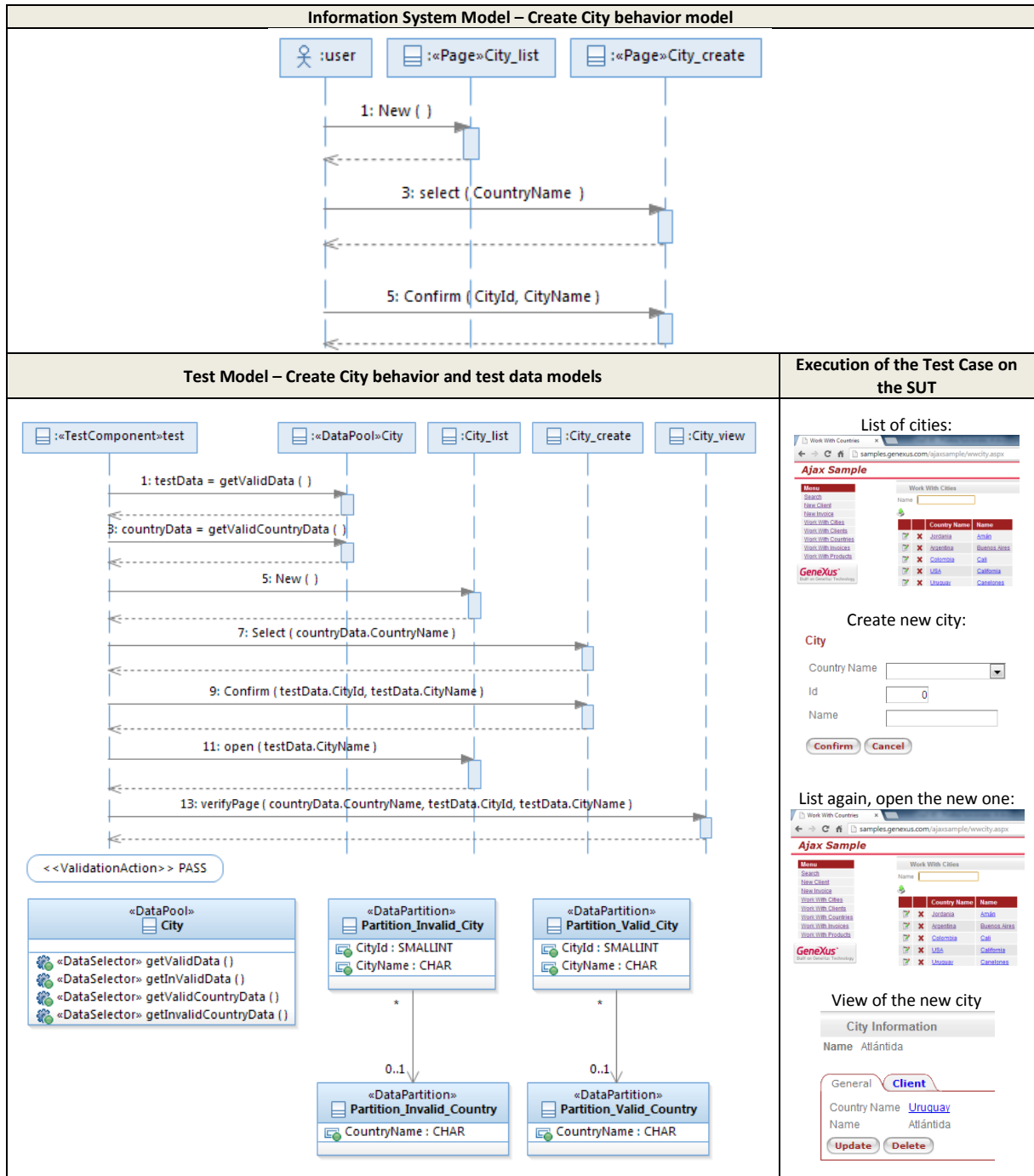
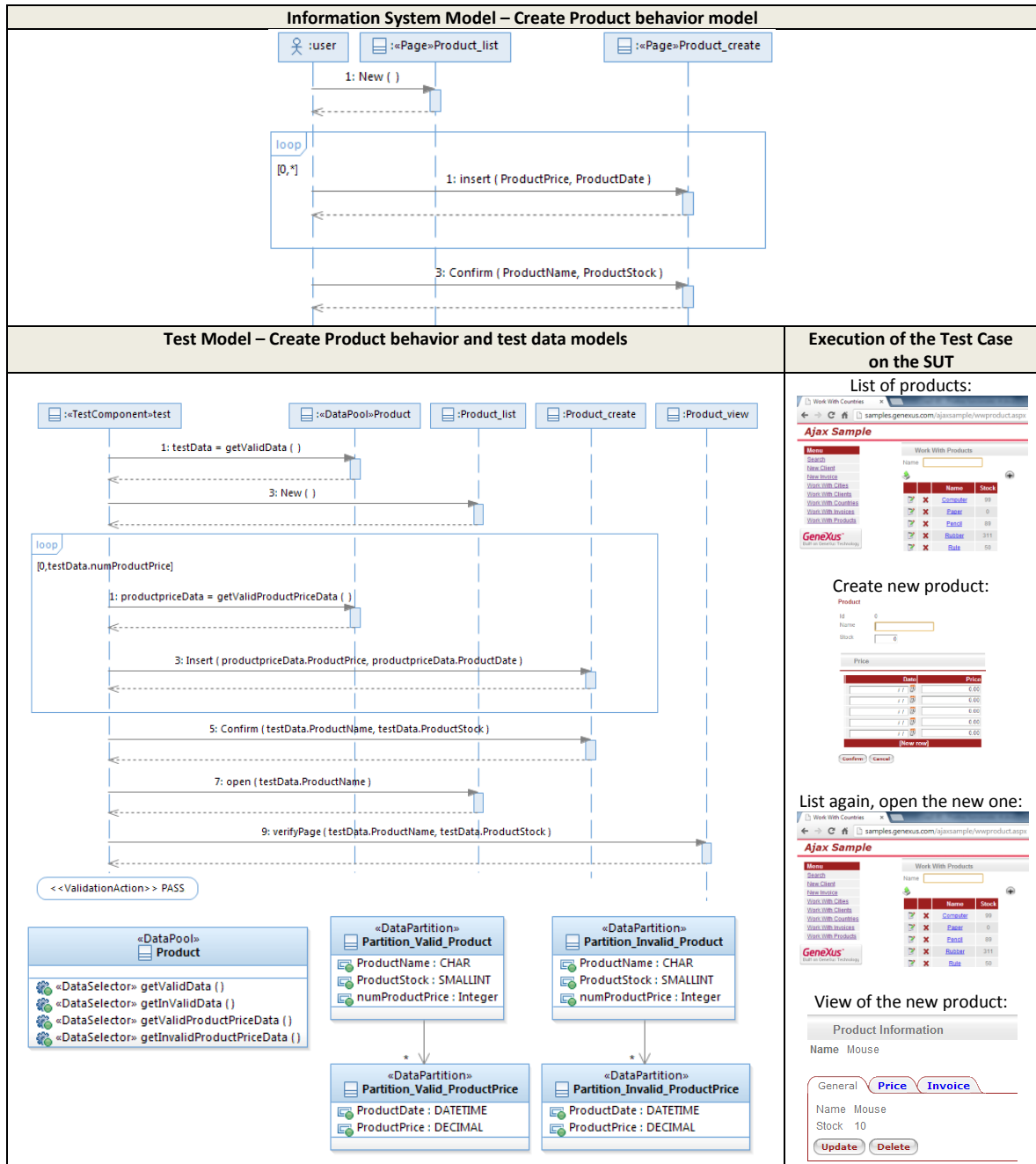


TABLE 16 - TWO-TABLE PATTERN – TEST GENERATION TO CREATE PRODUCT



It is interesting to note that in the case of *Product*, where it is possible to insert different numbers of *ProductPrice* rows in the interface because of the relation 1-N, the generated test case accesses the data from two different data partitions which are related. The datapool is able to provide the related data. In the data partition with the information of the *Product*, there is an extra attribute indicating how many *ProductPrice*

rows should be inserted for this test case. This attribute is used in the loop to count the number of times the data in the data partition for *ProductPrice* is obtained, and then used in the GUI by the test.

The *Data Partitions* for *City* have the same structure as the associated tables of the pattern (one for *City* and one for *Country*, with the same association), but in this case each *City* has only one *Country* associated. More details about the test data can be found in Section 5.2.3.2.

In the case of *City*, which has an input that corresponds to a foreign key, the test for the *create* operation considers that the valid data for this input is the existing keys in the referenced table and invalid data when it does not exist (it is interesting to test the creation of a *City* which references a *Country* that does not exist).

With the atomic operations it is possible to design different test cycles. The following sequence was designed in order to reach the CRUD and the AEM coverage criteria:

- Create City (without a Country) / should fail (rel.: 1 – 0)
- Create Country
- Create City (with a valid Country) / (rel.: 1 – 1)
- Delete Country (should fail, because it has a reference)
- Update City (for each attribute)
- Create City with same Country (rel.: 2 – 1)
- Delete both Cities
- Delete Country

Note that in this case the original database state is also preserved at the end of the test case execution.

5.2.2.2.3. THREE-TABLE PATTERNS

The two-table pattern does not include the binary relation *many to many* because at the database level it is implemented with three tables: two tables with the data of the entities, and another auxiliary table to store the relationships, referencing the primary keys of the entities, and defining its own primary key as the union of the primary key attributes. There may also be more attributes in the auxiliary table to store data related to the association.

In *AjaxSample* there is an example in the relationship between *Invoice* and *Products* (see Figure 39), where the same invoice can have many products, and the same product can be included in different invoices. The relation table *InvoiceLine* has some attributes to

store information about the relationship between the *Invoice* and *Product*, for example *InvoiceLineQty* indicating the number of products selected for this invoice.

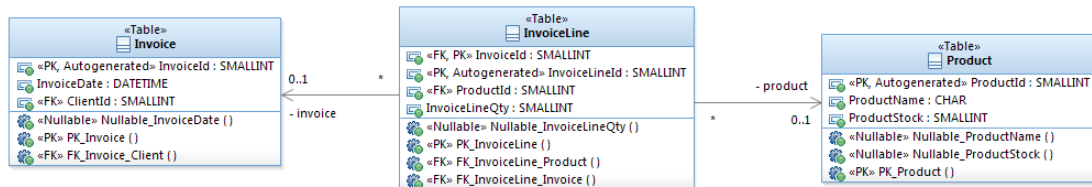
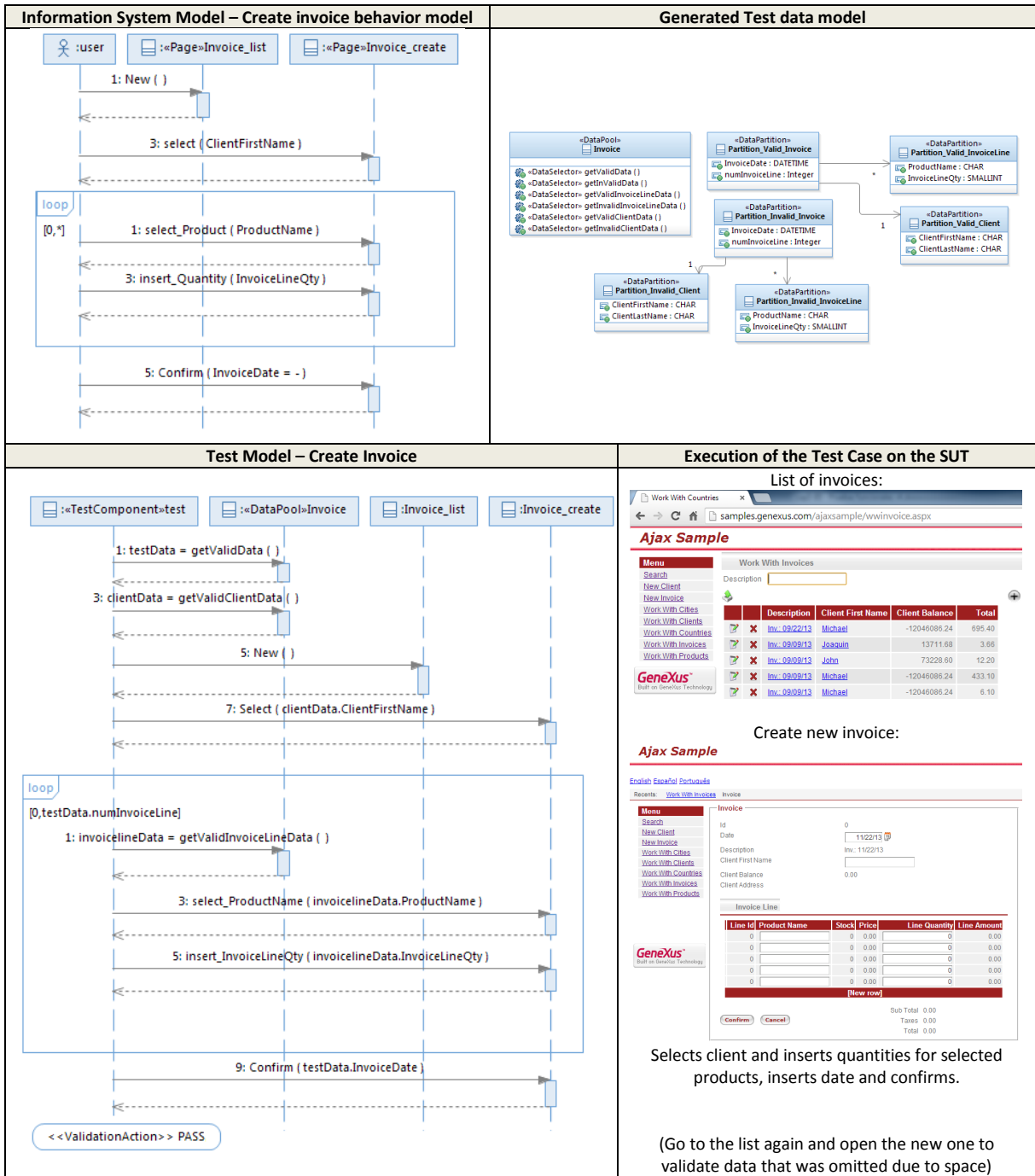


FIGURE 39 - DATA MODEL FOR AN N:N RELATIONSHIP

In this example it is interesting to note that it is possible for the tester to add extra information to the data model, in order to validate some aspects of the logic that cannot be represented in the database schema. In the relationship between *Invoice* and *Product* it does not make sense to have an invoice without any product, but this cannot be implemented in the schema, it must be managed in the logic, therefore, there should be a business rule with the restriction, or the user should change the model manually (simply changing the association end multiplicity from “0 – *” to “1 – *”), and there would be a test to check it.

Table 17 shows the design of the test case for the creation of invoices. It also shows the data partitions involved.

TABLE 17 - THREE-TABLE PATTERN TRANSFORMATIONS



It is possible to see that the test case manages the same strategy as for the two-table pattern, and actually, in this case, there are more tables involved (Client), which shows the extensibility of the idea for more complicated structures.

5.2.3. TEST DATA MODEL GENERATION

Test data design is based on the inputs defined in the GUI and data models, and their relationships. It is also important to consider the business rules and data types to determine the equivalence partitions and boundary values. The test data is generated as a group of datapools, data partitions and data selectors which are used by the generated test cases. Since the test data is separated from the test flow, it is possible to follow the data-driven testing approach [103].

5.2.3.1. EQUIVALENCE CLASS PARTITIONING

Each test data is categorized as *valid* or *invalid*, according to the defined constraints of the model. If, for example, there is a business rule stating that a certain attribute must be greater than zero, it is possible to infer two equivalence classes for this attribute: one with valid values (greater than zero) and another with invalid values (less than or equal to zero). Using this classification, the test cases are generated with oracles that use this information, supported by this procedure: if one of the values of the input data is invalid, check that the operation failed, and if all the input data are valid, check that the operation was successful. For example, if the operation is *Create*, then the test case with valid data must verify that the instance was created, and the one with invalid data must verify that the instance was not created. This defines the oracle at a data level, and the test cases must consider that they must verify the result according to the validity of the input data.

It is important to note that, in this way, this approach is facing a problem that is often not pointed out by the automatic test case generation approaches, that is the oracle generation: determining when the expected result is to pass or to fail. In our research this approach has been named “**data based oracles**”.

The procedure is: for each restriction defined in the Information System Model (taken from the database level structure, data types, business rules, etc.) identify the equivalence classes for each variable and then generate representative test data according to the definition of these equivalence classes. Each class could be valid or invalid.

This thesis suggests that considering the data model it is possible to generate better class partitioning than by only considering the GUI. In order to test the creation or update of *Products* and considering the equivalence partitioning criterion, it is possible to divide the values of the *Stock* variable (in *Product*) in two classes: greater and equal to zero (valid data), or less than zero (invalid data, it does not make sense to have negative stock). But, what happens if the value inserted cannot be represented in the corresponding column of the database? It probably means that the stored value is not

correct as a result of an overflow. Taking advantage of the metadata of the database it is thus possible to design another equivalence class: numbers greater than the maximum representable in the corresponding column. The same idea can be extended to strings and other data types.

These kinds of test cases are not derived from traditional coverage criteria. In these cases, our approach is to improve the equivalence partitioning with the information of the schema. In the section below, a group of different test data selection criteria are defined, in order to consider these situations. Therefore, **our criteria subsumes the equivalent partition criterion** (a coverage criterion subsumes another if every test suite that satisfies the first also satisfies the second [126]). Subsection 5.2.3.3 shows the design of invalid classes based on the data model.

Figure 40 shows what is considered a valid behavior of an information system when it is exercised with test data from (1) a valid class and (2) an invalid class according to the data model:

- The interface does not allow the invalid test data to be entered (for example, a combo box with the elements of a foreign key: you cannot enter an invalid element).
- The interface correctly indicates that the error is due to invalid input data that does not conform the business rules represented in the data model.
- In both cases, the database remains unchanged and the user realizes the problem.

If the test data is valid for the database and for the business rules, then, the IS should allow the data to be entered, and it must be stored in the database.

Thus, the generated test model has two types of test cases: (1) test cases which use valid data (asking the Datapool for valid Data Partitions through the valid Data Selector operation) and verifying whether the operation tested was successful, and (2) test cases which use invalid test data (asking the Datapool for invalid Data Partitions through the invalid Data Selector operation) and verifying whether the operation tested avoided the insertion of invalid data or correctly notified the user.

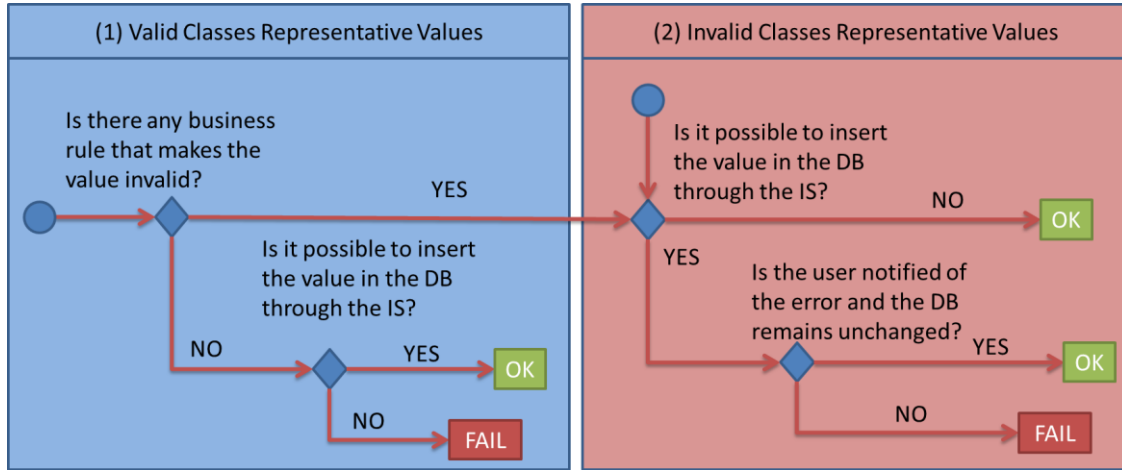


FIGURE 40 – DATA-BASED TEST ORACLE

5.2.3.2. STRUCTURED TEST DATA

Typically, when a form is stored in the database in a structure compound by different tables in the database, the input data must also be modeled as a similar structure. In the examples presented in this chapter, (i) for the creation of a City it is necessary to insert data for city and for the related country; (ii) to create a Product it is necessary to insert data for the product and for the different prices that it has; (iii) to create an Invoice it is necessary to select different products, and indicate their amounts for inclusion in the invoice, and it is also necessary to select a client.

There are two cases to differentiate between in these situations: data which it is filled in by the user on the form and goes to the database, and data that the user has to “select”, or fill in on the form according to previous information existing in the database.

In Information Systems it is very common that the inputs have to be related to the data in the database, presented many times in a list or combo box, or autocompleted with Ajax in an input. For example, the entity “City” has a foreign key to “Country” and the creation of cities requires a valid country name as input to set this relationship.

In the information system model, these situations are differentiated in the sequence diagrams, because, as already explained, when the user has to select a reference, there is a method called “select” and when the user inserts new data the invocation is to a method called “insert”. The test case and test data generator use this information in order to decide whether it is necessary to take data from the database or if it is necessary to design new test data.

In both cases, the generated test case will use a datapool that has different data partitions with the same structure as the related tables, as in the example in Figure 41.

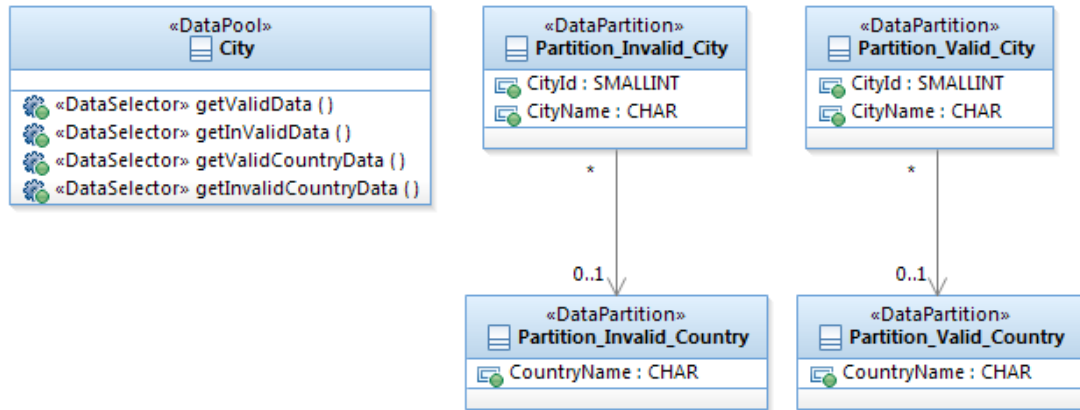


FIGURE 41 - STRUCTURED TEST DATA EXAMPLE

Then, when DBesTest generates code and specific test data, the transformation will take into account the structure of the data partitions and the kind of method that is being used by the test case, if it is a “select” or an “insert”, deciding whether the test data file is going to be filled with generated test data or with existing test data from the database.

5.2.3.3. INVALID DATA GENERATION

Possible errors that can be found in the AjaxSample application, and that are not considered for the coverage criteria, were summarized in Chapter 1, in the section *An Illustrative Example*. Sometimes these situations generate an exception, and in other cases do not throw exceptions, but persisting different values in the database than the one given as parameters, without notification for the user. Most of these situations are causes of error because the database designer and the developers (who coded the layers above the database) did not consider the same business rules and restrictions.

The examples presented are not completely covered by the different coverage criteria presented earlier. For example, some cases could be derived by applying equivalence partition and looking for invalid classes, but some of these partitions could only be guessed based on the database structure and restrictions. Using our approach, the equivalence partition method can be improved because of the extra information provided.

5.2.3.3.1. CRITERIA

Our goal is to generate test inputs and test sequences to identify faults in the system when managing the database, not only by testing with valid data and verifying the correct result, but also by testing with invalid data and verifying that the system can correctly manage the error situation.

Below, some criteria are defined to design invalid test data according to the information provided in the ISM in order to exercise all the violations of the database structure, to verify that the IS can manage it appropriately.

Primary Key Violation (PKV). If the primary key (PK) of a table is provided by the user in the GUI there should be a test case trying to violate the PK by trying to insert an element with a duplicated value.

Foreign Key Violation (FKV). If an input in the GUI corresponds to a value from another table through a foreign key (FK), there should be a test case trying to violate the FK by trying to reference an element that does not exist. Moreover, there should be a test case trying to delete a referenced element.

Unique Restriction Violation (URV). If there is a unique restriction (UR) defined in the DB for a field inserted by the user from the GUI there should be a test case trying to violate the UR by trying to insert an element with a duplicated value for the columns involved.

Not-Null Violation (NNV). For each input in the GUI stored in a column in the database with a not-null restriction (NN), there should be a test case trying to violate the NN restriction.

The test case *tc1* can violate the restriction by trying to insert an element with a null value for the column with the restriction.

Data Types Violation (DTV). For each input in the GUI stored in a column in the DB, there should be a test case trying to insert an invalid value according to the corresponding column's data type, i.e., a not numeric value in a numeric column, a bad formatted date in a date time column, an out of range number for the numeric type defined, a string longer than accepted, etc.

Table 18 shows an analysis of which criterion can be applied to each CRUD operation.

TABLE 18 - APPLICABILITY OF THE CRITERIA

Operation	PKV	FKV	URV	NNV	DTV
Create	X	X	X	X	X
Update	X	X	X	X	X
Delete		X			

For example, it is not possible to violate the PK restriction in terms of a delete operation, but it could be possible to violate a FK restriction with this operation, trying to delete a referenced register. Therefore, each coverage criterion applies to certain operations in the tables.

5.2.3.3.2. EXAMPLE

Let us consider the example of the creation of the entity *Invoice*, as already presented in previous examples (the GUI structure, navigation and data model). It is important to recall that for the creation of clients the test data generated corresponds to the different inputs of the different methods invoked in the sequence diagram indicating the interaction between the user and the SUT, and that the structure of the data partitions is the same as the structure of the corresponding elements in the database (see Table 17). Table 19 presents the design of the test data, i.e., the data partitions. For each attribute it shows the list of restrictions identified for those inputs by considering the corresponding database structure for them. It then also presents some examples of test data that can be generated for those parameters.

TABLE 19 - TEST DATA DESIGN EXAMPLE

Invoice Data Partition					
Attribute	Columns involved	Restriction type	Test action	Valid data	Invalid data
InvoiceDate	Invoice.InvoiceDate	DTV (datetime)	Confirm	12/03/2013	30/02/2000
numInvoiceLine		Relation N-N	Loop	1, 2, ...	0
		Business rule that says that each invoice should have at least 1 line			
Client Data Partition					
Attribute	Columns involved	Restriction type	Test action	Valid data	Invalid data
clientFirstName	Invoice.ClientId, Client.ClientFirstName	FKV, NNV	Select	Existing client	Incorrect client reference, null
InvoiceLine Data Partition					
Attribute	Columns involved	Restriction type	Test action	Valid data	Invalid data
ProductName	InvoiceLine.ProductId, Product.ProductName	FKV, NNV	Select_ProductName	Existing product	Incorrect product reference, null
InvoiceLineQty	InvoiceLine.InvoiceLineQty	DTV (smallint)	Insert_InvoiceLineQty	1, 2, 3	0, -1, AA
		Business rule that says that the amount must be greater than 0			

Note that not all the criteria are applicable. For example, the PKV is not applicable because the primary key is not an input but is autogenerated, thus, it cannot be tested from the user interface.

These kinds of test cases could not be designed without the information of the database, and managing this information with models allows a model-transformation designer to easily add new rules in order to produce more test cases automatically.

5.3. TEST CODE GENERATION

Finally, the test code is generated from the generated test cases (represented with UML-TP). Test models and ISM are transformed into test code with MOFM2T transformations, implemented with the Acceleo tool [8], which is compliant with the OMG specification. To transform this test case specification into executable code, this proposal continues the work presented by Pérez et al. [127], which uses UML-TP test cases automatically generated from UML sequence diagrams taken from the design of the SUT. The UML-TP test cases are transformed into JUnit or NUnit code using MOFM2T (specifically, MOFscript). In our case, as MOFscript is not supported any longer, and it presented some blocking difficulties, so DBesTest uses Acceleo.

Figure 42 shows the main inputs and outputs of the Acceleo transformations.

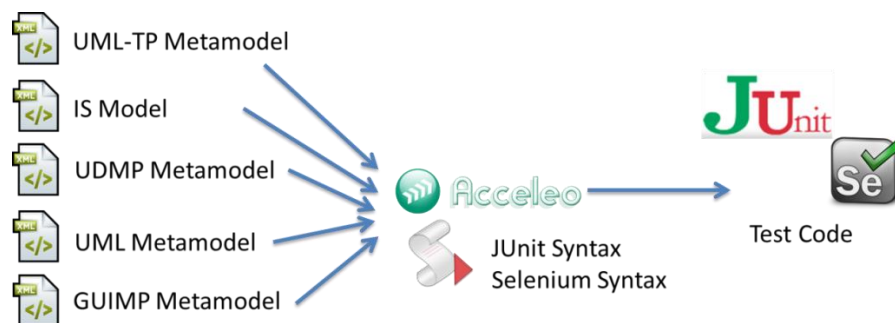


FIGURE 42 - INPUTS AND OUTPUTS OF THE ACCELEO TRANSFORMATIONS

Three actions are performed in this phase: generation of test data, generation of test cases (test behavior with the corresponding validations) and generation of the adaptations layer.

The last step is the transformation of the test model to test code, using model-to-text scripts. Three actions are performed in this phase: generation of test data, generation of test cases and generation of the adaptations layer. They are explained in the following subsections.

5.3.1.1. TEST BEHAVIOR

Test cases are generated in JUnit considering the test behaviors presented in the different sequence diagrams stereotyped as “Test Cases” (those automatically generated and, potentially, all the sequence diagrams added by the tester).

JUnit is probably the most popular automation framework specific to Java code, but it can be integrated with any kind of automation tool in order to execute tests on web systems, web services, mobile, etc., and JUnit will manage the executions and store the execution results. Figure 43 shows an example of generated code for the test case Create Country presented in the sequence diagram in Table 10.

```

@RunWith(value = Parameterized.class)
public class test_valid_country{

    private String countryName;
    public test_valid_country (String countryName) {
        this.countryName = countryName;
    }
    @Parameters
    public static Collection<Object[]> data() throws Exception {
        country_datapool dp = new country_datapool();
        return dp.getValidData();
    }
    I_home home;
    I_country_list country_list;
    I_country_create country_create;
    I_country_view country_view;

    @Before
    public void setUp() throws Exception {
        home = FactoryAdaptationLayer.get_home();
        country_list = FactoryAdaptationLayer.get_country_list();
        country_create = FactoryAdaptationLayer.get_country_create();
        country_view = FactoryAdaptationLayer.get_country_view();

        home.goHomeAndLogin();
        home.VerifyPage();
        home.goCountryMenu();
        country_list.VerifyPage();
    }
    @Test
    public void tc01_create_valid_country() throws Exception {
        country_list.New();
        country_create.VerifyPage();
        country_create.Confirm(countryName);
        country_list.VerifyPage();
        country_list.Open(countryName);
        country_view.VerifyPage(countryName);
    }
}

```

FIGURE 43 - GENERATED TEST CODE

There is a special object *home* to manage the access to the SUT (including *login* if it is necessary) and to access the different menus, as in this case for example, to access the *Country's* menu.

The generated test case ("*tc01_create_valid_country*") executes the steps indicated in the sequence diagram, taking the data from datapools (in this case, the country names). In the generated code, the datapools are transformed into classes which access CSV (comma separated values) files and give the data to the test cases using the "@parameterized" annotation of JUnit 4 (explained below, under parameterization).

Each message in the sequence diagram to the SUT is translated into a call to a set of classes considered as adaptation layers. The idea behind this is to follow a keyword-driven testing approach [103], so as to have the possibility of testing different components of the system that manage the same data model (i.e., a web component and a mobile phone component). Note that in the test case code there is not any reference to Selenium. The specific platform code is encapsulated in order to have a separation of the test steps from the specific execution. More detail about this is given below in the following subsection.

5.3.1.2. PLATFORM SPECIFIC EXECUTION

The different test sequences are in a certain way *abstract*; they cannot be executed on any platform, they only describe the intended test. In order to be able to execute these test sequences on a specific platform, it is necessary to add two things:

- Adaptation layer: the execution of the actions on each specific element of the GUI, working as wrappers of the SUT in order to control the access to it.
- Model-Implementation Mapping (MIM): the localization of every element of the GUI model on the current GUI of the SUT. This approach was presented by Xu [128].

The adaptation layer is generated with Acceleo scripts, and there are special scripts for each execution platform. For example, DBesTest includes specific scripts to generate Selenium code for web systems, but it could include another group of scripts to generate Robotium¹⁶ code for mobile applications. It generates code depending on the elements of the GUI that are used. For example, if the test has to insert data into an element tagged as “Input”, the generated code will be a “type” command, which is the corresponding Selenium command to insert data into an input of a web page.

The Model-Implementation Mapping is generated with the assistance of the tester, indicating how to locate each element of the model in the web page. This is done only once for each page, not for each test case, because the different test cases use the same pages. This is a well-known strategy in automation testing, called *page object pattern*¹⁷, reducing the duplication of test code, and making the test cases more maintainable and readable.

¹⁶ Robotium: <https://code.google.com/p/robotium/>

¹⁷ Page object pattern: <http://martinfowler.com/bliki/PageObject.html>

In this way, the user is adding platform specific information to the model, which until now was in the Platform Independent level.

The left side of Figure 44 shows an example of the MIM, where a *page* object is defined for one of the web pages of the ISM. In that way, the relationship between the elements in this model and the elements in the SUT is established, making the JUnit test cases executable on the web system. This information is read by the test cases and loaded to a hash map, accessed by keywords (see how it is used in Figure 46).

<<page>> Country_Create	
PAGE_NAME	/ajaxsample/country.aspx?INS,0
CountryName	id="COUNTRYNAME"
Confirm	name=BTN_ENTER
Cancel	name=BTN_CANCEL

FIGURE 44 - ADAPTATION LAYER EXAMPLE AND STRUCTURE

To generalize the management of this schema, DBesTest generates a hierarchy with a *Factory*, as seen on the right hand side of Figure 45, in order to make it possible to have different implementations of the adaptation layer in the same project in a simple way. This allows the implementation of different adaptation layers, and different page objects for different technologies.

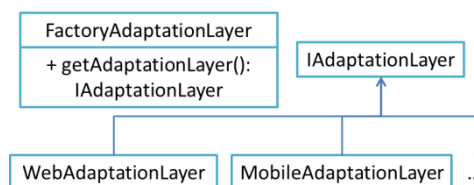


FIGURE 45 - FACTORY FOR ADAPTATION LAYERS

In the example, *FactoryAdaptationLayer* returns (in the corresponding *getter*) different interfaces implemented by specific classes using Selenium code:

- The interface *I_country_list* implemented by the class *country_list_selenium*
- The interface *I_country_create* implemented by the class *country_create_selenium*
- The interface *I_country_view* implemented by the class *country_view_selenium*

Figure 46 shows some generated code for the wrapper to access the page *Country_Create*. The method *Confirm* of the wrapper *country_create_selenium* executes the Selenium commands necessary to insert the parameters into the web page (in this case, fill the input *CountryName*) and execute the action *confirm* (click the button). The code for the method was generated considering the page elements in the user interface

model, executing a “type” Selenium command on each input, and a click on the submit button. The HTML identifiers are obtained from the corresponding MIM.

```

public class country_create_selenium extends SeleneseTestBase implements I_country_create{

    HashMap<String, String> mim = MIM.getHashMap();

    public void confirm(String CountryName) throws Exception {
        selenium.type(mim.get("Country_create.CountryName"), CountryName);

        //click on Confirm
        selenium.click("Country_create.Confirm");
        selenium.waitForPageToLoad("30000");
    }

    public void verifyPage() throws Exception{
        verifyEquals(selenium.getLocation(), mim.get("homePage.URL") +
mim.get("Country_create.PAGE_NAME"));
    }
}

```

FIGURE 46 - ADAPTATION LAYER USING SELENIUM

In this way, DBeSTest generates a set of test cases that can be executed against SUT, according to the test specification provided in the UML-TP model.

5.3.1.3. PARAMETERIZATION

DBeSTest generates test cases in JUnit. The most recent version of JUnit (the version number 4) has introduced the concept of *Parameterized tests*. Parameterized tests allow the developer to run the same test over and over again using different values. In that way, it gives support to data-driven testing.

There are five steps that must be followed to create parameterized tests:

- Annotate the test class with `@RunWith(Parameterized.class)`
- Add a public static method and annotate it with `@Parameters`, returning a Collection of Objects as test data set.
- Create a public constructor of the test class that takes in what is equivalent to one "row" of test data.
- Create an instance variable for each "column" of test data.
- Use the instance variables as the source of the test data.
- When running with the JUnit engine, the test case will be invoked once per each row of data.

In our case, paying attention to the test code presented in Figure 43, the method annotated with “`@Parameters`” corresponds to the Data Selector, and it will access the CSV files with corresponding test data. In this way, the test case is easier to read, and

the test data remains separate from the test behavior allowing the tester to add new data to test other situations simply by adding rows to the CVS file.

5.3.1.4. TEST DATA GENERATION

Test data is generated from the data model structure and the business rules, and considering the relationship between inputs and columns. This proposal deals with input data but not with the preparation of the states of the database as in some other proposals ([113], [115], [117], [129]), considering that it is possible to use the different test cases to generate different initial states. For instance, to create a city it is necessary to have information about countries; therefore, the test sequence first executes the creation of a country and then the creation of the city with this generated country.

DBesTest considers the Data Partitions in the test model to generate data according to the structure and data types. The result is a set of CSV files which will be read by the test cases using special classes that were generated from the Datapools, though the Data Selector methods.

As was explained in section 5.2.3, DBesTest generates valid and invalid data partitions. For the valid data partitions random data is generated according to the data types. In order to generate valid data considering all the business rules, *Constraint Solvers* should also be used, but this was out of the scope of this work, and many proposals already presented research results in this direction ([42], [130], [131]). On the other hand, in order to generate invalid data, there is a special consideration for each data type and for each business rule, as was explained in subsection 5.2.3.3.

Apart from the valid/invalid classification, the test data generator applies the boundary values technique [4] and combinatorial testing.

As an example of boundary values, considering the integer attribute *InvoiceLineQty* (amount of a selected product in an invoice line), and the business rule indicating that the value must be greater than zero, then a set of interesting values could be: {-100, -1, 0, 1, 100}.

Once the valid and invalid values are generated for each input field, it is necessary to combine them to generate the different data rows for the CSV files. For this, it is possible to combine the values with pair-wise algorithms, using our own tool called CTWeb (ctweb.abstracta.com.uy). In this way it is possible to obtain a reduced set of tuples with a higher probability of finding errors in comparison to the proposal suggested by Andrews et al. for the *CA criteria* [111], who proposed to use the Cartesian product of the different interesting values.

This combination of values should not use two invalid classes simultaneously. This is done not to hide errors, because if two invalid values are used at the same time, two error notifications are expected, and perhaps one error will not be flagged because of the other.

Figure 47 shows the correlation between the elements in the test model and the generated test data (considering the example presented in Table 19 for the test data to test the creation of invoices). The equivalence classes which correspond to a data partition, and the corresponding generated test data are also shown, which were combined with a pair-wise algorithm and stored in different CSV files.

Note that the invalid data is not combined, as previously explained.

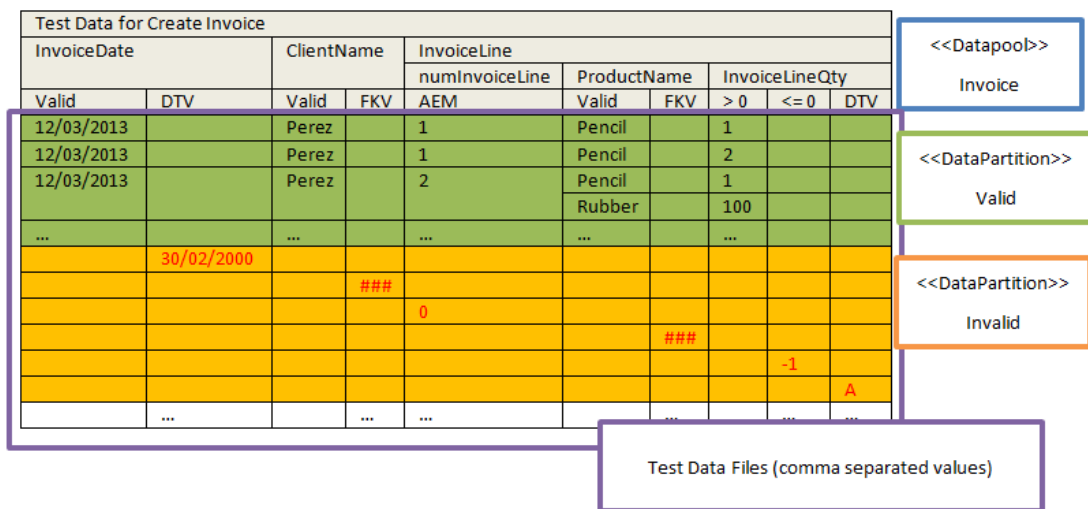
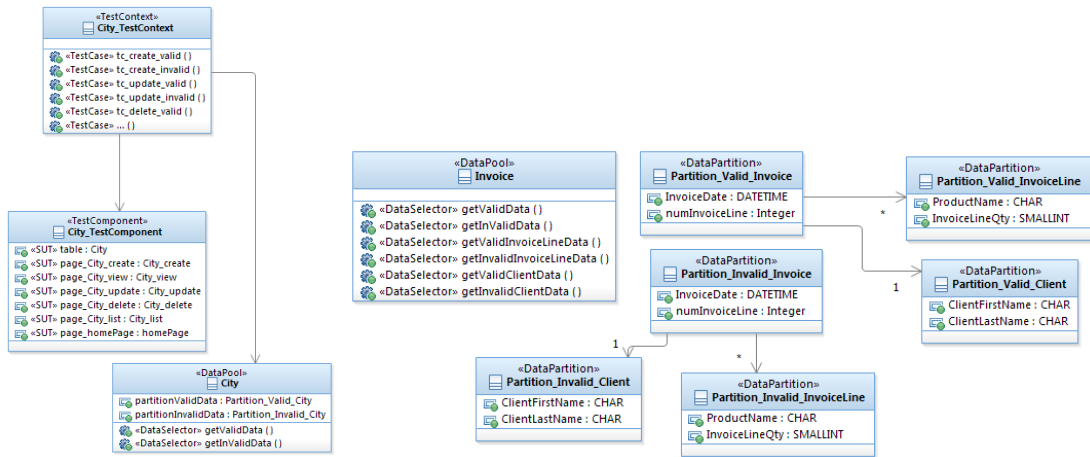


FIGURE 47 - CORRELATION BETWEEN DATA IN THE TEST MODEL AND GENERATED TEST DATA

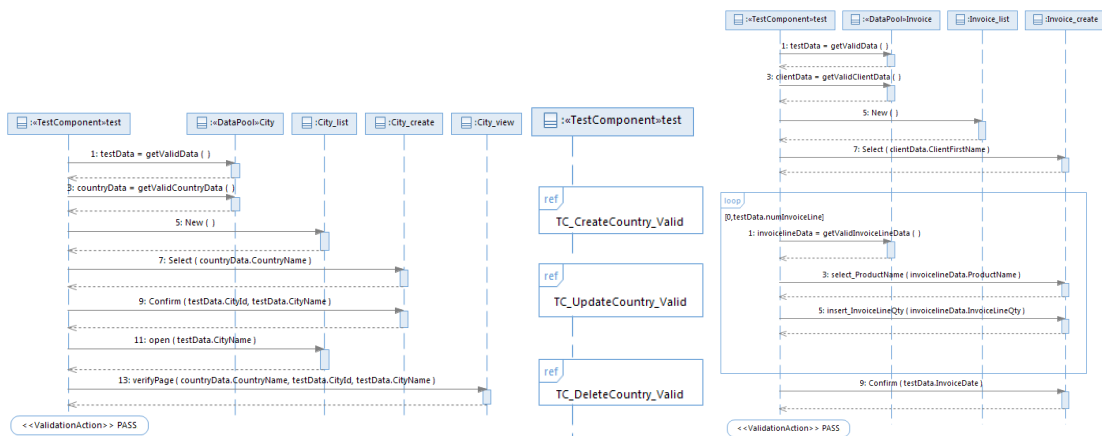
5.4. CONCLUSION

By considering the example presented at the end of Chapter 4, Figure 48 presents a summary of the results obtained after applying the methodology explained in this chapter.

TEST ARCHITECTURE AND TEST DATA MODEL



TEST CASE BEHAVIOR



TEST CODE, ADAPTATION LAYER, MIM AND TEST DATA

```

package tests;
import java.util.Collection;
@UmWith(value = Parameterized.class)
public class City_test {
    private String CityId;
    private String CountryId;
    private String CityName;
    //constructor
    public City_test(String CityId, String CountryId, String CityName) {
        this.CityId = CityId;
        this.CountryId = CountryId;
        this.CityName = CityName;
    }
    @Parameters
    public static Collection<Object[]> data() throws Exception {
        City dp = new City();
        return dp.getData();
    }
}
    
```

```

package adaptationLayer.City;
import com.thoughtworks.selenium.*;
public class City_create_selenium extends SeleniumTestBase implements I_City_create {
    HashMap<String, String> mIn = HDI.getHashMap();
    public void clickConfirm(Create(String CountryId, String CityId, String CityName) throws Exception {
        selenium.type(mIn.get("city_create.CountryId"), CountryId);
        selenium.type(mIn.get("city_create.CityId"), CityId);
        selenium.type(mIn.get("city_create.CityName"), CityName);
    }
    public void verifyPage() throws Exception {
        verifyEquals(selenium.getCollection(), mIn.get("homePage.url") + mIn.get("city_create.PAGE_NAME"));
    }
}
    
```

Test Data for Create Invoice						
InvoiceDate	ClientName	InvoiceLine	Product Name	InvoiceLineQty	Valid	DTV
12/03/2013	Peraz	1	Parocil	1	> 0	<= 0
12/03/2013	Peraz	1	Parocil	2	> 0	<= 0
12/03/2013	Peraz	2	Parocil	1	> 0	<= 0
12/03/2013	Peraz	2	Rubber	100	> 0	<= 0
12/03/2003	Peraz	0	Parocil	1	> 0	<= 0

FIGURE 48 – SUMMARY OF THE CHAPTER

“Only performance testing at the conclusion of system or functional testing is like ordering a diagnostic blood test after the patient is dead.”

Scott Barber

CHAPTER 6. AUTOMATIC GENERATION OF PERFORMANCE TESTS

This chapter introduces the consideration of non-functional aspects in the framework, specifically for workload simulation (known as load or performance testing). On the one hand, our proposal is to generate performance test cases from the functional ones; on the other hand, a non-functional requirements model is used to generate a complete test scenario. This is an integrated approach, considering functional and non-functional verification in a single model.

6.1. INTRODUCTION

In the traditional view, non-functional testing is performed after completing the functional test with the aim of covering non-functional aspects such as performance, dependability and security. In both traditional software development and in model-driven development, the construction of functional and non-functional test cases is carried out in two separated steps, with almost no relationship.

From our experience in dozens of projects, providing functional and performance testing services, the same situation has been observed many times: in the first stage the testers prepare the functional test specification, automate those test cases, and execute them; later, they prepare the non-functional test specification, automate those test cases, and execute them. Two different specifications, two different groups of test scripts, and most of the times, the non-functional test cases are a subset of the functional test cases, perhaps with some specific elements to measure performance, such as timers. Moreover, it is very common to find functional issues when looking for non-functional ones, or vice versa. Last but not least, many customers claim not to have the time (or budget) to execute both test sets, therefore, they opt to leave the performance test for another stage, assuming all the associated risks. The question is: why not to consider an integrated approach, designing, modeling and executing the functional and non-functional tests together?

Our first step towards the integrated execution and analysis is presented in this Chapter, showing how it is possible to derive an integrated test model from two requirement models: one for the functional specification of the system, and another for the non-functional properties. To simulate a workload three things are required:

- The test cases automated for a specific workload simulation tool.
- The workload specification to be simulated, which basically indicates how the test cases are combined to represent the expected load of the system.
- The oracle information, how the verdict of an execution can be determined.

This proposal unifies different research lines in order to solve these issues (see Figure 49, which shows the general approach already presented in previous chapters):

- Representation of functional and non-functional test model with UML-TP. This model is generated from the ISM (UML) for the functional specification and a PMM¹⁸ model for the non-functional specification (explained below in section 6.2).
- Generation of automated test cases for workload simulation platforms. This is done adapting the functional automated test cases to the workload tools (explained below in section 6.3).

The acceptance criteria are also different; in performance testing, the test criteria are defined based on the average of all the response times, or considering a percentage of pass/fail results.

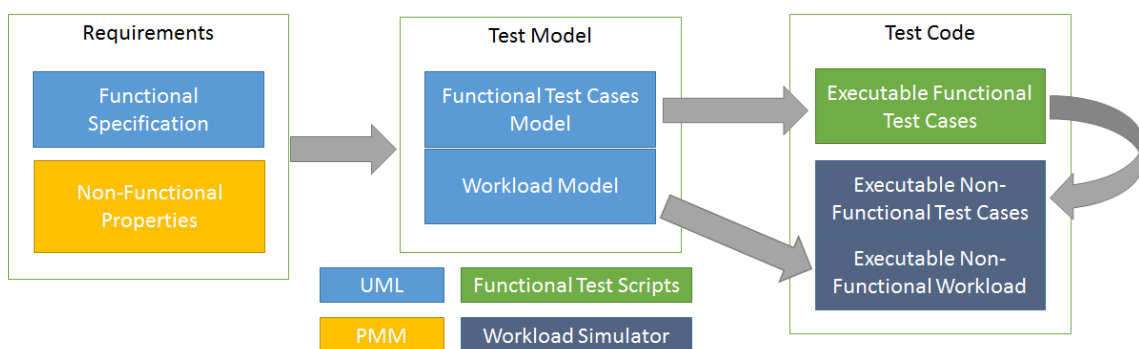


FIGURE 49 - SCHEMA FOR INTEGRATED TEST GENERATION

¹⁸ This research topic was developed during a research stage in Pisa, Italy, in the *Consiglio Nazionale delle Ricerche* (CNR), where PMM was developed. Our proposal uses this metamodel, but it may be possible to change this.

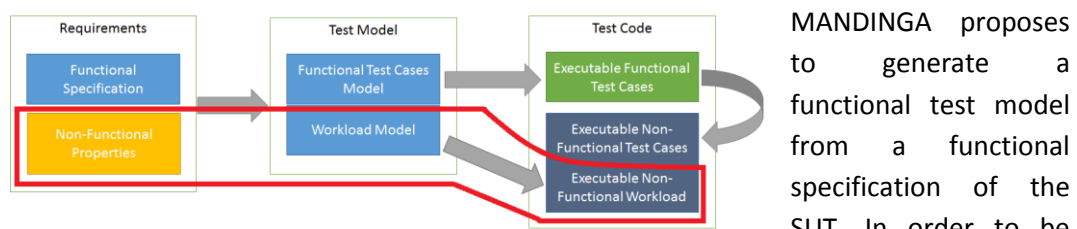
This chapter follows the example applied to AjaxSample in order to present both parts of our proposal. In this scenario after a market analysis and sales forecast, it has been defined that, in a peak hour, it is possible to continuously have approximately 50 users registering products (*Create Product* operation), while 100 users are buying products where the buying process includes registration on the site (*Create Client* operation), adding items to their invoice, confirming the purchase and payment (*Create Invoice* operation).

It is desirable to ensure that when the system is under this load situation, it is able to retain good response times and low error rates. To do so, the following non-functional requirements were defined:

- The average of the duration of the *Create Product* operation should be less than 15 seconds (only considering server time being defined as the time the system takes to answer, excluding user time).
- At least the 90% of the total amount of the buying process should be correctly processed by the system.

In order to prepare the test for these operations, it is therefore necessary to cover the non-functional properties, and generate the executable components. Section 6.2 shows the model-driven approach to generating the workload scenario, combining the executable test cases, and adding non-functional validations, covering the non-functional properties, and Section 6.3 shows how the workload simulation scripts (executable test cases) are generated from the functional test cases already generated.

6.2. MODEL-BASED TEST CASES DESIGN INTEGRATING FUNCTIONAL AND NON-FUNCTIONAL ASPECTS



able to extend the test model with non-functional considerations, it is also necessary to consider non-functional requirements. Towards this aim, MANDINGA uses PMM, which lets the user specify non-functional properties of the system, particularly performance and dependability, under certain workload conditions. With this information, our framework generates specific test components for the verification of those properties, covering the non-functional specification.

As evidenced in Chapter 2, UML-TP has some shortfalls related to the inclusion of non-functional properties.

This section presents some contributions made into PMM in order to improve its expressiveness for concurrent operations of web systems, as well as the contributions to UML-TP in order to improve its expressiveness for non-functional test modeling. It then also presents the non-functional test model generation, covering the specification provided by the user in the PMM model.

6.2.1. CONTRIBUTIONS TO PMM

PMM, presented in Chapter 2, is a generic metamodel for defining non-functional properties. In this thesis, PMM is used for defining performance and dependability property models that will be the input for the proposed test model generation approach. Specifically, the elements of the PMM models, such as the operators of the *MetricsTemplate*, the simple and complex events, the workload and the properties constraints, will be used to automatically generate a non-functional test specification.

Moreover, to better specify the workload associated with performance and dependability properties, an extension is proposed for PMM, introducing the ***WorkloadModel*** element which can include more than one workload and can be associated with a set of properties (see the frame of the Figure 50 with continuous lines). This extension allows the inclusion of different workloads for the same set of properties.

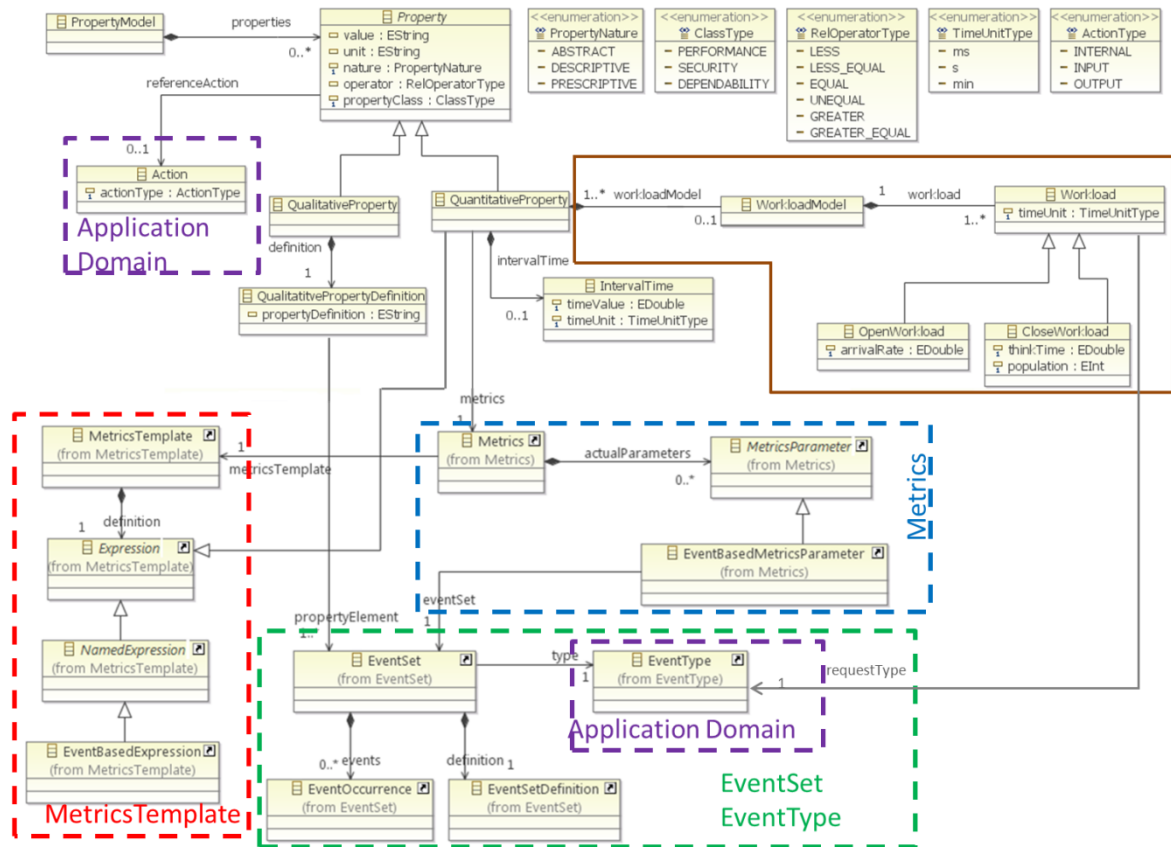


FIGURE 50 - PMM METAMODEL

Figure 51 shows an example of a PMM model of performance and dependability properties. The subsections following will show how these models are used for deriving the associated test model.

To represent the non-functional requirements of the operations of the SUT it is necessary to map them into PMM *event type* models. In the example, a simple *event type* was defined, called *CreateProduct* corresponding to the *Create Product* operation (see Figure 51 (a)), and a more complex *event type* named *BuyingProducts* showed in Figure 51 (b).

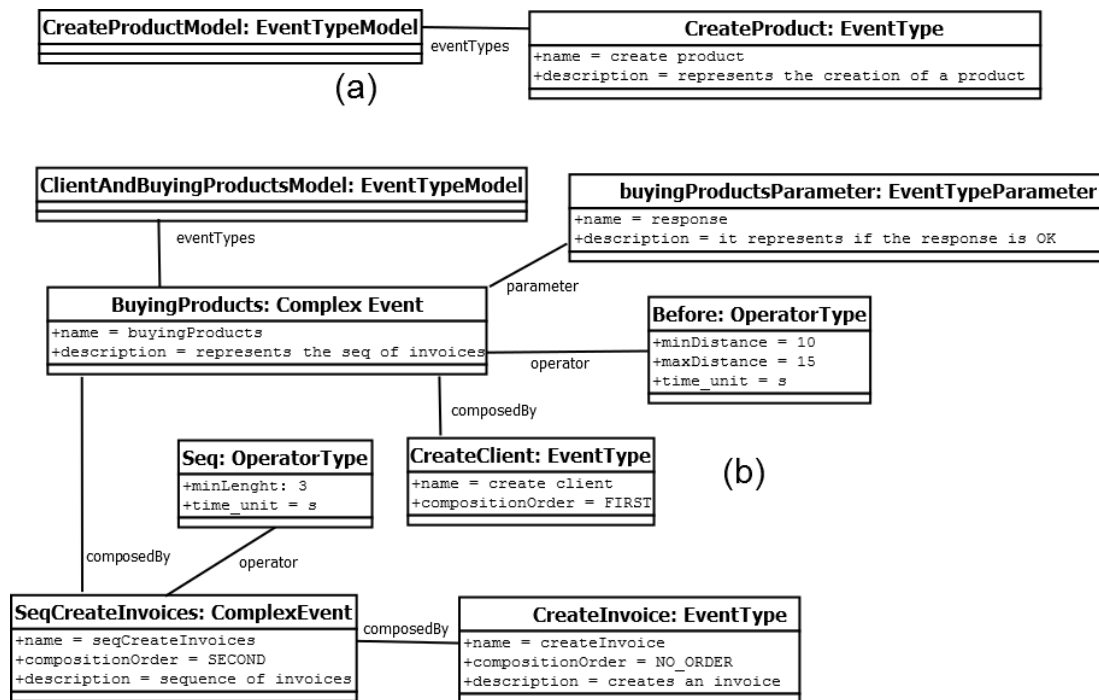


FIGURE 51 - PMM EXAMPLE

The *BuyingProducts* complex event model defines a *CreateClient* operation followed by a sequence of *CreateInvoice* operations. Specifically, it has a *Before* operator, with *minDistance* and *maxDistance* parameters equal to 10 and 15 seconds respectively, and it is applied to the simple event *CreateClient* (corresponding to the *Create Client* operation) and to another complex event that is named *SeqCreateInvoice* with *Seq* operator applied to the *CreateInvoice* simple event, representing a sequence (with minimum length of three) of *Create Invoice* operations. Finally, the *BuyingProducts* event has a parameter named *response* which has the *OK* value if the *Create Client* operation, followed by a sequence of *Create Invoice* operations, is successful.

The PMM non-functional properties involving the defined events models were then defined. Specifically, the *PropertyModel*, shown in Figure 52, is comprised of two properties. The former is a quantitative latency property with a performance class. It is a prescriptive property since it specifies a required average time response less than or equal to 15 seconds for the *Create Product* operation.

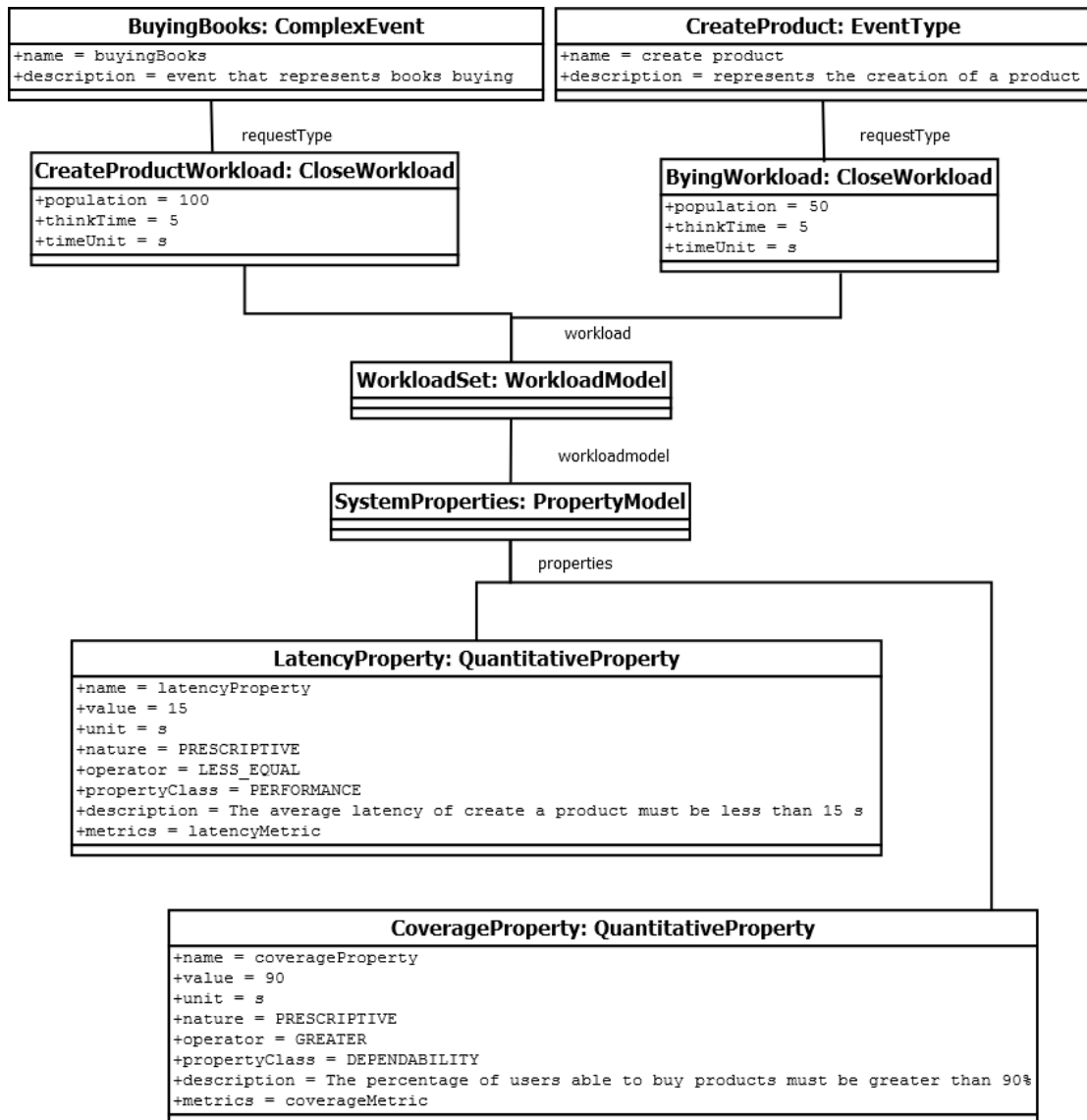


FIGURE 52 - MODEL OF THE COVERAGE AND LATENCY PROPERTIES

To this property is associated a metric (*latencyMetric*) linking the set of *CreateProduct* events involved in the property to a metric formula (*metrictemplate*) which specifies how to compute the performance measure specified in the property. This template applies the average operator to the duration of the *CreateProduct* events.

The latter is a quantitative dependability property (coverage property). It is also a prescriptive property, specifying that at least the 90% of the total number of *BuyingProducts* operations must be successful in an *IntervalTime* of 60 seconds.

To this property is associated a metric (*coverageMetric*) that links two sets of events (the total set of *BuyingProducts* events and the set of *BuyingProducts* events having the response value equal to *OK*) to a metric template which defines the mathematical expression for computing the dependability measure specified in the property, as the division of the cardinalities of the two sets.

Finally, this *PropertyModel* has an associated *WorkloadModel* that includes two workloads (*CreateProductWorkload* and *BuyingWorkload*) each of them specifying the values for *population*, *thinkTime* and *timeUnit*. This can be modeled as such only thanks to our extension, otherwise it could not be possible to associate more than one operation with a workload, which is the most common situation modeling the workload of a web system for example.

6.2.2. CONTRIBUTIONS TO UML-TP

In order to integrate non-functional aspects into MANDINGA, it is necessary to enrich the functional specification (metamodels and methodology presented in Chapter 4 and 5) with non-functional properties.

A systematic survey of MBT approaches for non-functional requirements evidenced that a standard and common language for designing non-functional test cases is lacking. The only contribution in this direction is the UML-TP. However, UML-TP provides only limited support for non-functional testing and it does not allow some important concepts of performance testing to be specified, such as the workload and the definition of a global verdict for concurrent executions. This section explains the aims to overcome this limitation, specifically, two aspects: the workload specification and the verdict definition involved in non-functional validations. The proposed extension allows for modeling a wider variety of test cases with different test goals in one single UML model¹⁹.

6.2.2.1. WORKLOAD INFORMATION

An important shortfall of the UML-TP standard is the representation of the workload, which is one of the most important aspects of the performance testing activity. Designing a performance workload model very similar to the SUT environment is one of the core activities in performance testing. The test case design should define the workload by including a list of parameters (load distribution, number of concurrent

¹⁹ The need for extension of the UML-TP to design non-functional tests and the ideas proposed in this section have undergone detailed and useful discussions with some members of the UML-TP development team by private electronic correspondence.

users, etc.) that are necessary for an accurate simulation in order to reach the test objective. In order to model all the parameters of performance testing, our proposal is to introduce the concept of WORKLOAD into UML-TP as a new stereotype with some tagged values. The stereotype would be applicable to Test Cases (to operations of a Test Context that already have a Test Case stereotype applied). With this new stereotype it is possible, for instance, to specify test suites (or Test Context) with an associated workload that indicates the number of concurrent users.

Conceptually, the “workload” stereotype applies to the relationship between the Test Context and the Test Case (see Figure 53).

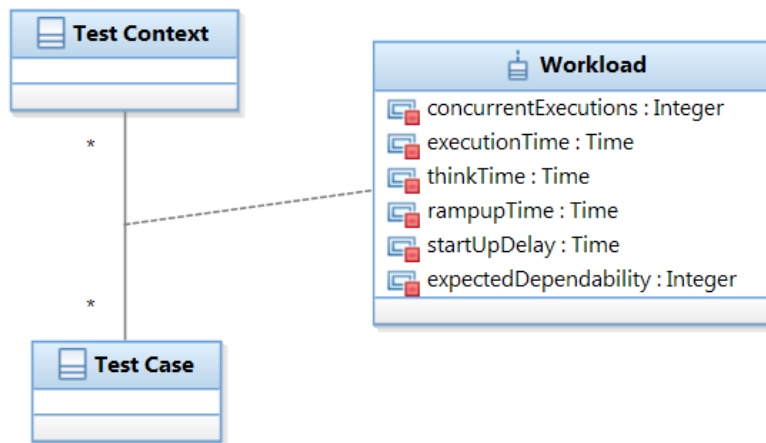


FIGURE 53 - WORKLOAD STEREOTYPE

The different tagged values (stereotype properties) proposed for modeling the workload are:

- *concurrentExecutions*: also known as virtual users, refers to the number of concurrent executions of the test case that are going to be considered in the load simulation;
- *executionTime*: represents the total execution time, that is how long the test case is going to be executed;
- *thinkTime*: represents the delay between iterations, also known as think times, which determines the pause between each execution of the test case for each virtual user;
- *rampupTime*: determines the initial time required to reach the expected load, when the virtual users are entering to the system progressively;
- *startUpDelay*: defines the time when the virtual users executing the test case are going to be started, relative to the beginning of the whole test suite;

- *expectedDependability*: represents the expected percentage of executions that should pass in order to consider the test case as passed. More details about this property in the next subsection.

The Test Scheduler is in charge of taking into account (in its default behavior) the above parameters specified in the workload.

6.2.2.2. NON-FUNCTIONAL VALIDATIONS

With the time restrictions provided by the UML-TP standard it is only possible to define a local verdict, namely if the test case takes more time than the defined restrictions, it is considered a failure. On the other hand, the default behavior of the arbiter has an *only-gets-worse* policy, which means that once it receives a fail result it can never improve on that.

Two typical situations in performance and dependability testing that are not covered by the above time restrictions and verdict definition are: i) it is necessary to give the verdict according to the average of the execution times; and ii) it is necessary to verify that a certain number of the executions pass, thus taking into account a policy different from the *only-gets-worse* policy.

In the current UML-TP standard version it is only possible to model the maximum and minimum acceptable values. It is not possible to report a failure according to the average, or a percentage of the response times of all executions of the test case in the test context. To address this issue and improve the UML-TP time restrictions, the addition of new constructors, *average* and *percentage*, is proposed. The former gives a verdict considering the average of the response times, and the latter gives a verdict according to the percentage of the response times under the expected value.

Figure 54 shows an example of the new UML-TP *Time Restrictions*. In this figure “average” has a range as a parameter (0..10), and “percentage” has an extra parameter (95) indicating which percentage of the times should be in the range (it is necessary in this case to validate that at least the 95% of the response times are in the range 0..10).

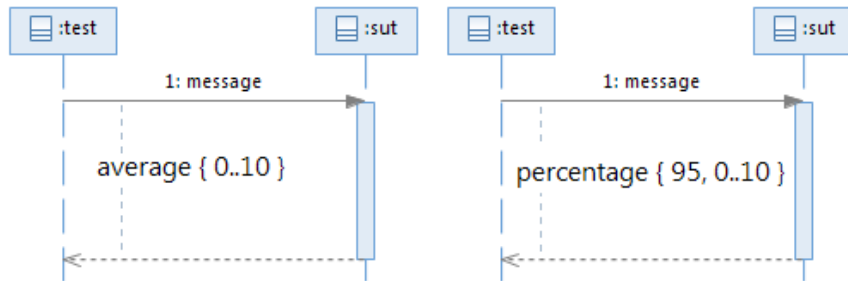


FIGURE 54 - EXTENDED TIME RESTRICTIONS

On the other hand, it is desirable to provide the tester with the ability to model performance test criterion (for example, a test case passes when a percentage of the responses give a pass verdict). For this, two changes to the standard profile were necessary. An extra property was added to the “*Workload*” stereotype, indicating the expected percentage of executions that should pass in order to consider the test case as passed. This attribute was called “*expected dependability*” (namely the failure tolerance). Secondly, it is necessary to change the default behavior of the arbiter. The test execution has only one arbiter associated with the Test Context, which has the only-get-worse policy. Adding one “partial” arbiter to each test case was proposed, in order to give a verdict for each test case considering all its executions. Those arbiters should not have *only-gets-worse* behavior; instead, they should give a verdict considering the level of tolerance for the corresponding test case. The main arbiter associated with the Test Context is then notified of each verdict and provides the final verdict as pass only if all partial arbiters returned a pass. It is important to note that the UML-TP user would only need to model the “*expected dependability*” value, and let the default arbiter be in charge of giving the final verdict.

This representation is enough to model the most common performance test scenarios and their typical validations. If it is necessary to calculate the verdict another way, it is always possible to describe this behavior with another UML diagram. It is evident that the proposed extension allows representation in a simpler way, and with more information. Including these extensions in the standard UML-TP increases the expressiveness of the metamodel, consequently the resulting models are easier to develop and understand.

In the next subsection a UML-TP model is generated, using this extension, and some examples are presented.

6.2.3. WORKLOAD GENERATION AND PMM OPERATORS COVERAGE

This section presents our proposal for the generation of an integrated test model starting from functional and non-functional requirement specifications (the ISM and the PMM models).

6.2.3.1. TEST MODEL GENERATION ALGORITHM

Taking into account the requirements specified in both input models, there is a model transformation script to automatically generate a test model addressing both aspects together, verifying functional behavior and non-functional properties. This model will be used to automatically generate a set of executable test artifacts and give a verdict about the functional and non-functional behavior of the system.

As a part of our proposal, a prototype was developed with ATL [19]. It receives two input models: the ISM (a UML model with the functional specification of the system) and a PMM model (with the non-functional properties). The output of this transformation is an integrated test model represented with the extended UML-TP. The transformation also needs the related metamodels as input, i.e. the UML metamodel, PMM metamodels and the UML-TP metamodel.

Using DBesTest, the functional test model includes the following components:

- One or more test cases for each functionality;
- A Test Context, containing the generated test cases as methods;
- Datapools;
- Test Components to interact with the SUT;
- Behavior Diagrams showing the different steps of the execution of each test method.

The test model is then enriched based on the information provided in the PMM models.

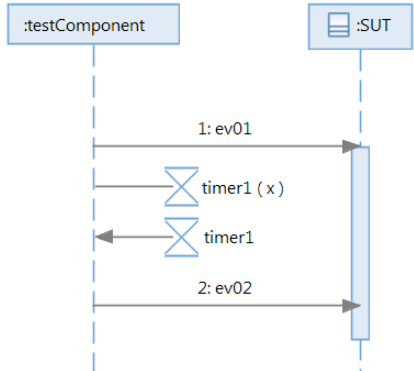
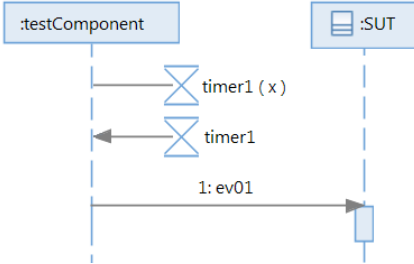
First, the workload information defined in the PMM model is considered in order to enrich the already generated Test Context. Specifically, the event specified in each workload is mapped into a test case with a *workload* stereotype in the Test Context. The information from the PMM workload model (*population*, *thinkTime*, *timeUnit*) is mapped into the *concurrentExecutions* and *thinkTime* parameters, of the *workload* of the test model, indicating the number of concurrent users and the wait time between executions respectively. These two parameters are the most important for defining a test load. The remaining parameters of the *Workload* stereotypes are defined by the tester according to their testing experience.

Our approach assumes that the events specified in PMM are expressions (manifestations) of the operations (functionalities) of the system. These events can be

simple or complex and the complex events are represented as a composition of other simple or complex events by means of operators. The functional test cases generated in our approach cover simple operations or functionalities. The functional test cases are then combined according to the operators specified in the PMM complex events, in order to generate a test model able to cover more complex functionalities expressed by complex events.

Table 20 presents an overview of the main complex event operators of PMM and the corresponding test cases generated, specified with UML-TP. For example, if there is a property with the “Before” operator relating “ev01” and “ev02”, meaning that the property should be checked every time that the ev01 happens before the ev02, then the idea is to force this situation in order to be able to test whether the property is respected by the SUT. Thus, the generated test case executes ev01 and then ev02. In this case, the operator has two parameters: *minDistance* and *maxDistance*, indicating the minimum and maximum time distance between the occurrences of both events. The generated test case will therefore execute a pause between both invocations, in order to respect this restriction. The rest of the examples presented in Table 20 follow the same idea: generating a test case to force the situation indicated by the operand, in order to be able to verify the associated property.

TABLE 20 - GENERATED TEST CASES FOR THE MOST COMMON PMM OPERATORS

Event Operator	Generated Test Case
<ul style="list-style-type: none"> • After: involves two events indicating that the first event (ev02) occurs after the second event (ev01). • Before: involves two events indicating that the first event (ev01) occurs before the second event (ev02). • <i>minDistance</i> (x) and <i>maxDistance</i> (y) represent the min and max time distance between the first event finishing and the second event starting. 	
<ul style="list-style-type: none"> • AfterT: involves one event and occurs when the event (ev01) happens after the specified time period (x). 	

<ul style="list-style-type: none"> • BeforeT: involves one event and occurs when the event (ev01) happens before the specified time period (x). 	
<ul style="list-style-type: none"> • Concurrent: involves two concurrent events and occurs when both events happen irrespective of their order. 	
<ul style="list-style-type: none"> • FollowOut: involves three events and occurs when the first event (ev01) is followed by the second event (ev02) and without the occurrence of the third event (ev03) 	

<ul style="list-style-type: none"> • Meets: occurs when the first event (ev01) finishes when the second event (ev02) starts. • MetBy: occurs when the current event (ev02) starts when the correlated event (ev01) finishes. • <i>maxDistance</i> (x) represents the max time distance between the first event finishing and the second event starting 	
<ul style="list-style-type: none"> • Or: involves two events and occurs when one of the two events happens. 	
<ul style="list-style-type: none"> • Not: indicates that the specified event (ev01) doesn't happen. 	
<ul style="list-style-type: none"> • Seq: involves one event (ev01) and occurs when there is a sequence of occurrences. • minLenght (x) represents the min length of the sequence 	

The PMM model also specifies a set of properties associated with a workload model. Each property refers to a metric template modeling how to compute the performance measure specified in the property. Every metric is related to an event set, which refers to a certain operation of the user on the SUT (let us say that these are the methods under test).

The metrics are defined on the same kind of events as the workload, because generally it is desirable to measure the latency or dependability of the actions that are part of the workload.

Also, and independently of the coverage based on the operators, there are some elements inserted in the test cases in order to provide a verdict. If the metric is related to time duration, it is necessary to add a time restriction for the referenced event. Generally, the time restrictions are defined on average or over a percentage of the executions, and thanks to the proposed UML-TP extension, it is possible to add these kinds of time restrictions to the test model.

The main steps of the test model generation strategy are summarized in the following:

1. Generate functional test cases from the functional specification. In this step, a set of *test cases* are generated for each method being tested.
2. For each *workload* of the *workloadModel* (in the *Test Context*) a *Test Case* is generated with a *workload* stereotype (specifying the workload information). Those *Test Cases* have a sequence diagram specifying their behavior.
3. If the *event* associated with the *workload* is a *simple event*, then the sequence diagram of the Test Case invokes (within an *alternative* combined fragment) all the functional test cases generated in the first step for the functionality corresponding to this event. If the *event* associated with the workload is a *complex event* then the test case is generated taking into consideration the coverage criterion explained before and presented in Table 20.
4. If a *performance property* is defined for this event then the generated test case includes a *time constraint* according to the property specified (considering the operators taken from the *metricsTemplate*).
5. If a *dependability property* is defined for this *event*, then the tagged value named "expected availability" of the stereotype "workload" is set as equal to the value specified in the property.

Figure 55 represents this algorithm for the generation of test cases (it is a graphical and simple representation in order to explain the algorithm). Three functionalities are represented (A, C and D) as the basis for the example.

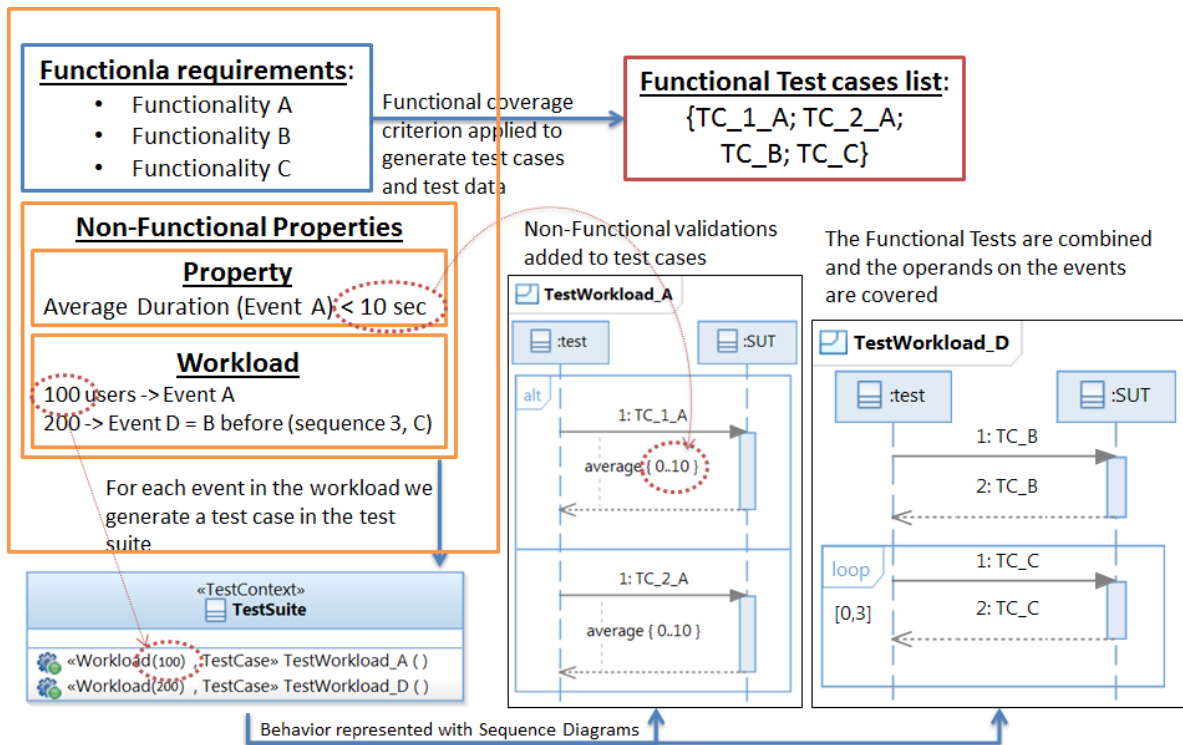


FIGURE 55 - TEST CASES GENERATION PROCEDURE

The algorithm has as input a PMM model specifying that the time response of *event A* corresponding to *functionality A* must be less than 10 seconds. This requirement is associated with a *workload model* composed of two *workloads*, one for simple *event A* with 100 users and one for complex *event B* with 200 users. The complex event B is specified as *C before a sequence of D*.

Applying the first step a functional test set is generated, which is composed of two test cases for *functionality A*, specifically *TC1_A* and *TC2_A* (one with valid data and one with invalid data). For functionalities *C* and *D* there are two test cases, one each (*TC_C*, *TC_D*). These test cases are grouped in a *Test Context* called “Test Suite”.

Applying the second step the *Test Context* is enriched with two *test cases* specifying the two workloads (*TW_A*, *TW_B*). The stereotype “workload” of the extended UML-TP is applied to specify the number of concurrent users for each workload (100 for *TW_A* and 200 for *TW_B*).

Applying the third step the behavior of *TW_A* and *TW_B* is specified. As two test cases were generated for *Functionality A*, then the *TW_A* invokes both alternatively. As the *event B* is complex, *TW_B* was generated, applying the coverage to the operators (specifically, for the *before* and *sequence* operators).

Finally, applying step 4, and considering that there is a performance property in the PMM model, a time restriction was added to *TW_A*.

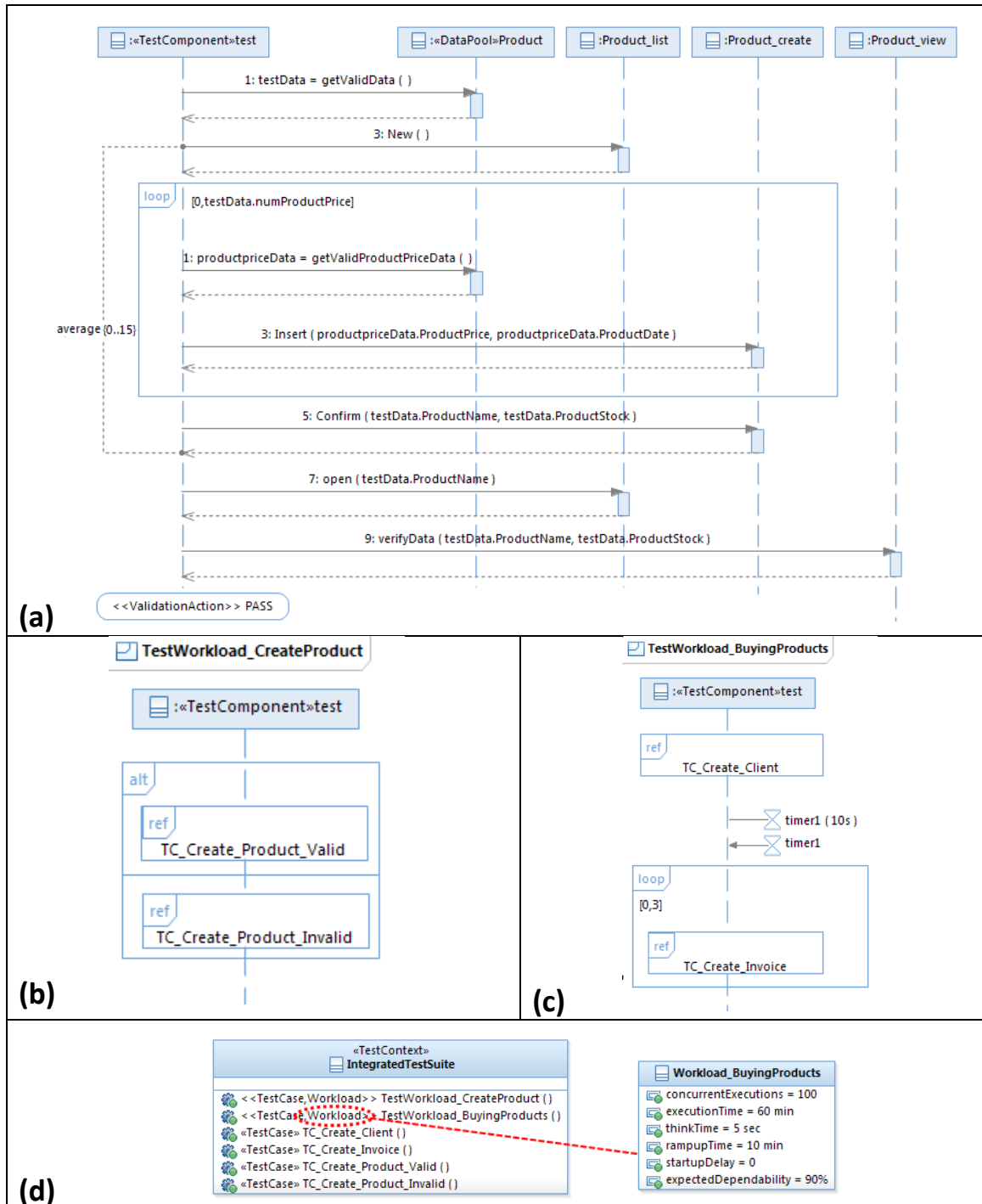
Thanks to this strategy, it is possible to generate the appropriated situations to verify the non-functional properties while executing the functional test cases.

6.2.3.2. EXAMPLE

Table 21 presents part of the resulting UML-TP model generated for the PMM model previously presented as an example. First of all, paying attention to Figure (d) of Table 21, the Test Context will include information about the workload, and thus to everything related to the concurrent user execution. For this, there are new Test Cases which also include the application of the stereotype “Workload” (i.e. *TestWorkload_CreateProduct* and *TestWorkload_BuyingProducts*) where there are different tagged values storing the important information taken from the workload defined in PMM. For example, it is also possible to see the representation of the workload stereotype and its properties for the application on *TestWorkload_BuyingProducts*. The workload information is either directly taken from the non-functional requirements (the PMM model), such as the number of concurrent executions and the delay between executions (also known as think time), or designed by the tester, as the ramp-up time, for example, the execution time and the start-up time, according to their past experience. For those values, the transformation generates default values based on typical values, for example, 60 minutes for the execution time, and 10 minutes for the ramp-up, but they can be adjusted as desired.

Figures (b) and (c) of Table 21 show the behavior of the test cases included in the workload. Figure (c) shows that the test cases were generated according to the coverage criteria on the PMM operands. First, the test executes the test case for *Create Client*, after a pause according to the *minDistance* parameter of the *before* operand of 10 seconds, then executes the test case for the creation of an invoice three times. To simplify the example, in (c) only one test case for the creation of clients and invoices was considered, but if there are more test cases they should be included in the diagram with different alternatives (using combined fragments). This can be seen in (b), where different alternatives were used to include the creation of products with valid and invalid situations.

TABLE 21 - GENERATED TEST MODEL WITH THE EXTENDED UML-TP



The example presented in Table 21, Figure (a) shows that the functional data, flow and verifications are performed as usual, but at the same time there is a time restriction for

verifying non-functional properties for the creation of products. As explained before, thanks to the proposed extension to UML-TP, the Arbiter can give a verdict summarizing all the response times, verifying the average of the response times obtained from the different executions of the test case, and verifying that the constraints expressed in the requirements model are reached. In this case it can be seen in the use of the constructor *average {0..15}* to give a verdict considering the average of all the response times, returning *pass* only if it is between 0 and 15 seconds.

Related to the dependability property, the expected number of correct responses (that is, our tolerance to failures) is indicated in the *workload* stereotype. This can be seen represented in Figure (d) of Table 21 in the *expectedDependability* attribute of the stereotype *workload* applied to the test case. The arbiter can then give a global verdict according to this expected value.

The information in the resulting model is enough to configure (manually or automatically through model-to-text transformations) a workload simulation tool combining the workload simulation scripts (Section 6.3 will explain how they are generated automatically).

Figure 56 shows the OpenSTA's user interface with the workload configuration represented in the test model. It is possible to see that it includes the two executable test cases (*TestWorkload_CreateProduct* and *TestWorkload_BuyingProducts*) with the corresponding configurations. In the figure it is possible to see the configuration of the virtual users for *TestWorkload_BuyingProducts*, where 100 virtual users are reproduced, introducing them in batches as represented in the ramp up (all users in 10 minutes). With this tool and the artifacts generated, it is possible to simulate the workload and analyze the results to give a verdict that considers the functional and non-functional aspects together.

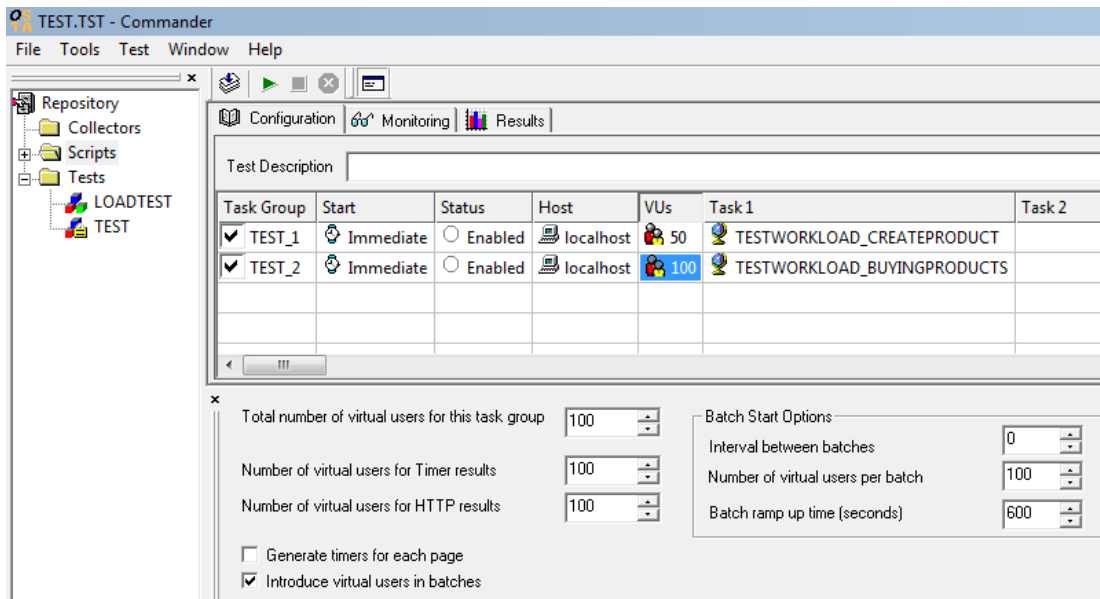
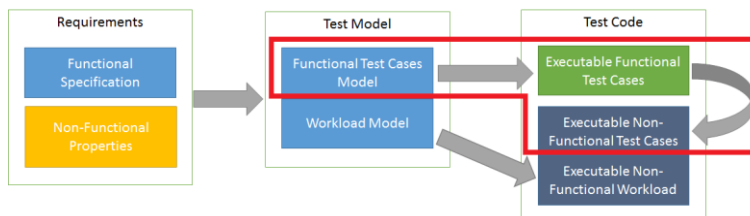


FIGURE 56 - WORKLOAD SCENARIO IN OPENSTA

6.3. EXECUTABLE NON-FUNCTIONAL TEST CASES



GENERATION

Different tools are available to automate the execution of functional test cases [132]. They simulate

interactions of the user against the system and are usually implemented and executed by “capture and replay tools”. Non-functional tests also consist of test scripts that are launched against the SUT, but they are described at a much lower level, which includes details on the communication protocol. Load simulation tools are used to concurrently generate multiple users connected to the SUT [59]. When the load is simulated, the infrastructure experts analyze the health status of the system, looking for bottlenecks and improvement opportunities.

Traditional workload simulation approaches have important drawbacks that make its automation very costly and demanding. The resulting testing artifacts are also very fragile, in the sense that they are very susceptible to changes in the SUT: modifications in the SUT (even bug fixes) often require the adaptation and maintenance of test cases for the next stage of regression testing.

Since functional test automation is much easier than automation for workload simulation (which includes ease of maintenance and comprehension), our proposal here is to take advantage of the functional test scripts to automatically generate workload simulation scripts. Focused on web systems, the idea involves automatically executing the functional test scripts while the HTTP traffic is captured. Later, the HTTP trace is analyzed to generate a workload simulation script model which is finally used to generate the script code to be executed by a load generator.

6.3.1. BACKGROUND AND MOTIVATION

Technically speaking, there is a big difference between test scripts for functional and performance testing (workload simulation). For example in AjaxSample, a Selenium script with only four lines was equivalent to a performance test script with 848 lines using OpenSTA. Those lines in OpenSTA correspond to each HTTP request sent to the server: taking into account that each request triggers a sequence of secondary requests, which correspond to images included in the webpage, CSS files, Javascripts, etc. Each request (primary or secondary) is composed of a header and a body, as shown in the example of Figure 57. The http message includes parameters, cookies, session variables and any kind of elements used in communication with the server. The example in this figure corresponds to the primary HTTP request as a result of the invocation of a search (filling an input and pressing the button “Search”). It includes the value “computer” in the parameter “vSEARCHQUERY” (the searched string inserted in the input).

Once the script is recorded, a number of adjustments must be performed in order to make it completely reproducible and representative of real users: for example, there is no sense in executing all the test cases with the same user name and password, the same search key (because of caches), etc. These scripts will be executed by concurrent processes (known as *virtual users*). The cost of the adjustments necessary depends on the automation tool and the SUT. In most cases, it is necessary to adjust the management of cookies and session variables (many of them must be unique or fulfill other restrictions). Adjustment of parameters in the header and body will also be required.

```

POST URI "http://localhost/sampleapplication/search.aspx HTTP/1.1" ON 1 &
HEADER DEFAULT_HEADERS &
,WITH {"Accept: */*", &
      "Accept-Language: en-US", &
      "Referer: http://localhost/sampleapplication/search.aspx", &
      "Cookie: "+S cookie_1_0+"; "+S cookie_1_1} &
,BODY "vSEARCHQUERY=computer&BUTTON1=Search;GXState=%7B%22_EventName%22%3A%22E~<27>SEARCH~<27>.%22%2C%22_EventGridId%22%3A45%2C%22_Ev" &

```

FIGURE 57 - EXAMPLE OF SOME LINES OF AN OPENSTA SCRIPT FOR A PERFORMANCE TEST

According to more than 20 workload simulation projects analyzed, the scripting phase takes between 30% and 50% of the total invested effort. On the other hand, the maintenance of these scripts (when the SUT changes) tends to be so complex that it is better to rebuild a script from scratch instead of trying to adjust it. As a result, the process becomes pragmatically inflexible. The test will generally identify improvement opportunities, which implies modifications to the system; however, our scripts will stop working if changes are introduced to the system.

There are some research lines that try to solve these problems. A summary analysis of these, and a comparison with our proposal was published in [133].

6.3.2. AUTOMATIC GENERATION OF EXECUTABLE TEST CASES

MANDINGA proposes an extension to the automation phase of the process that the author of this thesis presented in Vázquez et al. [106]. Instead of building the workload simulation scripts from scratch, the user has to provide a set of automated functional tests.

In this case, the prototype of this proposal was implemented as a GXtest module rather than part of DBesTest, taking advantage of some of the useful services it provides for these tasks and due to external requirements.

As shown in Figure 58, this tool builds a model of the HTTP traffic captured from the execution of a functional test script. The tool generates the script code for the preferred load testing tool with this input.

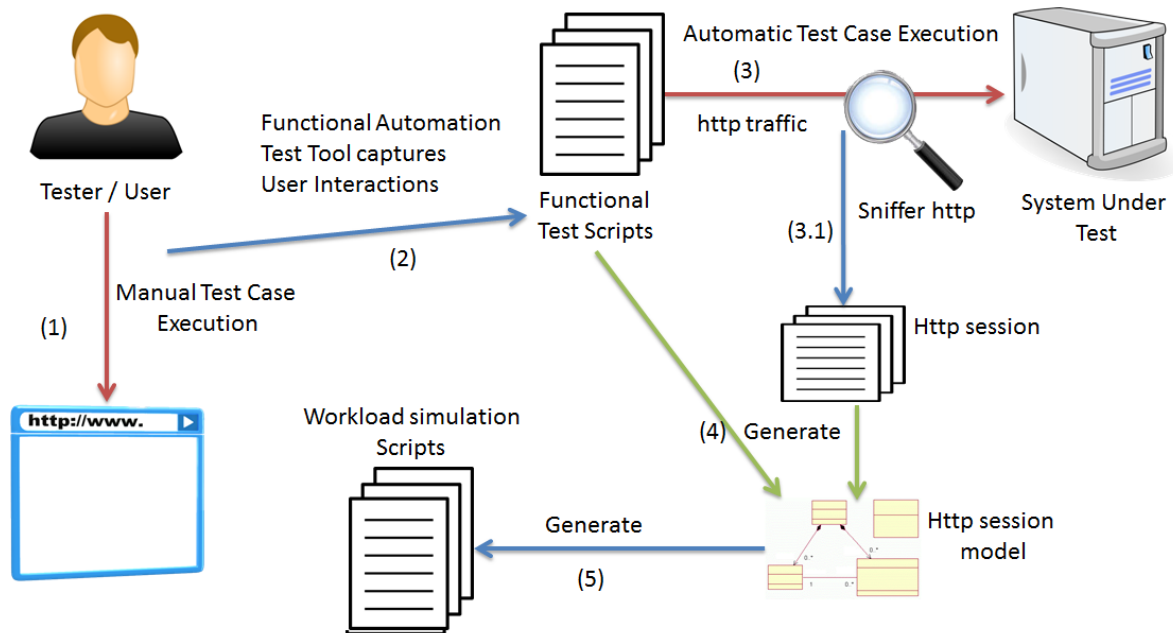


FIGURE 58 - PERFORMANCE TEST SCRIPT GENERATION APPROACH

GXtest executes Selenium and WatiN scripts, but it can be easily extended to more automated testing tools. According to the proposed methodology, during the execution of the functional test scripts, the HTTP traffic between the browser and the SUT is captured by an HTTP sniffer (a tool capable of capturing the network traffic) called Fiddler (fiddler2.com). With this information it builds a model that is used to generate the scripts for OpenSTA or JMeter. It is easily extensible to generate scripts for other load simulation tools.

Figure 59 shows the main elements of the HTTP traffic model designed to collect the necessary information. This model is useful to generate the workload simulation scripts. It is built using the information obtained by the sniffer (all the HTTP requests and responses) and by the functional test scripts, correlating the user actions with the corresponding HTTP traffic. It is therefore composed of an ordered sequence of actions, including invocations to the application through HTTP (*requests*), or *validations* of the response to verify that it is as expected. Each HTTP request is composed of a *header* and a message *body*. Both parts of the message are composed of parameters with their corresponding values. The header also has a set of fields that includes, among other things, cookies and session data. Each value can be hardcoded or can be taken from a data pool. It is important to keep the references between each HTTP request and its response, and with the corresponding functional test script command that generated them.

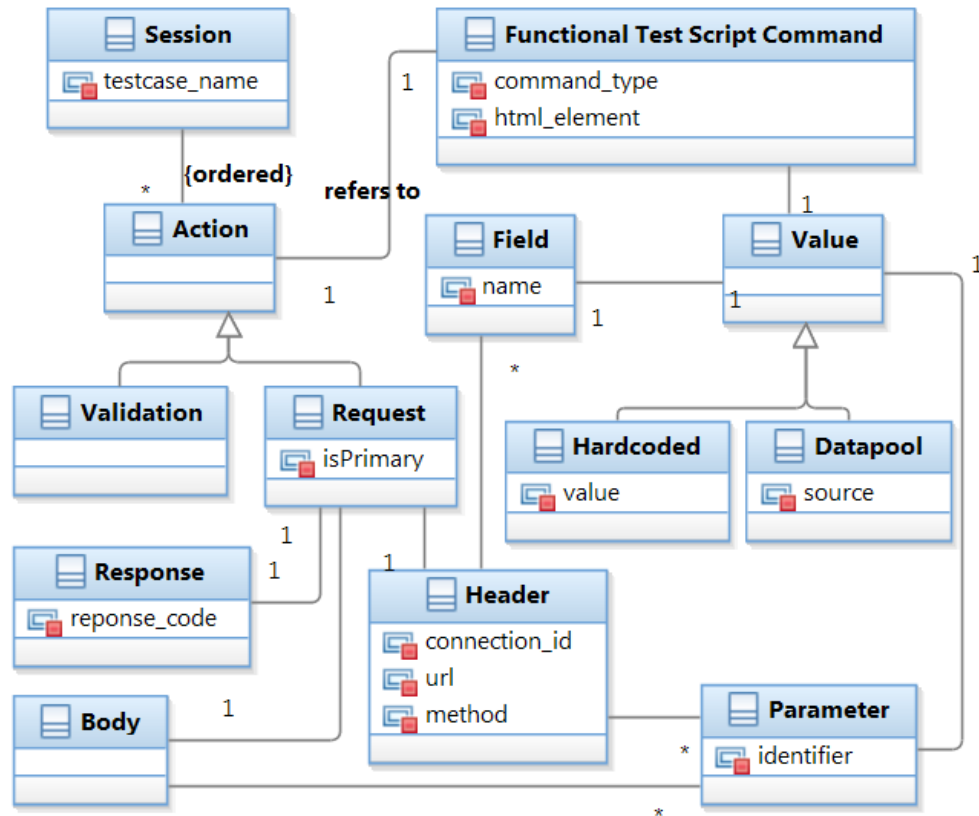


FIGURE 59 - HTTP TEST CASE METAMODEL

This model is used to generate code according to the language provided by the load generation tool. The generated code is specifically for OpenSTA or JMeter, according with the user's preferences for the workload simulation. To perform this code generation the tool uses an approach similar to that proposed in model-driven environments for the model-to-text transformations [17], where the code generation is defined by code templates for each element of the model. Table 22 includes some examples of those templates for OpenSTA; the first is for the general structure of the script, used for each test case of the model, and the second corresponds to an HTTP request, according to the specification of the HTTP protocol.

TABLE 22 - TEMPLATES FOR TEST CODE GENERATION

<pre> [template public generateScript(s: Session)] [file (s.testcase_name().concat('.scl'), false, 'UTF-8')] Definitions Timer T_TestCase_[s.testcase_name/] [s.variableDeclarations()] CONSTANT DEFAULT_HEADERS = "Host: [s.getBaseURL()]/ User-Agent: Mozilla/4.0" Code Entry USER_AGENT,USE_PAGE_TIMERS Start Timer T_TestCase_[s.testcase_name/] [s.processActions()] End Timer T_TestCase_[s.testcase_name/] Exit [/file] [/template] </pre>
<pre> [template public processRequest(r: Request)] Start Timer [r.name/] [if ([r.isPrimary/])]PRIMARY [if] [r.header.method/] URI [r.header.url/] HTTP/1.1" ON [r.header.connection_id/] & HEADER DEFAULT_HEADERS, WITH [r.header.processFields()/]} [r.processBody()/] [r.response.processLoadCookies()/] End Timer [r.name/] [/template] </pre>

As mentioned, in the traditional approach the resulting script must be adjusted after the recording. Many of these adjustments are very repetitive tasks. Our tool makes this kind of task automatic, using the templates mechanism. Some of these tasks are:

- Adding *timers* to each user action in order to measure the response time when executing the test scenarios, considering the type of actions performed in the functional test script and the corresponding HTTP requests for each one.
- Taking advantage of different design aspects of the functional test script, in the performance test scripts: (1) the data is taken from the same data pools; (2) the same validations are performed; (3) the same structure and modularization in different files promote the readability of the test script.

In this way the generated scripts are even better than when recording them with the OpenSTA or JMeter recorders and it helps avoid mistakes during the codification of the test scripts.

Once the scripts are finished, effort can be invested in the most important (and the most interesting and beneficial) part of a performance testing project: the execution of the load scenario and the system's behavior analysis.

For the example, only by providing the test cases *tc_create_product*, *tc_create_invoice* and *tc_create_invoice* (generated as it was shown in previous Chapters) is it possible to obtain the workload simulation scripts for OpenSTA or JMeter. It is necessary to combine them appropriately to simulate the expected workload and provide a verdict according to the non-functional requirements.

6.4. CONCLUSION

This section presented a novel and unified approach to model-based testing considering functional and non-functional aspects together. It particularly focused on model-based testing for non-functional requirements, integrating it with the results of previous chapters, specifically on the modeling languages used for defining a test model, and generating them from the non-functional requirements model. It also presented an improved methodology for the generation of workload simulation scripts, which is faster and more flexible than the traditional approach because it does not start from scratch, but from the functional automated test cases.

In this section an extension of PMM and another for UML-TP were proposed. The extension of UML-TP gives it more expressiveness for non-functional validations, and the extension of PMM enables it to associate different properties to the same workload, which can be compounded by different concurrent operations.

Another important aspect of this approach was the coverage criteria of PMM specifications, generating a test model able to test all the non-functional properties under the workload conditions specified in the model.

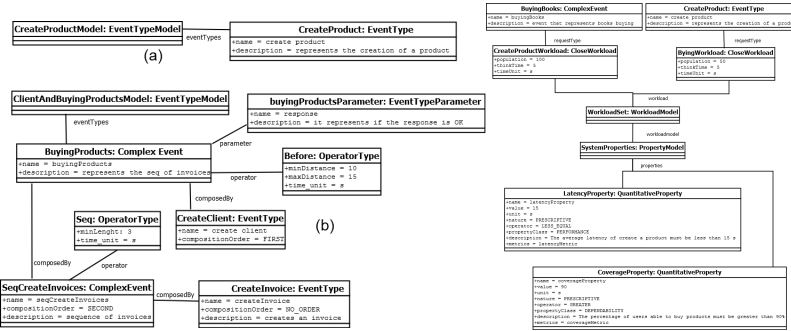
The traditional approach indicates performing functional test cases first, and then the non-functional ones if there is still enough budget to afford it. Our paradigm addresses this problem by modeling both kinds of test cases in one model, and then using these models to generate executable test cases able to verify all the quality aspects of the SUT at the same time.

Some threats to validity were identified in our proposal. First, only simple workloads are being considered. It is necessary for future work to improve representation by considering different necessities, such as allowing users to represent pikes, and different expected values (response times for example) according to the current workload. Finally, the proposal is valid under an important assumption: the test data and test cases must be independent; otherwise, the result of concurrent execution could be unpredictable.

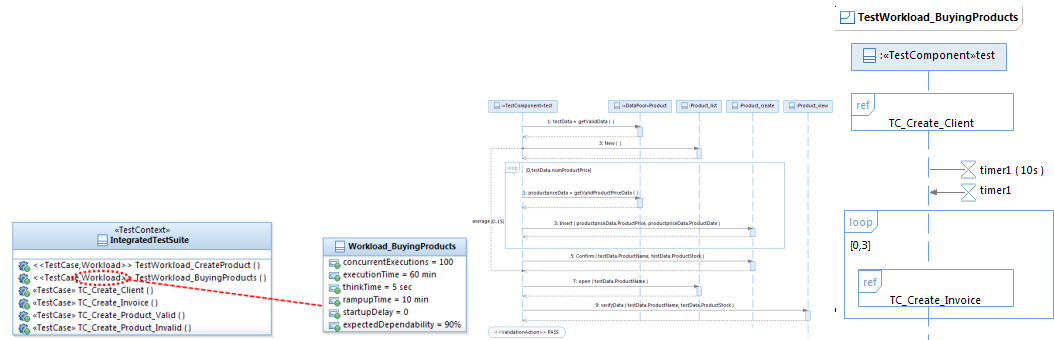
Despite the identified limitations and assumptions, the proposal promises to bring solutions to our customers for validating different aspects of the quality of their systems.

Figure 60 shows a summary of all the artifacts that are obtained by applying the presented methodology.

NON-FUNCTIONAL PROPERTIES WITH PMM



TEST MODEL WITH UML-TP CONSIDERING FUNCTIONAL AND NON-FUNCTIONAL ASPECTS



EXECUTABLE TEST CASES FOR A WORKLOAD SIMULATION PLATFORM

Task Group	Start	Status	Host	VUs	Task 1	Task 2
TEST_1	Immediate	Enabled	localhost	50	TESTWORKLOAD_CREATEPRODUCT	
TEST_2	Immediate	Enabled	localhost	100	TESTWORKLOAD_BUYINGPRODUCTS	

```

Start Timer T_TCL_ADDRESSBOOK
Cookie Reader:
/> ASP.NET_SessionId=033we452zeofyngqkq45
Fake Cookies:
PRIMARY GET URI "http://localhost/AjaxSample.NetEnvironment/home.aspx HTTP/1.1" OK 1
HEADERS DEFAULT_HEADERS
WITH ["Connection: keep-alive",
"Cache-Control: max-age=0",
"Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
"Accept-Language: en-US,en;q=0.5,en-gb;q=0.6",
"Cookie: *$cookie_1_0"]
WAIT 4560
Cookie Reader:
/> ASP.NET_SessionId=033we452zeofyngqkq45
Fake Cookies:
PRIMARY GET URI "http://localhost/AjaxSample.NetEnvironment/axproduct.aspx HTTP/1.1" OK 1
HEADERS DEFAULT_HEADERS
WITH ["Connection: keep-alive",
"Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8",
"Referer: http://localhost/AjaxSample.NetEnvironment/home.aspx",
"Accept-Language: en-US,en;q=0.5,en-gb;q=0.6",
"Cookie: *$cookie_1_0"]
WAIT 1059
Cookie Reader:
/> ASP.NET_SessionId=033we452zeofyngqkq45
Fake Cookies:
PRIMARY GET URI
"http://localhost/AjaxSample.NetEnvironment/Resources/GenXuxK/line_header.gif HTTP/1.1" OK 1
HEADERS DEFAULT_HEADERS
WITH ["Connection: keep-alive",
    
```

FIGURE 60 - SUMMARY OF THE CHAPTER

Istruitevi perché abbiamo bisogno di tutta la vostra intelligenza.

Agitatevi perché avremo bisogno di tutto il vostro entusiasmo.

Organizzatevi perché abbiamo bisogno di tutta la vostra forza”

– Antonio Gramsci

“Educate yourselves because we’ll need all your intelligence.

Agitate yourselves because we’ll need all your enthusiasm.

Organize yourselves because we’ll need all your strength.”

– Antonio Gramsci

CHAPTER 7. MANDINGA: IMPLEMENTATION, AUTOMATION AND APPLICATION IN THE INDUSTRY

As a result of this thesis, a set of prototypes was implemented, covering, to different extents, almost all the presented approaches. For each of them, different proof of concepts and some case studies in the industry were carried out. This Chapter shows the current state of the prototypes implemented to validate the proposed ideas. It also shows how all these proposals were transferred to the industry, as a part of our Action-Research plan.

7.1. STANDARD AND GENERALIZED APPROACH

DBesTest, the tool implemented to give support to the MANDINGA methodology, has been presented in previous chapters. It was implemented as an Eclipse plugin, integrating the extraction of database metadata with the execution of ATL and Aceleo transformation scripts. The idea is that, as Eclipse is a generic IDE with lots of extensions, it may be possible to manage the whole process from a single working environment. Along the same lines, PMM models are managed by a specific Eclipse plugin, as are UML models, transformation languages and different test execution tools.

The truth is that some of the process is not completely integrated in the same framework. In the majority of situations it was because of limitations imposed by base technology. Annex 1 - Difficulties Faced with Model-Driven Approaches, presents some of the difficulties faced during the implementation of the different prototypes.

Figure 61 depicts the current implementation of the framework, and the following subsections explain the current state and limitations of each part.

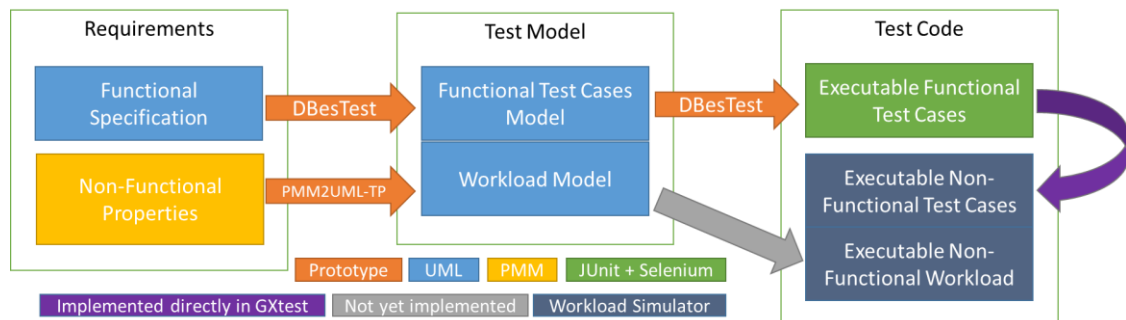


FIGURE 61 - PROTOTYPES IMPLEMENTATION

7.1.1. DEALING WITH MODEL-DRIVEN STANDARD TOOLS

The complete framework follows a model-driven approach from the beginning with the data model, to the last step with the test model. The test code is then necessary for the execution, but it should not be necessary to read, understand or maintain this code. There is only one area where it is necessary to correlate the model elements with the elements of the web pages, and this is also automatable.

The goal was to build an integrated framework, where it was possible to manage the whole process: the data model extraction, transformation execution, model edition, and test execution. Because of the current state of the different tools that are necessary to carry out this work, there are some limitations related to the maintenance of models and to model transformation. This subsection presents a summary of these.

7.1.1.1. UML AND MODELING TOOLS

If the tester decides to modify any model (data, information system or testing model) it is necessary to do so with an appropriated UML tool. Even though UML is a standard and one of the benefits of being a standard is that the different tools support the same metamodel, each tool has small differences in its implementation of the standard. UML tools are, in our humble opinion, not mature enough. Many different tools were tested (commercial and open source) to verify whether they can read a UML file and generate the corresponding diagrams.

Other researchers and practitioners have also observed this. Recently, Garzás²⁰ pointed out: *“However, UML specification is not still very clear, and tool vendors are still not interested in that the models created with their tools could be exported and used in any other tool without information lost.”*

²⁰ Dr. Javier Garzas - <http://www.javiergarzas.com/2013/11/uml-2-5.html>

There are many tools which support the UML standard, but some also claim to give support to the UML standard, or even to some UML Profiles, but if they cannot store or export the models to the correct XMI file format, the *obtained standard* is not the *actual standard*. This is why it is necessary to verify if the tool to use supports what is necessary. Table 23 lists the modeling tools that have been analyzed, indicating the main problems/limitations found.

TABLE 23 - UML TOOLS THAT HAVE BEEN TESTED

Tool name	License	Limitations
Papyrus	Open source	It was not possible to generate a diagram from a generated UML file. It is not possible to define reply messages in sequence diagrams, and to instantiate diagrams from UML/XMI files. Many errors, very immature.
Eclipse UML2tools	Open source	The tool has not been maintained since 2009.
Obeo UML Designer	Open source	Problems creating certain kind of diagrams. Not very mature.
Rational Software Architect	Commercial	Many bugs when editing models. The UML metamodel has some differences from the one in Eclipse UML SDK. Little support for UML 2.4.
Rational Rose	Commercial	It is not possible to export to UML standard XMI format.
Enterprise Architect	Commercial	The UML format is not compatible with the UML SDK. It was not possible to load a model with a profile.
Argo UML	Open source	It does not support the UML version required to work with ATL and Acceleo in order to process UML profiles.
Magic Draw	Commercial	Works reasonably well with UML models based on Ecore, even with profiles, but it was not possible to generate Sequence Diagrams from a generated UML file. The development team told us that this is not yet implemented.
Modelio	Open source	Very new and immature. Not complete.
Visual Paradigm	Commercial	It does not support UML 2.4 for importing and exporting UML files with Eclipse.

These were the only tools considered that claim to be respecting the standard, rather than just the graphical notation (as for example Creately²¹).

The former idea was to use a modeling tool integrated with Eclipse, so, DBesTest would allow the tester to work in a unified way, with everything in the same repository and managed with the same tool. For this, Papyrus [134] should be a good option (after 2009 Eclipse UML2tools was no longer maintained), because it is the modeling tool used in the Eclipse Modeling Project (<http://www.eclipse.org/modeling/>).

During the development of the thesis it was decided to use Rational Software Architect (RSA) [135]. This was the only tested tool capable of reading a UML file and generating the corresponding visualization diagram for the Sequence Diagrams in the model, as these diagrams were a key piece of the proposal.

²¹ Creately: www.creately.com

DBesTest uses UML SDK [121] to manage UML elements and to store the models as XMI files according to the standard. RSA can import and export diagrams to this format, so it is capable of interoperating with Ecore-based models and generating different diagrams starting from the imported models. Despite this it was necessary to program an import and export feature into DBesTest, specifically for RSA, because the UML model generated with UML SDK has some differences with the expected format of RSA and vice versa.

The process for working with the models is depicted in Figure 62, from the extraction of the database schema to the test model. It is necessary to export and import in each tool for each interaction between the different tools.

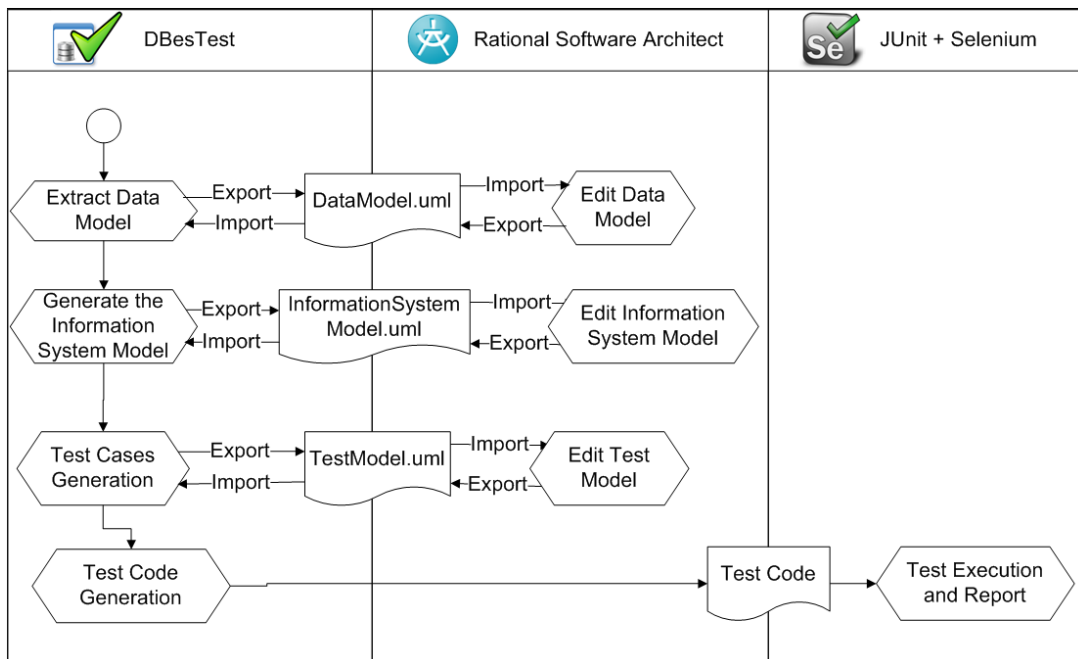


FIGURE 62 - MODEL DRIVEN PROCESS

It is expected that in the near future the tools will be able to interoperate completely respecting the standard, allowing us and the rest of the researchers, to use the different UML modeling tools and extensions, making the whole process easier.

7.1.1.2. MODEL TRANSFORMATION TOOLS

Similar problems appeared when working with model transformation tools. There are several proposals to work with model-to-model and model-to-text transformations, and although there are standards defined by OMG, none of the languages and tools seems to have the enough maturity to develop the transformations. Having worked with

different tools, even though the model-driven approaches are very promising, it is difficult to think of using them industrially.

Table 24 shows the model-to-model and model-to-text environments that were considered during the development of this thesis.

TABLE 24 - MODEL TRANSFORMATION TOOLS

Model-to-Model Engine	Description
MediniQVT	A tool set (editor, debugger, engine) that implements the OMG QVT Relations Standard (the declarative part). It is integrated in Eclipse as a plug-in, and it is freely available under Eclipse Public License. http://projects.ikv.de/qvt Last release: 2011-April
QVTo	A partial implementation of the Operational Mappings Language from the OMG QVT Standard (the imperative part). It is integrated in Eclipse as a plug-in, and it is freely available under Eclipse Public License. http://www.eclipse.org/mmt/?project=qvto#qvto Last release: 2013-January
ATL	A model transformation language and toolkit (editor, debugger, engine), also integrated into Eclipse as a plug-in. ATL is a hybrid transformation language, combining declarative and imperative constructors, based on the QVT Standard. http://www.eclipse.org/atl/ Last release: 2013-March
Model-to-Text Engine	Description
JET	A code generator, transforming abstract models into code. It is an Eclipse plug-in, freely available under Eclipse Public License. http://www.eclipse.org/modeling/m2t/?project=jet#jet Last release: 2011-February
Acceleo	A pragmatic implementation of the OMG MOF Model to Text Language. It is integrated to Eclipse as a Plugin, freely available under Eclipse Public License. http://www.eclipse.org/acceleo/ Last release: 2013-March
MOFscript	A tool and language to generate code, according to the OMG MOF Model to Text standard. The project is no longer available. http://marketplace.eclipse.org/content/mofscript-model-transformation-tool

MediniQVT was originally used for model-to-model transformations (mainly because it was the only official implementation of the QVT relations standard), but ATL substituted it because it does not have support for UML Profiles. MediniQVT has no longer been supported since 2011 (it cannot be installed in the newest Eclipse releases). On the other hand, ATL is the *de facto* standard, with better community support. It follows the guidelines of the QVT RFP, and is probably the most used. ATL supports any kind of model represented under the MOF metamodel. ATL is a hybrid transformation language, combining declarative and imperative constructors. The preferred transformation style is the declarative one, expressing mappings between the source and target model elements. However, imperative constructs make it easy to specify mappings that can hardly be expressed declaratively. There are many examples to show the usefulness of the tool, in a public repository called the *ATL Transformations Zoo*, where it is possible

to find more than hundred scenarios in which to use ATL, with documentation and the transformation code. Table 25 shows an excerpt as examples.

TABLE 25 - EXAMPLES TAKEN FROM ATL TRANSFORMATIONS ZOO

Input	Output	Description
UML class diagram	Relational model	To generate a database schema from a class diagram.
MySQL	KM3 (Kernel MetaMetaModel)	It translates data structure description from the database schema to the modeling space.
RSS	ATOM	It transforms from one syndication XML format to another.
UML	OWL (ontology format)	This transformation is used to produce an OWL ontology and OWL Individuals from a UML Model and UML Instances.
UML sequence diagram	UML State chart	The script transforms a set of UML sequence diagrams into a (hierarchical) UML state chart.

It is also expected that the tools supporting model transformations will continue growing and reaching the level of any modern development environment, which is a primordial requisite to be adopted, or even considered, by industry. Nowadays, the support of the tools is very limited, comparing mainly with modern development environments such as Eclipse for Java or Microsoft Visual Studio for C#. The limitations are not only in the development, but also in the debugging capabilities. The environment does not work properly when debugs, exceptions and problems received are not well described or documented. Annex I describes in more detail the problems experienced when working with ATL and UML, presenting their limitations.

7.1.2. DBESTEST



As already explained, an Eclipse plugin called DBesTest was implemented to automate the execution of different steps of the proposal. This tool was able to perform the complete process from the extraction of the metadata of the database, to the generation of the test code using JUnit [5] and Selenium [102]. It uses ATL for model-to-model transformations and Acceleo for model-to-text transformations.

7.1.2.1. DESCRIPTION

One of the main characteristics of DBesTest is the possibility of dealing with models for test case generation using well-known standards along the whole process, mainly from the OMG, and especially UML [11]. This opens up the possibility of using our method and its supporting framework with any UML modeling tool. For this, and considering that Eclipse is a very important and extended project, all the metamodels used are based on EMF and on the **UML SDK** [121] (an implementation of the UML 2.4 metamodel based on EMF).

Figure 63 shows a representation of the different components with which DBesTest interacts, explained below.

After the user configures DBesTest with the database connection information, it applies reverse engineering to load the database schema into a UML model representing the data model with a class diagram. For this task it uses *RelationalWeb* [30], which reads the database metadata, and gives an output in a proprietary XML format. This tool supports several database management systems (DBMS) such as Oracle, MS SQL Server, MySQL, etc., and can be easily extended to others. It was adapted in order to generate an output that was completely UML compliant, using the UML SDK [121]. The output can be displayed as an XML or tree structure or, using an appropriated UML modeling tool, it is possible to generate a graphic representation with diagrams (see section 7.1.1). For more details of the reverse engineering process refer to Chapter 4.

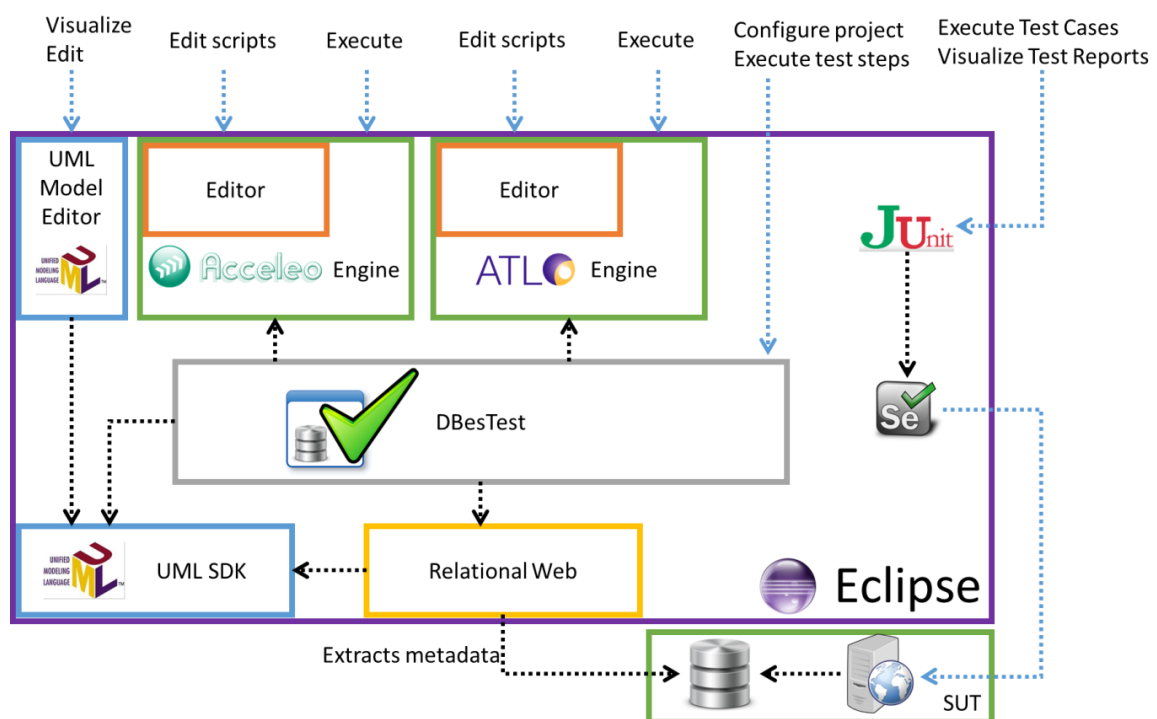


FIGURE 63 - DBESTEST ECLIPSE PLUGIN ENVIRONMENT

The tool allows the creation of a new type of project which is initialized with the folder structure shown in Figure 64. The Data Model is stored in the folder “datamodels”.

The tester is supposed to provide more information about the SUT through the Information System Model. In order to make this task easier for the tester, DBesTest has a set of ATL rules to generate a first version of this model, based on the data model that

was generated from the database structure. The Information System Model includes the data model, the GUI model (structure and navigation) and business rules in OCL. For more details refer to Chapter 4. For the generation of the Test Model, DBesTest also executes a set of ATL scripts. The current implementation of those scripts includes some test patterns, showing the feasibility of the approach. The ATL scripts are managed in such a way that it is possible for the tester to modify the rules and add new rules, in that way adding new generation strategies without changes in the tool. If the user wants to implement more UI or test patterns, it is thus enough to extend these ATL transformations. There is a folder in the DBesTest project for each group of ATL scripts: “Scripts/UI_Patterns” and “Scripts/Test_Patterns” respectively. Refer to Chapter 4 for the generation of the Information System Model, and to Chapter 5 for the generation of the Test Model.

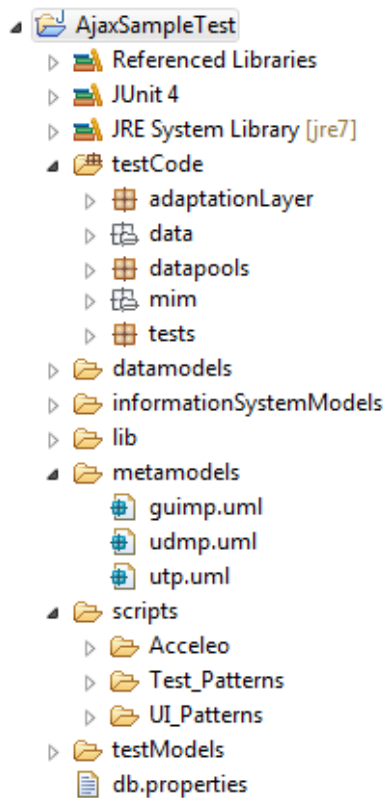


FIGURE 64 - FOLDER STRUCTURE OF A DBESTEST PROJECT

After the execution of the transformations, the models are stored in the corresponding folder of the project structure: in the “informationSystemModels” and “testModels” folders.

DBesTest implements the code generation with Acceleo scripts. The scripts provided generate code using JUnit and Selenium and are stored under the folder “testCode”. Under this folder (included in the project build path in order to make Eclipse compile it) there are different packages: “tests” includes the test classes with the test cases, according to the sequence diagram specifying its behavior; “datapools” contains classes to access the test data, which is stored in the folder “data” as “.csv” files; the “adaptationLayer” has classes responsible to resolve the relationship between the models and the SUT, using the information stored under the folder “mim”.

The Acceleo scripts (under the folder “Scripts/Acceleo”) are easily extendable or interchangeable, with the goal of having the ability to use a tool other than Selenium if the tester has different preferences. The code generation was also explained in Chapter 5.

Under the project’s structure there is also a folder to store the metamodels (UML-TP, UDMP and GUIMP).

Some common bugs have been found thanks to the use of our approach, such as:

- It is common to define the maximum size of a string column and its corresponding field in the web page with a different size. Depending on the implementations, sometimes an exception is thrown, or an error message is displayed, and sometimes the input is truncated to the length of the input, meaning a loss of information. Errors of this kind are discovered because the test data generator considers the sizes of fields and columns, generating valid and invalid data.
- As the maximum value that can also be stored for numeric values is known, an overflow was found, calculating the subtotal when creating invoices selecting products with the highest price representable in the column where it is stored.
- During the process, the tester is guided to verify the lifecycles of the entities, and the relationship between tables and pages. This process was useful to detect inconsistencies and obsolete attributes (columns that remain in the tables, but were not used in the logic and user interface).

As can be seen, the approach is able to find different kinds of errors. The test cases were generated using the information extracted with reverse engineering methods, which shows that this information is useful for designing and identifying interesting test scenarios.

7.1.2.2. LIMITATIONS

Some limitations were identified during the implementation of the prototypes. Some were time-related, the focus is not on them but on others more associated with the approach itself. They are listed below, grouped by the process stages.

Information System Model construction: Reverse engineering does not have a single solution for all the platforms, technologies, etc. DBesTest was implemented with one particular solution, and in any case, this module could easily be changed by any other able to generate a UML model.

There is more information in the database metadata that is not currently read and that could be exploited in order to generate test cases: check rules, store procedures, triggers, cascade updates and deletes, etc. By analyzing these it could be possible to generate more and more precise Business Rules. Since this requires the syntactic analysis of the SQL code (which varies among vendors), this aspect has not yet been addressed.

Test Case Generation: The main problem with this approach is that the quality of the generated test cases and test data depends on the information added by the user. Some tests could therefore give false positives: they report an error where they should not. This happens when the user does not add certain business rules that mark some input data as invalid; the test cases will then report an error in the SUT when the error is in the input test data.

Test Code Generation: The tester has to provide the MIM (model-to-implementation matching) as a properties file. Although there is a utility implemented in order to simplify the task of selecting the HTML identifier and associate it with the model element, it is still a demanding task affecting the scalability of the solution.

Modeling: Evidently, it would be better to work in a framework able to integrate test case modeling and execution, but this was not possible because of the existing gap between the promises of the standards and the current state of the tools supporting them.

Model-transformation approach: the implementation of the prototype showed us that the design and developing of model-transformation rules is not as mature as modern programming environments and languages. It is not easy to build transformations that are easy to maintain and extend, and there is little research about the topic. Finally, *declarative language* is not synonymous with *easy* and *clear*, or *user-oriented*. Actually, many transformations are much more difficult to code when trying to build them in a declarative way. One of the bases of our framework is to be extensible, allowing the

user to introduce new test patterns in a declarative way because this approach is always sold as “closer to the user” or “with a higher abstraction level”, but for us the reality is that it is not easy to generate new rules for this aim.

7.1.3. GENERATION OF AUTOMATED PERFORMANCE TEST CASES

This was implemented directly on GXtest because the benefits of the idea were visualized as positive before its proof of concept, and there were specific projects which required the tool and paid for it. Moreover, the implementation of a prototype for a proof of concept would have been very expensive, because it is necessary to deal with the HTTP protocol, and it would have been a disposable prototype (without the ability to evolve) because GXtest is implemented in C# Dot NET environment, and DBesTest is in Java and holds model-transformation languages.

7.1.3.1. DESCRIPTION

Figure 65 shows a feature of GXtest thanks to which it is possible to import and integrate Selenium scripts in a GXtest project (GXtest has its own test case modeling tool and format). The Selenium scripts generated with DBesTest can therefore be imported in GXtest in order to use the functionality described in Chapter 6, to generate scripts for workload simulation in OpenSTA or JMeter.

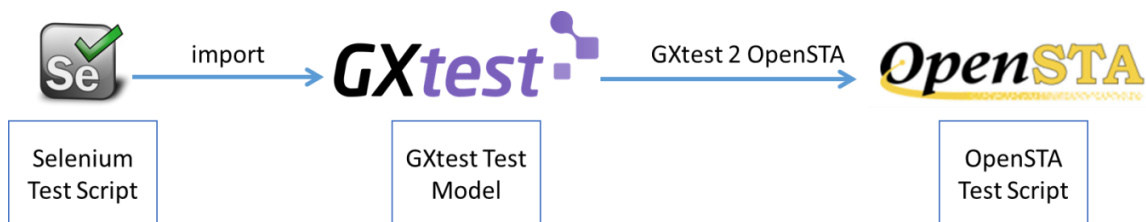


FIGURE 65 - WORKLOAD SIMULATION SCRIPTS GENERATION

7.1.3.2. LIMITATIONS

The limitations are presented in section 7.2.3.4, after showing how the tool was applied in different real projects.

7.1.4. GENERATION OF NON-FUNCTIONAL TEST SCENARIOS

The PMM2UML-TP prototype was developed during the research stage in the CNR, Pisa, Italy. The implementation is not complete; it was focused on verifying the feasibility of the transformation approach for the coverage criteria defined for PMM event operators, considering only some them and only some non-functional properties and metrics.

7.1.4.1. DESCRIPTION

Our main goal in this phase is to provide a unified framework to allow testers to verify functional and non-functional requirements of a system following a black box and model-driven approach. Our proposal assumes that the functional design of the system is given by UML models, and the non-functional specification of the system is modeled with PMM.

This proposal therefore extends the DBesTest framework with the specification of non-functional properties: in this case, given as a PMM model. The test model generated with DBesTest is extended with time restrictions and with a workload definition, in order to represent a test suite where the test cases are executed concurrently.

Taking into account the functional test cases generated with DBesTest and the non-functional requirements specified in PMM model, PMM2UML-TP automatically generates a test model (represented with our extension of UML-TP) addressing both aspects together, verifying functional behavior and non-functional properties. This model will be used to automatically generate a set of executable test artifacts and obtain a verdict about the functional and non-functional behavior of the system.

7.1.4.2. LIMITATIONS

One of the limitations evidenced with this approach is that PMM is not standard or well-known, and the modeling of properties, metrics and workload is not simple. Therefore, there is an extra effort required for the modeling of these requirements in order to take advantage of the workload model generation. It is necessary to evaluate whether this approach is better than allowing the user to model the workload in the test model using the extended UML-TP.

7.1.5. CONCLUSIONS ABOUT THE PROOF OF CONCEPT

In the examples presented in Chapters 4, 5 and 6, it was possible to see how the different tasks previously performed manually were executed with the assistance of the different prototypes in a semi-automatic way, with an integrated approach considering functional and non-functional aspects of the SUT.

Even though there were some limitations identified with the approach, and there is a lack of empirical validation, the prototype was very useful for its purpose: to demonstrate the feasibility of the approach, and identify the most important risks and limitations of the approach. The implementation of the prototype gave us confidence to translate these ideas to the industry, applying them to the specific context of Abstracta, and successfully extending the power of GXtest tooling.

7.2. TRANSFER TO INDUSTRY: GENEXUS AND GXTEST

Most of the research in this thesis was directly transferred and experimented with in the industry. In the specific context of the GeneXus community, it was implemented directly in GXtest.

This section presents our most important empirical validation, explaining how the equivalent tools were implemented in Abstracta and used in real projects, either by the Abstracta team or by some of its customers.

Figure 66 shows the different tools and extensions that were implemented or used for each equivalent part of the thesis proposal. In this case, the functional requirements are not provided by UML but in GeneXus models. These models are used to automatically generate test cases that are stored in a test model implemented in GXtest instead of UML-TP. In GXtest, the test model is executable by the same tool, it is not necessary to translate it to test code in order to execute it. Finally, the test model is used to generate scripts for performance testing using OpenSTA.

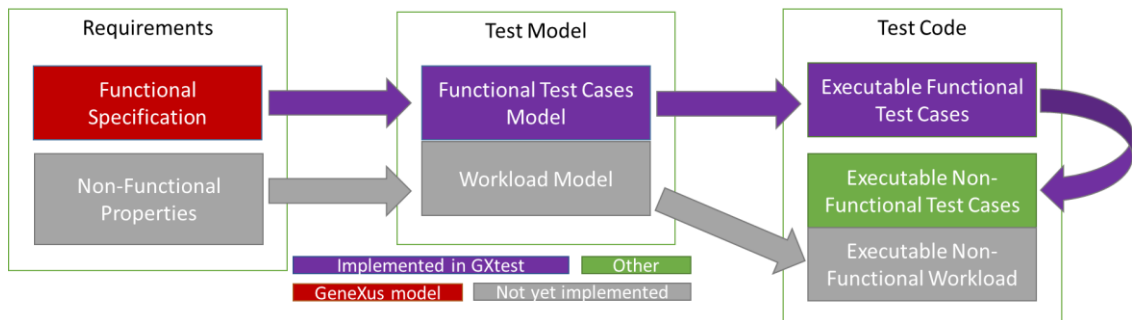


FIGURE 66 - IMPLEMENTATION IN GXTEST

In the figure it is possible to see that the last part of our research was not yet implemented in GXtest. It corresponds with the part related to PMM models, and non-functional workload generation.

This section presents the case studies performed in the industry with these implementations. Therefore, below is a deeper introduction to GeneXus and GXtest (for a better understanding of the reader) and then the functional test cases generation and performance test cases generation case studies are presented.

7.2.1. BACKGROUND: GENEXUS AND GXTEST

GeneXus™ GeneXus is a tool which has been developed and maintained since 1988 by the Uruguayan company Artech. The tool

is nowadays spread out through Latin America, United States, Canada, China, Japan, Spain and Italy, counting with more than 100.000 users within 7.000 companies.

The tool enables the automatic creation, development and maintenance of Information Systems, mainly to manage the information stored in a database. GeneXus permits users to represent the application, regardless of the technology involved, in a model called Knowledge Base (KB), from which it generates applications in multiple environments (servers, PCs, mobile devices and the Cloud) and languages (Ruby, Java, C#, Objective-C, Cobol, RPG, Visual FoxPro, among others).

From the specification of the data model, it generates tables for the database, and the logic and presentation layers for the creation, reading, updating and deletion operations, allowing the user to extend data processing with business rules definition and a high level programming language. For example, Table 26 shows some business rules extracted from the *Client* entity from AjaxSample.

TABLE 26 - GENEXUS BUSINESS RULES EXAMPLES

<code>error("You must enter a name for the client.") if ClientFirstName.IsEmpty();</code>
<code>error("The initial balance cannot be negative.") if ClientBalance < 0;</code>
<code>noaccept(ClientLastUpdated);</code>
<code>ClientLastUpdated = servernow() if after(confirm);</code>

The first two lines work by verifying the user inputs, the first ensures that the client name is not empty, and the second that the initial balance is not negative. The last two lines show an example where an attribute *ClientLastUpdated* is maintained, registering the last modification timestamp of each registry. The system does not allow the user to insert data (the *noaccept* rule) and inserts the server time automatically after any submission of the related form. Any of these rules are executed every time the entity is accessed, whether it is through the user interface, or via the system's logic through a managing data procedure.

It is also important to note that GeneXus generates system code for several platforms, including green screen, client server, web, mobile, and in different languages (for the web for example, it is possible to generate Java, C# and Ruby).

In Model-Driven Development (MDD) environments, such as that offered by GeneXus, models are used to generate the source code of the application, based on the specification given through those models. The traditional automation testing tools are prepared to work at a source code level, but do not do so with the models with which the development team is used to working, which brings certain difficulties:

- Necessity to understand/manage the source code generated, while the goal of MDD is to avoid such a thing.
- High maintenance cost of the testing artifacts while regenerating the source code.
- The same model can be used to generate code for different platforms, but the automated test cases must be built for each platform, and thus increase the cost of maintenance.

To address this problem a new tool was proposed that manages the testing artifacts with models at the same level as the development models. In this way those artifacts will be easier to maintain, and they can be used to perform the testing of the different platforms for which the application has been generated.



That motivation brings us to GXtest, which is a tool that was developed by the Uruguayan company Abstracta in 2007. After 2009, Artech decided to start distributing and commercializing the tool, integrating it to the GeneXus suite, since they visualized the benefits that GXtest offers to the GeneXus users. GXtest then became a commercial tool, having started as an innovation project.

GXtest allows the test automation for web systems developed with GeneXus, at the same abstraction level of GeneXus, i.e., associating test artefacts with the KB elements instead of with the system's generated code elements as in the traditional approach (Selenium, WatiR, etc.). The test cases are represented with models, in order to follow the GeneXus philosophy and for the sake of ease of use, in order to allow domain experts to develop test cases, and not just technical users with programming skills.

Firstly, the user creates a new test project indicating the KB to test. This is equivalent to the Information System Model, including the data model and the different elements with which the user interacts (pages, inputs, buttons, etc.), and its navigation.

The test cases are related to the elements of the KB, and GXtest knows how to map these elements with the correspondingly generated HTML element when it executes the test cases. This means that the MIM (model-implementation matching) is not necessary, it is built automatically. Figure 67 has a representation of a simple example in order to show how the test model actions reference the KB elements. GeneXus will generate the system's code according to the KB specification. GXtest will then find the generated elements (it does not matter whether the application is generated in Java, C# or Ruby) because a detailed analysis of the code generation performed by GeneXus is realized to each web platform.

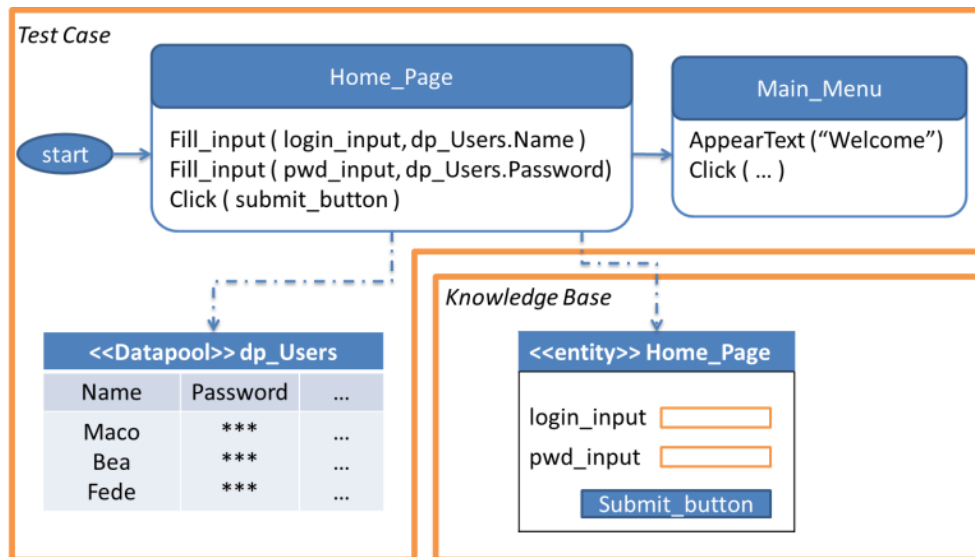


FIGURE 67 - GXTEST CASE EXAMPLE

Test data are stored in datapools, which are equivalent to the UML-TP concepts. The test flow is represented by something similar to an activity diagram that can be considered equivalent to a behavior diagram in UML.

It is possible to see that GXtest manages the test case behavior with a graph which is more similar to a state machine or activity diagram. It is also important to see here that the test flow in GXtest is executable on the SUT, because it automatically does the mapping between the model elements and the HTML elements. Comparatively, in the UML model it is necessary to solve this gap with extra information provided by the user.

In order to ease the creation of the test cases, GXtest follows the “record & playback” approach [6]. It offers a component (called *GXtest Recorder*) which is installed as an extension of Internet Explorer, able to capture all the user actions and generate from that the test model to play them back later. This component is able to associate the HTML element with which the user interacted, with the corresponding KB element. In that way, the test model only keeps references to the KB and not to the platform specific elements. For more detail about GXtest refer to its online user manual [136].

Table 27 shows the GXtest versions releases, which depict the correspondence between the research on the prototype and the research in Abstracta on their product.

TABLE 27 - GXTEST VERSIONS

Version number	New Features	Released	Thesis contributions
1.0	<ul style="list-style-type: none"> Record and playback Adaptation layer for GeneXus applications Data driven testing Test management 	2009	
1.1	<ul style="list-style-type: none"> Generation of test cases (GXtest Generator) – beta version Multiplers browsers execution (Internet Explorer, FireFox, Chrome) Improved commands Bug fixes 	2012	<ul style="list-style-type: none"> Generation of functional test cases with the same strategy as DBesTest (based on the data model)
2.0	<ul style="list-style-type: none"> GXtest Generator complete Mobile testing Generation of OpenSTA scripts <i>VerifyResponseTime</i> command in functional test cases Bug fixes 	2013	<ul style="list-style-type: none"> Generation of performance test scripts from functional test scripts Non-functional validations in functional test cases
2.1	<ul style="list-style-type: none"> Generation of JMeter scripts Performance testing for mobile Bug fixes 	Not released yet	<ul style="list-style-type: none"> Generation of JMeter scripts

Two different satellite tools were developed for GXtest, one for the automatic generation of test cases based on the data model (KB GeneXus), and another to generate performance tests from a functional test case. The first corresponds to the DBesTest research, and the second has already been explained, and was directly implemented on GXtest, without the implementation of a previous prototype. Of all the research presented in this thesis, there is only one aspect that has not yet been transferred to the industry, which corresponds to the last period of research, and involves the generation of non-functional workloads (from the PMM models). Only the command *verifyResponseTime* was added to the GXtest test cases in order to manage non-functional validations in the functional test model, as presented in the extension of UML-TP.

Below is explained how each topic researched in this thesis was translated to an improvement or a new feature in GXtest. Our first experiences using both components in the industry are detailed and analyzed.

7.2.2. GXTEST GENERATOR

GXtest Generator is basically the adaptation of DBesTest to GeneXus and GXtest. It follows an MDT approach to generate a test model (conforming the GXtest metamodel) from a data model designed with GeneXus (conforming to the KB metamodel). The generated test model can verify whether the application correctly manages the data structure. The test model will therefore include test cases for the CRUD operations of

the SUT. These test cases are also useful for the tester as building blocks in order to develop other test cases.

7.2.2.1. DESCRIPTION OF THE ADAPTATION TO GENEXUS

Table 28 shows a mapping between the research for DBesTest and what and how it was implemented in GXtest Generator.

TABLE 28 - RELATIONSHIP BETWEEN DBESTEST AND GXTEST GENERATOR



 DBesTest	
Information System Model. It is composed of the data model, GUI model (structure and navigation) and business rules. It is generated from the database schema with assistance of the tester.	GXtest directly uses the KB, which is the source model for the applications generated with GeneXus. It includes the data model, GUI and business rules. GeneXus provides a library to manage those elements directly.
Test Cases Generation. The test cases are generated based on patterns identified on the ISM, by executing model-to-model transformations. The test cases are generated in UML-TP.	The test cases are generated based on patterns identified in the KB, using the library provided by GeneXus, and generating the test cases in the GXtest model. The algorithm is implemented in C#.
Test Model Representation. The test model is represented with UML-TP. The main concepts are: Test Context, Test Case, Datapool, Data Partition, Data Selector, etc.	GXtest has its own and proprietary metamodel to represent and store the test cases. However, there is a direct correspondence between its elements and UML-TP presented in Table 29.
Test code. The test code is generated from the test model in order to be able to execute it on the SUT. This generation is made with model-to-text transformations.	The test model is executable. It has enough information to allow the test engine reproduce the user actions of the test case.
Adaptation Layer and MIM. The generated code is not completely executable on a specific platform unless the tester provides the MIM information, and the model-to-text transformations generated the adaptation layer for a specific tool such as Selenium.	As the test model is executable, it is not necessary to provide MIM information, and the adaptation layer is automatically provided by GXtest.
Test data. The test data is generated considering different coverage criteria such as AEM from Andrews et al. [111], and the violations of the database schema, as presented in Chapter 5.	The test data generation strategy was implemented with the same approaches.
Data-based oracles. According to the validity of the selected data for a test case, it verifies whether the entities are created or not, updated or not, deleted or not, or viewed with the corresponding data.	The oracles were designed with exactly the same strategy.

Table 29 shows the correspondence between UML-TP concepts and the artifacts in GXtest.

TABLE 29 - CORRESPONDENCE BETWEEN GXTEST ARTIFACTS AND UML-TP CONCEPTS

UML-TP Concept	Corresponding Artefact	GXtest	Description
Test Context	Test Suite		Group test cases.
Test Case	Test Case		Ordered set of actions and validations. In every action and validation it is possible to use static test data or take it from datapools.
Datapools	Datapool		Tables storing test data.
Data Partition	Data Scope		It is possible to divide the test data into datapools for different purposes, different test cases, etc.
Data Selector	Command DPnext()		It initializes the variables with data from the datapool according to the Data Scope selected.
Test Components	Executors		The test cases can include different actions or events in order to interact with the SUT. There are special components called "Executors" able to read this test specification and initialize the test, reproducing all the actions and validations in the test cases.
SUT	KB		GXtest keeps a copy of the KB of the system under test, in order to have information about the different elements that the test cases will be interacting. This information is useful for the test engine and for the test oracle.
Behavior Diagrams	Test Models (directed graphs)		Instead of sequence diagrams, only directed graphs (similar to activity diagrams) are used. They represent pages and transitions, and in every element there are commands to simulate the user actions.
Validation Action	Validation commands		There are specific commands to make validations, in the GUI or at a database level.

GeneXus is able to generate system code for CRUD operations for each entity defined in the data model. In the same way, GXtest Generator is able to generate executable test cases for CRUD operations for each entity of the data model, and then combines these test cases to test the whole life cycles.

The generation algorithm basically follows these steps:

- For each entity on the KB, the generator determines whether it is possible to generate tests (which means, if the entity has a table, whether it is persistent), and in this case it generates the test flows for each test case and to invoke the actions.
- For each attribute in the entity it is determined whether it is necessary generate data (if the attribute is editable) and how to do it.
- **The same test pattern strategy presented for DBesTest is applied to generate a test model.**
- After generating the test model (test cases and test data) the user can manage, edit and execute it with GXtest.

In order to correlate a test case represented in UML-TP with those in GXtest, Figure 68 shows a sequence diagram for the creation of an Invoice at the top (as it is generated by the DBesTest approach), and at the bottom the same operations, on the same system

(AjaxSample) according to the graphical representation of a test case model in GXtest (generated with GXtest Generator).

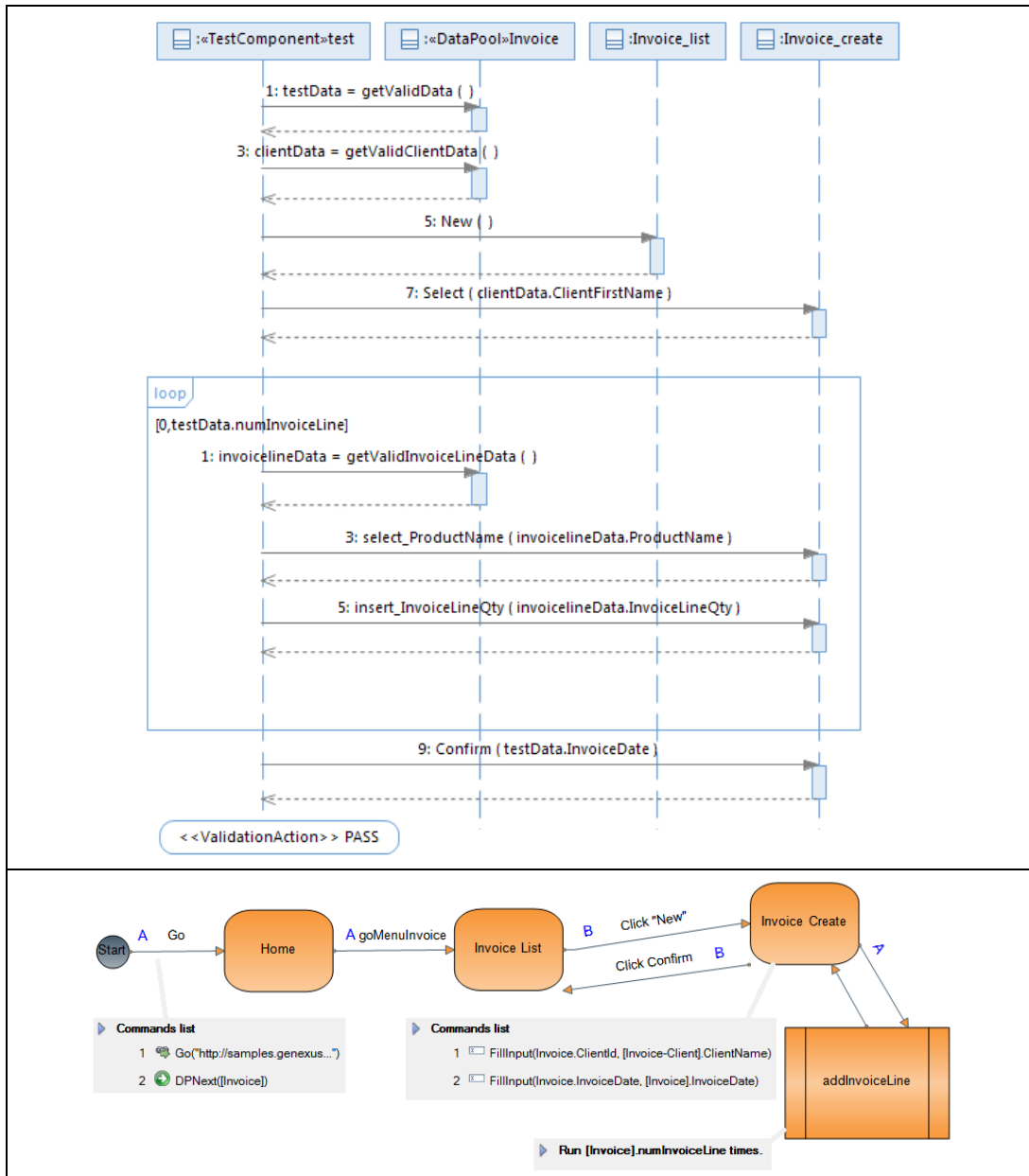


FIGURE 68 - COMPARISON OF A UML-TP TEST CASE FLOW AND THE SAME ONE IN GXTEST

In the figure it is also possible to see the representation of test cases in GXtest as directed graphs, where the circled nodes represent pages, boxed nodes represent the inclusion of another test case (equivalent to *interaction use* and *combined fragments* in UML sequence diagrams), and the edges connect them, indicating the flow. Each element has a set of commands which simulate the user actions and verifications.

There are some improvements over the test generation strategy of DBesTest. On the one hand, the user has the possibility of defining dictionaries for the different attributes or for a data type. In that way, to fill the datapool for “Products”, the product names can be taken from a file provided by the user with real product names. On the other hand, the business rules are loaded directly from the KB, such as those presented in Table 26. The user does not therefore need to provide the business rules model, and the test cases and test data are going to be generated taking them into consideration. The test data is also generated testing the boundaries of the conditions defined for these business rules, improving the coverage of the test set.

The main advantage of GXtest Generator over DBesTest is that the adaptation layer is not necessary, because GXtest deals with it. One of the identified limitations of DBesTest was scalability due to the fact that the user had to provide the Model-Implementation Mapping (MIM), indicating the association of each element in the model with the element in the GUI of the SUT. In GXtest this is done automatically, reducing costs and ensuring the scalability of the approach.

GXtest Generator has been used in different scenarios²²:

- To initialize the testing environment (*One Click Start-up*). The tester can generate automatically and with zero cost, a set of test cases for all the entities of the SUT. Then, they combine them and generate more complex ones.
- To reduce risks in the migration from one platform to another (generating test cases and comparing both executions).
- To minimize risk and costs, executing smoke tests with low cost (by executing simple test cases generated completely automatically, verifying that the SUT was generated correctly, and executing basic functionality, trying with different browsers, environments, configurations, at almost no extra cost).

7.2.2.2. EXPERIENCES IN INDUSTRY

The tool has already been sold to more than 65 customers, as an optional feature of the GXtest framework. The Abstracta team has used the tool in example applications in order to validate the implementation, and in a real project that is reported below, validating its scalability.

²²

More

information:

http://gxtest.abstracta.com.uy/wiki/index.php?title=GXtest_Generator_Tutorial

7.2.2.2.1. STUDY CASE: BANCARD PARAGUAY

This case study corresponds to the first real project in which the automatic generation of a test with GXtest was used. This project was useful for finding important errors in the SUT at low cost, validating the approach and giving the confidence to continue the research to improve and extend it.

The SUT was one of the main systems of a financial company from Paraguay called Bancard²³, which is responsible of most of the financial transactions by credit card and POS (point of sale) in this country.

The company decided to migrate the platform of their system, which was the main cause of the necessity for execution of a complete regression testing. The development team did not know how the new system would behave on this new platform.

Thanks to GeneXus the migration from one platform to the other is very simple, but the test case execution would have cost a lot; it is a big system developed for many years (more than one thousand entities). The manual execution of the test cases therefore implied a very important cost in time, effort and money, making these factors a good opportunity for the validation of our proposal.

The system was migrated to Java/Web on IBM infrastructure, including WebSphere application servers and a DB2 database on an AS400 machine. The system was developed in five separated KBs, which can be seen as five different modules, in order to simplify the management of the entities. The scope of the test project focused on four of these five KBs: Access Control, General, Authorization and Switch. The fifth module was not yet released.

Table 30 shows an overview of the project results. The test team, which included the main developer of GXtest, was involved in the project for one month. There were four test cycles defined, one for each module. Between each test cycle different improvements were included in the tool based on the experience obtained in the previous test cycle. They were executed in the order of the table rows, starting with the smallest entities to test.

The table shows, grouping by module, the number of test cases that were executed. The numbers refers to tests that verify the complete life cycle of the entities involved: creation, update, list (and search) and deletion. Many entities do not have a graphic user interface, or else one page in the user interface stores the information in different

²³ Bancard: <https://www.bancard.com.py>

entities (as the example presented in AjaxSample for invoices, which has four tables associated with the creation of this entity). The test team defined the scope with a team from Bancard, deciding which entities consider for this first project.

For each test case different test data was also generated, according to the strategies already presented in this thesis, as for example equivalence classes and boundary values, taking into account the metadata of the data types and business rules.

TABLE 30 - SUMMARY OF GENERATED TEST CASES

KB	Number of Entities	Executed Test Cases	OK	Error	Warning
Access Control	155	16	10	2	4
General	415	81	38	15	28
Authorization	423	29	18	6	5
Switch	604	81	33	33	15

The time invested by the tester was mainly to verify the test execution reports, and corroborate whether it was a bug (reporting it to the development team) or whether it was a problem with the test case generation strategy. The “warnings” column in the table corresponds to those test cases which mistakenly reported error. Generally, this implied a correction to the test generation strategy. In some other cases, the error was originated because the test case tried to delete an object that it did not have a “delete” operation available from the GUI. This is not an error, but it is important to verify the absence of this functionality with a domain application expert. In other circumstances, the warning corresponded to test cases which failed because the tester did not have the required permissions on the system.

7.2.2.3. ANALYSIS OF ERRORS FOUND

Within the incidents detected in the project and in some example applications, there were some error types that should be highlighted:

- Inconsistencies between database and logic layer: every time the development team make changes in the KB (development model) then GeneXus prepares programs to propagate those changes in the database schema automatically, including data migration. Due to environment management, there were problems identified in the migrated application where the logic layer of the application did not correspond with the database schema. It is possible to see these kinds of errors when a user accesses the corresponding entity pages, but others are not that evident. Some tests revealed that the database accepted 40 characters in one column, but in the logic and presentation layers this field was changed in order to support longer strings. The test case using a longer string therefore provoked a not captured exception. This test case was designed based on the column data type.

- Non-existent resources: due to problems of various kinds, mainly in the generated code or in the final platform environment, certain elements were not present in the application (images, CSS files, or programs that were not compiled successfully after the migration).
- Non-editable fields in the update page: because of problems in the migration of the KB, the generated programs did not allow editing of some attributes of entities in their update pages.
- Derived attributes could be edited: in some entities, attributes defined as derived from other tables, or as calculated, were also defined as editable. The generated test cases inserted data in these inputs but when the new tuples were inserted in the database the inserted values were substituted by the derived values. Therefore, the error was discovered when executing the complete lifecycle of the entity, after the creation of the read operation was executed, verifying the values inserted against the values stored in the database, identifying the difference and reporting the error.
- Entities invoked in an unexpected way: some objects were designed to be invoked in a specific way, and the generated test case executed them out of context (directly invoking the CRUD operations of the entity) causing an unexpected behavior because some session variables were not present.
- As a corollary, some test situations also helped to detect errors in the data model. For example, if the PK of an element is a numeric value (this is a common practice in database design) and the name of the entity, or the field used to select the foreign key from another entity is not required to be unique or not-null, it could be a problem, and the database design should control this.

As can be seen, it was possible to find errors, not only in the SUT, but also in the environment, and even in the code generator. Some were basic and evident errors, but most were not easily found by conventional testing.

The testers have an initial group of test cases that can be used to generate more complex cases by combining them.

7.2.2.4. EVALUATION OF THE CASE STUDIES

The main conclusion and achievement for the team is that the customer is satisfied. They were very thankful and evaluated the contributions of the test project positively, as to the stability of the system, and to the risk control before the release with a reduced cost for test execution than that planned.

Bancard now also has a test set that can continue to be used for regression testing for future releases, thus improving the obtained benefits for the same work.

The test generator was also tested and improved, verifying its scalability and applicability for big KBs.

7.2.2.5. LIMITATIONS

The generated test cases were completely executable on the SUT. The main limitation identified was that not all the tests are valid, resulting in false positives (a test reporting an error where it should not). This happens because not all the business rules are taken into account, which generates invalid test data and an oracle which tries to verify a positive execution. The oracle reports an error, but the error is in the test data.

The development team from Abstracta is still working on the implementation, prioritizing the most common business rules according to the case studies and sample applications under test. Basically, it is necessary to interpret the new business rules and consider them when the data is generated.

Some of the identified situations are:

- When selecting a random value for a primary key where the values universe is limited (for example *NUMERIC(2)*, has a total of one hundred possible values) it is probable that repeated values will be selected (already existent in the database).
- When the entity has more than one foreign key, and the valid values for one depends on the other. If this information is in the data model or business rules there is no problem, but sometimes this is managed in the application code, and it is more difficult to obtain this information.

Even though there are some limitations that force the tester to adjust or verify the errors reported by the tool, the result has been evaluated as very positive for the company.

7.2.3. GXTEST FOR PERFORMANCE TESTING

One of the main services provided by Abstracta is outsourced performance testing. Working on different projects the team realized that:

- The most common scenario to test is found in web environments
- Between 30% and 50% of the working hours are dedicated to test automation
- The automated test cases require a great deal of maintenance because they are very susceptible to changes in the application.

The main goal was therefore to take a new approach in order to reduce costs and obtain flexibility. For this, the proposed approach was implemented as an extension module of

GXtest. In this case, it was not necessary to show the relationship between the implementation in DBeST and GXtest because **it was directly implemented on GXtest.**

7.2.3.1. DESCRIPTION

GXtest executes Selenium and WatiN scripts, but it can easily be extended for more automated testing tools. During the execution of the functional test scripts, it captures the HTTP traffic between the browser and the SUT with an HTTP sniffer (a tool capable of capturing the network traffic) called Fiddler (fiddler2.com). With this information it builds a model that is used to generate the scripts for OpenSTA or JMeter. It is also easily extended to generate scripts for other load simulation tools.

The tester must open the automated functional test case. This is a GXtest model, and it can contain, for example, Selenium commands. The tester decides to generate an OpenSTA script or a JMeter script. GXtest then executes the functional test case, captures the traffic, and generates the performance testing script for the corresponding tool. The rest of the test is done in the load simulation tool as usual, but the most demanding task had already been automatically performed by GXtest.

7.2.3.2. EXPERIENCES IN INDUSTRY

At the moment of writing these lines, the presented tool has been used in five different projects by five different customers of Abstracta. One of the most common services provided by Abstracta is performance testing, and for this kind of project the methodology is that explained in Chapter 2. Below there is a description of each case study and then the summarized analysis.

- **Case Study 1: Logistics System:** This project was executed from Uruguay for a Chilean company, which is one of the biggest beer distributors in Latin America. They have to process more than 70 thousand invoices during the night, in order to allow the operators and truck drivers to prepare the delivery in time the following morning. The SUT was developed with GeneXus, which raises a special complication, because even small modifications to the development models (KB) could mean many modifications to the generated code and therefore to the HTTP traffic. The process was the same as in the other systems that were tested: first it was necessary to adjust the functional test scripts to regenerate the workload simulation scripts with our tool. It is in this kind of system, where the SUT suffers many modifications during the testing project, that our approach reports the best benefits, because it was necessary to regenerate the scripts several times, and this would have required a major effort if manually executed.

- **Case Study 2: Production Management System:** The peculiarity of this project was that there were no previous functional test scripts, so it was necessary to automate functional test scripts to use the tool. These functional test scripts were developed by a user (without knowledge of regression testing) which is almost impossible with any load generator. Once the project ended, the testing team started to manage a regression testing environment, using the scripts that were developed in the performance test project. In a way, the performance quality control favored the functionality quality control.
- **Case Study 3: Courts Management System:** The SUT, also developed with GeneXus, was destined for the Uruguayan government. The system was already in the production environment and had some identified issues. The goal was to study solutions to those problems in a controlled environment, and try to see how many users could be supported by the system with the current infrastructure.
- **Case Study 4: Auction System:** This system was developed by a Uruguayan company for a Chinese customer. It is important to highlight that only one script was required, which was defined based on statistical analysis of the normal use of the system, which revealed that 80% of the load is generated with only a few use cases. However, the combination of this test script with the test values from datapool leads to different execution flows in the SUT.
- **Case Study 5: Human Resources System:** This project was a migration from one version of GeneXus to another. The tool had important changes even in the code generator, thus making the migration very risky in terms of functionality and performance. Moreover, the generation platform also changed from a Windows, Client/server application, to a full web environment. The team already had regression tests, and our participation was in the performance testing. The system allows the users, among others, to mark the working times, check the salary and payments, share announcements, etc. It is therefore very important for the daily operations of the company and it could not have problems after the release.

7.2.3.3. EVALUATION OF THE CASE STUDIES

There were two testers working on all the projects, both with strong knowledge of GXtest, Selenium and OpenSTA. The SUTs were web systems from different domains, developed with different technologies, and very good results were obtained in all of them. Table 31 shows the number of generated scripts for each project, the number of simulated virtual users concurrently accessing the SUT, and the number of PCs required for the execution of the workload simulation.

TABLE 31 - USE OF THE TOOL IN PERFORMANCE TESTING PROJECTS

Project	SUT	# Scripts	# VU	# PCs
Production Management System	AS400 database, C# Web system on Internet Information Services	5	55	1
Courts Management System	Java Web system on Tomcat with Oracle database	5	144	1
Auction System	Java Web system on Tomcat with MySQL database	1	2000	4
Logistics System	Java Web system on Weblogic with Oracle database	9	117	1
Human Resources System	AS400 database, Java Web system on Websphere	14	317	1

The table also shows the number of PCs required for the execution of the workload simulation in order to highlight that, with our approach, it is possible to execute the simulation with a reduced test infrastructure.

Even considering the limitations, the time invested in the preparation of the scripts was reduced from an average of between 6 to 10 hours in the traditional approach (in our previous projects) to 1 to 5 hours with our tool in these five case studies. This implies an important cost saving for the automation phase. In addition, as it is easier to regenerate the scripts, it also gave us more flexibility for the maintenance of the scripts.

The different problems that these performance testing projects helped to prevent and identify can be summarized to include, among other things:

- Maximum concurrent connections: most of the time, when executing a certain number of concurrent users, it is typical to find a bottle neck in the number of the connections between the web server and the database server. This is only detected and optimized by simulating concurrent users.
- Required memory: it was possible to determine the required memory for the Java Virtual Machine of the web servers. This is a key factor in the performance of any web system running on a Java platform.
- Cache: by executing the expected load, and varying the input data executed by the virtual users, it is possible to detect whether the database cache is being used efficiently or if it can be optimized.
- Indexes: by using a test database with a similar volume to the production environment it was possible to detect the absence of certain indexes in very large tables for certain queries. With specific database analysis tools it is possible to identify those situations and optimize the indexes for a better performance in the database responses.
- SQL queries not optimized: simulating the real use of the system with concurrent access it is possible to see which are the most resource-demanding

SQL and optimize their execution plan. That could be done with the aid of specific tools.

These kinds of improvements are very expensive to make when the system is already released to the final users, and any issue may have a very negative impact. These projects therefore gave our customers a high return for their investment, avoiding many problems, and reducing risks before the release of their systems.

To summarize, the case studies have shown promising results in the performance testing, demonstrating that performance testing can be made in a more flexible way and with less effort, according to what the testers involved in the projects reported. These results are also aligned with those reported in the case study of [137].

7.2.3.4. LIMITATIONS

The main limitation identified when working with the tool was related to the wide range of uses of the HTTP protocol. The different systems tested have shown us that each new technology implies some adaptations to our templates to generate the workload simulation scripts (for example, it is not the same a simple PHP system than one that makes use of Ajax) mainly in the parameterization of the HTTP requests. However, once the template is adjusted for specific technology, it can be used for any system implemented with it.

Even though this technique has been used recently for mobile applications, it is only useful for HTTP traffic. It is necessary to make a big effort to extend it to other protocols, and there is no such a way to be independent of the protocol when working with workload generators.

At the moment the tool only helps with the generation of test case scripts, then the tester needs to prepare the workload combination in the corresponding load testing tool (which is generally easy if the scenario is defined) and validate the results (which is already automated with spreadsheets).

7.3. FINAL DISCUSSION

As explained in Chapter 1, one of the main goals was to obtain useful research results to be applied and adopted directly in Abstracta. This chapter showed how the research performed using standards, model driven transformations, etc., was adapted to the GeneXus environment in order to make this knowledge available to this specific community.

Although there is not much empirical validation of DBesTest, the results obtained were useful for delineating the research and development of the corresponding features in

GXtest. It was thus possible to obtain very successful results allowing the test team to obtain automatic test cases with low cost to test functional and performance aspects of different applications under test.

*Algunas cosas del pasado desaparecieron pero otras abren una brecha al futuro y son las
que quiero rescatar.*

— Mario Benedetti

*“Some things from the past disappeared, but others open a gap into the future and those are the ones I wish
to highlight”*

— Mario Benedetti

CHAPTER 8. CONCLUSIONS AND FUTURE RESEARCH LINES

This chapter presents the summary of our research and the analysis of the achievements of the proposed goals. It also shows the publications obtained and to conclude, it presents some future lines of research.

8.1. SUMMARY

This thesis proposed a methodology based on model-driven testing, in order to verify functional and non-functional aspects of Information Systems in an integrated way. The functional aspects are based on the Information System Model, initialized from the data model obtained from the database schema, and completed by the tester in a semi-automatic way. The non-functional properties are provided with a PMM model specifying the expected behavior under certain workload situations. From these artifacts, the thesis has proved that it is possible to generate a set of executable components to verify the requirements of the SUT.

Table 32 shows a summary of all the prototypes and tools developed for each contribution of this thesis. It also indicates which kind of SUTs are tested, the interface through which the SUT is stimulated, the type of testing (functional, performance, non-functional), the metamodels involved and the programming language in which it is implemented.

TABLE 32 - SUMMARY OF THE CONTRIBUTIONS AND THEIR IMPLEMENTATIONS

Criteria	DBesTest	PMM2UML-TP	GXtest Generator	GXtest for Performance
Kind of System Under Test	Web based application with databases	Web based application	GeneXus application	Web based application, GeneXus application
Interface to Access to the SUT	Graphical User Interface	Graphical User Interface	Graphical User Interface	http protocol
Test Level	Functional test	Non-functional test	Functional test	Performance test
Generated Artifacts	UML-TP model and test code with JUnit and Selenium Scripts	UML-TP model	GXtest Test Cases	Test code with OpenSTA or JMeter
Metamodels	UML, UDMP, GUIIMP, OCL, UML-TP	PMM, UML-TP	Proprietary (KB GeneXus and GXtest)	Proprietary (GXtest) and HTTP
Implementation of the Solution	ATL, Acceleo	ATL	C#	C#

Using the proposed approach it is possible to work on test specifications before the SUT is completely developed. When it is available, the user must provide the MIM (model-implementation mapping) in order to make the generated test cases completely executable.

In the rest of this section a complete analysis of the results is presented.

8.2. CONTRIBUTIONS

It is possible to differentiate between two groups of contributions: academic and industrial.

The academic focus was on model-based testing for information systems, mainly based on standards, specifically on UML. In that sense the main contributions presented in this thesis, were:

- A functional test model generated from a representation of the SUT, which is generated in a semi-automatic way using reverse engineering techniques.
- A PMM extension to model typical workloads in web systems. The usefulness of the metamodel in a new area of application was proved, modeling non-functional properties for information systems.
- A UML-TP extension with the ability to model workloads and non-functional validations such as global time restrictions and dependability metrics.
- A non-functional coverage criterion on PMM models, in order to generate a test model to verify all the properties defined.
- An algorithm for the generation of a test model integrating functional and non-functional requirements.

- A new approach for the generation of executable performance tests. This approach is faster and more flexible than the traditional one, basing the generation of simulation scripts on the functional automated test cases.
- An Eclipse plug-in integrating model-to-model and model-to-text executions to support the complete methodology in an integrated environment.

Considering the contributions from an Action-Research point of view, it should be mentioned all the new components of GXtest were implemented based on the thesis research:

- A test case generation algorithm based on the data model of GeneXus applications (the Knowledge Base). From a data model it is possible to completely generate automatically a set of test cases to test the CRUD operations of each entity.
- From the test cases modeled in GXtest it is possible to generate performance test cases to be executed with different load simulation tools and verify non-functional aspects of the applications developed with GeneXus.

8.3. ANALYSIS OF THE GOALS CONSECUATION

The goals of this thesis were presented in Chapter 1.

The first goal is to automate the test case design and execution for applications that make use of databases, in the context of web environments, but with the possibility of extending the approach to other types of applications, such as mobile.

This goal was achieved by:

- Studying the state of the art in test cases and test data generation for information systems. Depicting the gaps and unexplored research areas for this goal.
- Defining a model-driven testing methodology using UML to work with Information Systems. The final result is a set of executable test cases capable of giving a verdict.
- Identifying interesting test situations based on data model substructures, defining test cases to cover them.

Part of this analysis was presented in Chapter 2, 3, 4 and 5.

The second goal is to reduce the costs associated with performance testing, improving the automation process with more flexibility.

This goal was achieved by:

- Analyzing tools and methodologies for workload simulation
- Implementing a tool to automatically generate the performance test cases for different workload simulation tools, taking a set of functional test cases as input.

Part of this analysis was presented in Chapter 2 and Chapter 6.

In order to save costs related to execution time, and help testers to perform these tasks automatically, **the third goal is** to propose an integrated view, considering functional and non-functional properties verification.

This goal was achieved by:

- Studying the literature for model-driven proposals addressing non-functional aspects, and analyzing whether they were also applicable to functional testing or vice-versa.
- Extending PMM and UML-TP in order to form an integrated framework for testing the functional and non-functional properties of a system.
- Designing a coverage criterion and an algorithm to generate a test model from the requirements, considering in the whole process the specification and verification of functional and non-functional properties.

Part of this analysis was presented in Chapter 2 and Chapter 6.

From the beginning the focus was to obtain practical and industrially applicable results. The results obtained in DBesTest were all translated to GXtest and were tested in real projects with satisfactory and promising results. This was presented in Chapter 7.

In Chapter 1 a set of drawbacks was also identified. Table 33 summarizes them and shows the improvements in these directions that were part of the contribution of this thesis, which in turn were part of the original goals proposed.

TABLE 33 - DRAWBACKS AND THEIR IMPROVEMENTS

Identified drawback	Improvement and contribution in that direction
The tester requires knowledge in the database schema, and needs access to it.	DBesTest presents the information from the database with UML models. Chapter 4.
The tester requires knowledge in testing techniques.	DBesTest generates test cases automatically. The testing knowledge can be stored in test patterns, developed by a testing expert, and used any time. Chapter 5.
The tester requires knowledge in functional testing automation tools. Typically this is equivalent to a programming language.	DBesTest generates executable test cases, allowing the user to concentrate on the Test Model without necessarily having to cope with the code or automation tools. Chapter 5.
The tester requires knowledge in load simulation testing tools. Typically this includes not only knowing a programming or scripting language, but also knowledge about the communication protocol (at least HTTP in web systems).	From the previously generated functional automated test, it is possible to generate executable workload simulation test scripts to be used in a load simulator. Thus, the tester only has to cope with the test model. Chapter 6.
If the system is migrated to another platform, or if the test team decides to change the automation tool, etc., it is necessary to rebuild all the test artifacts from scratch.	With DBesTest it is necessary to change the model-to-text transformations in order to manage a different syntax. If this is done once, then it can be easily generated at any time from the test model to this new testing platform execution. Chapter 5.

As presented in Chapter 7, these aspects were also addressed with GXtest.

It is also necessary to consider that the expected contributions were:

- Reduce cost of functional testing for applications that make use of databases
- Reduce cost of performance testing for web applications that make use of databases

In the different studies in real projects the company considered that both goals have been reached successfully.

In this way, it is considered that this thesis fulfilled the various objectives and expected contributions.

8.4. PUBLICATIONS

Most of the results of the research performed in this thesis have been published in different peer-reviewed forums (see Table 34).

TABLE 34 - PUBLICATIONS

Title	Authors	Date	Type	Published in
Journals under revision				
Model-based test cases design integrating Functional and Non-Functional aspects.	Federico Toledo, Francesca Lonetti, Antonia Bertolino, Macario Polo Usaola, Beatriz Pérez Lamancha	To be sent in the first quarter of 2014	International Journal	To be sent to IET Software
A language for the automated addition of oracle to combinatorial test cases [138]	Macario Polo, Federico Toledo, Beatriz Pérez Lamancha	Sent December 2013	International Magazine	IEEE Software
Generación de Pruebas a Partir de Modelos de Datos en Entornos Generadores de Código [139]	Federico Toledo, Matías Reina, Fabián Baptista, Sebastián Grattarola, Beatriz Pérez Lamancha, Macario Polo	Sent May 7th 2013	Spanish American Journal	Sent to IEEE Latin America
National Journals (Spain)				
Utilización de MDE para la Prueba de Sistemas de Información Web [140]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	July-August 2013	Magazine	Novatica, Revista de la Asociación de Técnicos de Informática. Number 224, pp 33-39
Books Articles				
Automated Generation of Performance Test Cases from Functional Tests for Web Applications [141]	Federico Toledo, Matías Reina, Fabián Baptista, Beatriz Pérez Lamancha, Macario Polo	2013	Article in book	Evaluation of Novel Approaches to Software Engineering (Best papers book from ENASE conference)
International				
Extended UML Testing Profile for Improving Non-Functional Test Modeling. [142]	Federico Toledo, Francesca Lonetti, Antonia Bertolino, Macario Polo Usaola, Beatriz Pérez Lamancha	January 7 th /9 th 2014	Conference	2 nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14) Lisbon, Portugal ISBN: 978-989-758-007-9
Towards a Framework for Information System Testing – A model-driven approach [143]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	July 24 th /27 th 2012	Conference CORE B	7th International Conference on Software Paradigm Trends (ICSOPT). Rome, Italy. ISSN: 978-989-8565-19-8 pp. 172-177
From Functional Test Scripts to Performance Test Scripts for Web Systems [133]	Federico Toledo, Matías Reina, Fabián Baptista, Beatriz Pérez Lamancha, Macario Polo	July 5 th 2013	Workshop	The 1st International Workshop in Software Evolution and Modernization (SEM). Angers, France ISSN: 978-989-8565-66-2, pp. 12-20
Data Model Centered Test Case Design – Model-driven Information System Testing [144]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	November 18 th /23 rd 2012	Conference	The 4th International Conference on Advances in System Testing and Validation Lifecycles (VALID). Lisbon, Portugal. ISSN: 978-1-61208-233-2, pp. 127-132

Spanish American				
Metodología de Pruebas de Performance [106]	Gustavo Vázquez, Horacio López, Matías Reina, Federico Toledo, Simón de Uvarow, Edgardo Greising	November 10 th /15 th 2008	Conference CORE C	XX Encuentro Chileno de Computación - Jornadas Chilenas de Computación. Punta Arenas, Chile. ISBN: 978-956-319-507-1
Test case generation for information systems using reverse engineering techniques [145]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	June 20 th /23 rd 2012	Conference	7ª Conferencia Ibérica de Sistemas y Tecnologías de Información (CISTI). IEEE Xplore digital library. Madrid, Spain. ISSN: 978-989-96247-7-1, pp. 1-6
Metodología para Testing de Performance [146]	Gustavo Vázquez, Horacio López, Matías Reina, Federico Toledo, Simón de Uvarow, Edgardo Greising	November 12 th /14 th 2008	Conference	5ta Edición del SEPGLA (Software Engineering Process Group Latin America). Mar del Plata, Argentina.
National (Spain)				
Generación de Pruebas de Rendimiento a partir de Pruebas Funcionales para Sistemas Web [147]	Federico Toledo, Matías Reina, Fabián Baptista, Beatriz Pérez Lamancha, Macario Polo	September 18 th /20 th 2013	Conference	Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Madrid, Spain. ISBN: 978-84-695-8310-4
Enfoque dirigido por modelos para probar Sistemas de Información con Bases de Datos [148]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	September 17 th /19 th 2012	Conference	Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Almería, Spain. ISSN: 978-84-15487-28-9, pp. 315-328
Técnicas de pruebas basadas en modelos para procesos de negocio [149]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	September 5 th /7 th 2011	Conference	Jornadas de Ingeniería del Software y Bases de Datos (JISBD). La Coruña, Spain. ISSN: 978-84-9749-486-1 pp. 549-554
National (Uruguay)				
Metodología para Pruebas de Performance [150]	Gustavo Vázquez, Horacio López, Matías Reina, Federico Toledo, Simón de Uvarow, Edgardo Greising	December 2008	Technical Report	PEDECIBA (Programa de Desarrollo de las Ciencias Básicas), Facultad de Ingeniería, Universidad de la República. Montevideo, Uruguay. ISSN: 0797-6410-RT 08-20
National (Italy)				
Extending the Non-Functional Modeling of UML-TP [86]	Federico Toledo, Francesca Lonetti, Antonia Bertolino, Macario Polo Usaola, Beatriz Pérez Lamancha	December 2013	Technical Report	PuMa – Digital Library at CNR. ID: 2013-TR-040
Tutorials				
Tutorial de Pruebas de Rendimiento [151]	Federico Toledo, Beatriz Pérez Lamancha, Macario Polo	September 17 th /19 th 2012	Tutorial	Jornadas de Ingeniería del Software y Bases de Datos (JISBD). Almería, Spain.
Curso de pruebas de Rendimiento [152]	Federico Toledo	June 4 th /7 th 2012	Tutorial	8 th Jornadas de Calidad y Testing de Software, expoQA. Madrid, Spain.

Posters				
Generating test cases with the same abstraction level of 4th generation environments. [153]	Federico Toledo, Matías Reina, Fabián Baptista, Sebastián Grattarola.	October 22 nd /24 th 2013	Poster in conference.	User Conference on Advanced Automated Testing. http://ucaat.etsi.org/2013/ Paris, France

Table 35 shows the contribution of each publication to the research topics covered in the thesis.

TABLE 35 - RELATIONSHIP BETWEEN THE PUBLICATIONS AND THE RESEARCH SUBJECTS

Chapter	Main research topic	Specific research topic	Reference
3	Methodology	Functional test model generation	[140] [143] [144] [145] [148] [149]
		Integrated approach	Not sent yet.
4	Information System Metamodel	Reverse engineering	[140] [143] [144] [145] [148]
5	Functional Testing	Test model design	[140] [143] [144] [145] [148] [138]
		Test code generation	[140] [143] [144] [145] [148]
6	Performance Testing	Performance testing methodology	[106] [146] [150] [151] [152]
		Performance test modeling	[142] [86]
		Performance test model generation	Not sent yet.
7	Transfer to industry	Test cases generation (GXtest)	[139] [153]
		Performance tests generation	[141] [133] [147]

8.5. FUTURE WORK LINES

This thesis focused on developing and testing different approaches with the aim of making them available for the industry. Along these lines, some interesting topics are identified for future research.

MANDINGA in Practice

From now on the MANDINGA project will continue growing by experimentation on different industrial projects. In this way it will be possible to improve it and make it more usable. Our main goal for the tool is for it to become part of the developing methodology of any developer using GeneXus, helping them to improve the quality of their products.

UML-TP Request for Proposals

Apart from the extensions to UML-TP presented, there are more improvements required. More non-functional aspects such as security and usability need to be considered before its completion. On the other hand, the expressiveness of the workloads defined in the tests should be improved, and it should be possible to represent different expected values for different levels of workloads.

By continuing work with the team of CNR, and coordinating with the team responsible for UML-TP who showed us to be open to changes, the extension could be presented as an alternative to the RFP planned for next June, making these changes accessible to every user of the profile.

Monkey Testing

Abstracta began a new research project²⁴ on the area known as Monkey Testing (low cost programs which interacts automatically with the SUT with random inputs trying to make it crash). Literature distinguishes two types of monkeys [154]:

1. Dumb monkeys: without knowledge of the SUT and generating completely random interactions;
2. Smart monkeys: which have an understanding of the SUT through the use of models to guide the execution and compare the actual with expected behavior.

The goal of Abstracta is to build a web platform (based on Cloud services) in order to simplify the preparation of the test environment for final users, and provide them with different algorithms to test their systems. Each algorithm will ask for different inputs, the more the user gives the more the monkey can test. So, one of the ways visualized to continue the research on DBesTest and make it available for the testing community, is by integrating it as one of the modules of this new project.

In that way, the user will have to provide access to the database of the SUT, and then in a guided process will have to complete the Information System Model (UML and PMM models) in order to generate the test model and the executable test cases, which will also be executed from the cloud platform. In that way, the user will have a report for functional and non-functional requirements.

As a part of this research, the author of this thesis started to mentor a student on the master's degree at the University of Pereira, in Colombia, working on the subject of "*Smart monkey testing techniques for web and mobile systems*".

²⁴ The project was approved and a subvention was given by *Agencia Nacional de Investigación e Innovación* from the Uruguayan government. Reference: PPI_X_2013_1_10390 Smart Monkey Testing (www.anii.org.uy)

*"If debugging is the process of removing bugs, then programming must be the process of
putting them in"*

– Edsger W. Dijkstra

Annex 1. DIFFICULTIES FACED WITH MODEL-DRIVEN APPROACHES

This thesis used a model-driven approach, adopting standards as much as possible. These decisions were taken because of all the benefits of using standards, and because of all the proven advantages of model-driven approaches. This chapter shows that *it is not all a bed of roses*, presenting the different problems of the model-driven approaches based on our experience and on the current state of the tools and platforms that give support to model-driven approaches.

A1.1. INTRODUCTION AND MOTIVATION

Even though it is possible to start working with model-driven approaches, it is a long way from reaching utopia. Not only through the academy, but also through some companies, it can be seen that new efforts have been made to accomplish the dream, providing new tools and different approaches, and sharing experiences around MDE. However, not all the tested platforms were complete, or stable, and some lacked important aspects related to software engineering tasks, such as version control, modularization, debugging facilities, etc.

Often the reasons for these kinds of problems are well-known: tools are not completely compliant to the standards. This is why it is not always possible to work with different tools with the same models. It is necessary to prepare a model and then use a different tool to apply model-to-model transformations, and then it is necessary to visualize the resulting model, and even modify it, and finish the process using another tool to translate those models to code. It is necessary that vendors and tool developers work together and ensure that the complete tooling chain, IDEs and languages, work without losing information, in a transparent way, setting aside the compatibility problems, or more technical aspects.

On the other hand, it is also necessary to improve the usability of the different frameworks, to reduce the steep learning curve. This must be done not only for the

creation of diagrams, but also for the rest of the process, including testing, debugging, version management, etc.

During the thesis many problems were addressed. The most important are described below. The complications presented were determinants for the decision about the implementation of GXtest Generator. As it was determined that the current state of the technologies was not enough to build software, GXtest Generator was developed with C# and with a proprietary metamodel stored in the database.

A1.2. USING UML WITH MODEL TRANSFORMATIONS

As Marian Petre says in her article “UML in Practice” [155] *UML has been described by some as “the lingua franca of software engineering” but evidence from industry does not necessarily support such endorsements.*

When selecting UML and model-transformations languages there are some expectations, such as:

- Graphical notation should be easy to use by different members of a team, with and without programming skills.
- As they are standard, it is supposed that one can work with different tools, exporting and importing XMI files to share the information (completely, without any loss).
- A complete set of graphical editors for the different diagrams offered by UML.
- Ease of extensibility, allowing users to add more semantics to their models.
- Easy management of models to apply model-to-model and model-to-text transformations, without requiring programming skills.
- A transformation language is a programming language; therefore it provides the appropriate tools to manage the entire development cycle (source versioning, debugging, testing, etc.).

In our experience, none of these promises were fulfilled.

Technical difficulties for interoperability: Chapter 7 presented many problems regarding portability and interoperability, which were addressed. Even though one of the main advantages of using standards is that it should be easy to use different tools to work with the same format, language and notation, the situation seems to be the opposite. As already noted, it is not possible to manipulate a model with one tool and continue working with another without information loss.

Technical difficulties to initialize the graphical representation: Our approach generates different models and diagrams. It is expected (and necessary) that the UML modelers

are able to read these files and generate the graphical representation for the generated models. It was very hard to find one tool to accomplish this task. Rational Software Architect (RSA) was the only tool (of many) capable of generating a sequence diagram from the UML file. However, it is not completely direct. It was necessary to implement specific code in order to process the generated UML file and adapt it to the format expected by RSA.

Technical difficulties with UML Profiles: The standard extensibility mechanism of UML is not well-supported by all the tools. Reviewing forums of ATL, Acceleo, MOFscript, MediniQVT and other tools, it is easy to find dozens of messages with questions and reporting incidences in the use of UML Profiles. UML is mostly useful when its semantic is extended with domain specific concepts, defining stereotypes and tagged values, but after that, it is not easy to work with it. For example, it is common to face unexpected behaviors, exceptions, and more problems exporting and importing XMI files in different tools, etc. The same problem was identified in many UML modeling tools, such as Enterprise Architect.

Performance: What happens when the transformation takes so much time? It is not clear how a model transformation rule can be optimized because the processing is completely obscure and hidden.

Development Environment: The development environment has serious limitations. For example, a typical feature of modern environments for offering the accessible methods in a class, helping with autocompleting using the “CTRL” key plus the space bar, does not work. Instead, different methods are shown, even when they do not belong to the corresponding class or type. Debugging is impossible and exceptions are not very descriptive.

Complexity: UML is too complex. Also Petre remarked [155] that UML is considered “unnecessarily complex”. From our humble point of view, it is not only complex in its notation, but even more in its internal representation. For instance, representing a sequence diagram with two lifelines and one message from one to another requires defining at least the following elements:

- a Collaboration
- an Interaction
- two Lifelines
- one property for each Lifeline in the Interaction
- a Message
- a Message reply
- Parameters

- an Assembly Connector
- two Connector Ends nested in the Assembly Connector
- a Behavior Execution Specification
- four Message Occurrence Specification
- four Events.

This is the most simple sequence diagram. Figure 69 shows the tree representation on the left and the graphical representation on the right.

For someone trying to process a UML model with a model-transformation language, the meaning of each element and how to connect them properly in order to represent the intention should be clear.

If this diagram includes more elements of different kinds, it is easy to imagine that a transformation so as to build a diagram like this will be very complicated to design and develop. Another important aspect to considerate is the code maintenance.

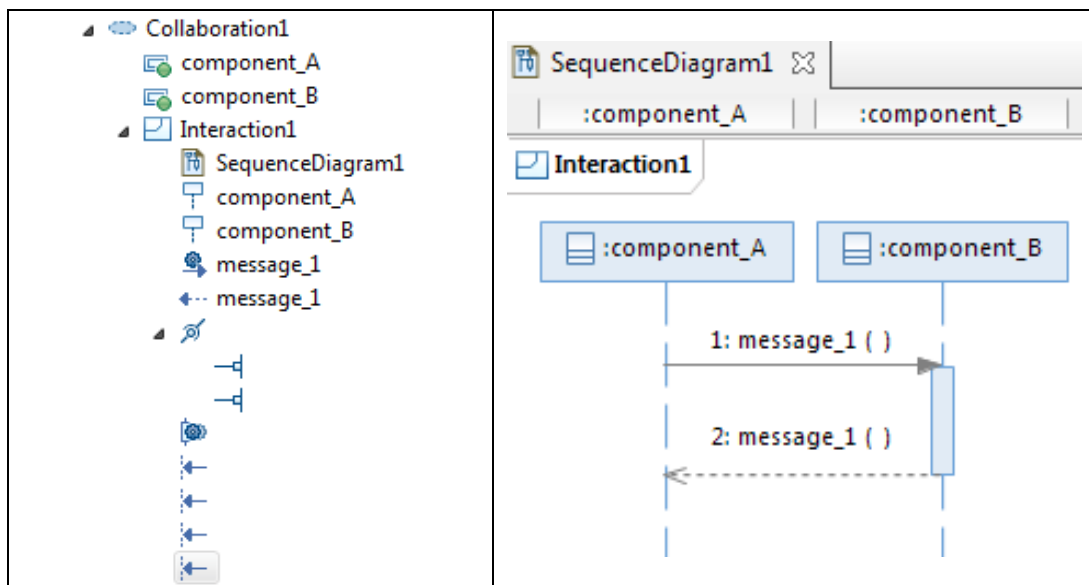


FIGURE 69 - EXAMPLE OF A SEQUENCE DIAGRAM

To show how much code is necessary to build a simple model like this one, Figure 70 shows an excerpt of an ATL transformation. The code is explained on the right side of each section.

<pre> -- @nsURI UML=http://www.eclipse.org/uml2/4.0.0/UML module example; create OUT : UML from IN : UML; lazy rule createSequenceDiagram { from ... to ... collaboration : UML!Collaboration(name <- 'Collaboration1', --package <- , ownedAttribute <- getlinea1, ownedAttribute <- getlinea2, ownedBehavior <- sequenceD), sequenceD : UML!Interaction(name <- 'Interaction1'), getlinea1 : UML!Property(name <- 'component_A', type <- getType1), getType1 : UML!Class(name <- 'component_A'), getlinea2 : UML!Property(name <- 'component_B', type <- getType2), getType2 : UML!Class(name <- 'component_B'), conector1 : UML!Connector(end <- conectorEnd1, end <- conectorEnd2), conectorEnd1 : UML!ConnectorEnd(role <- getlinea1), conectorEnd2 : UML!ConnectorEnd(role <- getlinea2), A_lifeline : UML!Lifeline(name <- 'component_A', interaction <- sequenceD), B_lifeline : UML!Lifeline(name <- 'component_B', interaction <- sequenceD), bes1 : UML!BehaviorExecutionSpecification(covered <- A_lifeline, start <- msg1Receive, finish <- msgReplySend), msg1Send : UML!MessageOccurrenceSpecification(covered <- A_lifeline, message <- msg1), msg1Receive : UML!MessageOccurrenceSpecification(covered <- A_lifeline, message <- msg1reply) } </pre>	<p>Reference to the UML metamodel.</p> <p>Header of the ATL module indicating that the input and the output are UML models.</p> <p><i>Collaboration</i> element with the <i>lifelines</i> as attributes (properties).</p> <p><i>Interaction</i> element.</p> <p>The abovementioned properties to relate the <i>Collaboration</i> to the <i>Lifelines</i>.</p> <p>The connector and the ends.</p> <p><i>Lifelines</i>.</p> <p><i>Behavior Execution Specification</i>. This is graphically represented as a box in the lifeline, to represent the life of an invocation.</p> <p>Four <i>message occurrence specifications</i>. They connect the messages with the life line in each extreme.</p>
--	---

<pre>), msgReplySend : UML!MessageOccurrenceSpecification(covered <- B_lifeline, message <- msg1reply), msgReplyReceive:UML!MessageOccurrenceSpecification(covered <- B_lifeline, message <- msg1), bes2 : UML!BehaviorExecutionSpecification(covered <- B_lifeline, start <- msg1Receive, finish <- msg2Ocurr), msg1: UML!Message(name <- 'message_1', receiveEvent <- msg1Receive, sendEvent <- msg1Send, connector <- conector1, signature <- operation --) msg1reply: UML!Message(name <- 'message_1', receiveEvent <- msgReplyReceive, sendEvent <- msgReplySend, connector <- conector1, messageSort <- #reply -- signature <- operation) do { sequenceD.lifeline <- A_lifeline; sequenceD.lifeline <- B_lifeline; A_lifeline.represents <- getlinea1; B_lifeline.represents <- getlinea2; sequenceD.ownedConnector <- conector1; sequenceD.ownedConnector <- conector2; sequenceD.fragment <- bes1; sequenceD.fragment <- bes2; sequenceD.message <- msg1; sequenceD.message <- msg1reply; sequenceD.nestedClassifier <- getType1; sequenceD.nestedClassifier <- getType2; } } </pre>	<p>The <i>Message</i> elements, first the invocation and then the response. It is connected with the corresponding <i>message occurrence specifications</i> and the <i>connector</i>. It should be linked to the represented operation.</p> <p>These lines in the “do” section were required to be there because (and it is difficult to find out why) if they are in the declarative section they just do not work as supposed. The main problem with this kind of things is that the debugger does not help, and these kinds of solutions are found by trying something without explanation.</p>
---	--

FIGURE 70 - ATL CODE TO GENERATE A SIMPLE SEQUENCE DIAGRAM

The code in the “do” section is imperative, and in the “form” and “to” sections is declarative. In the “do” section there is typically code for operations that are too complicated to be declarative, giving the language more power. This also indicates that the declarative approach is not considered the best even for the providers of the model-to-model transformation engine. Do not forget that the presented example corresponds to the simplest possible diagram, and it required two pages of code.

In addition, Table 36 shows some common transformations that from our point of view are not easy to understand, maintain and develop.

TABLE 36 - COMMON ATL TRANSFORMATIONS

Code	Explanation
<pre> operation:UML!Operation(name <- 'confirm', ownedParameter <- source.attribute- >iterate(p; r : Sequence(UML!Property)=Sequence{ if not p.type.oclIsTypeOf(UML!Class) and not p.hasStereotype('X') then r->including(thisModule.attr2Param(p)) else r endif)), lazy rule attr2Param { from source : UML!Property to dest : UML!Parameter (name <- source.name, type <- source.type, lower <- source.lower, upper <- source.upper ...) } </pre>	<p>A new <i>Operation</i> is created with certain parameters, according to the attributes in a class. For this there is an iteration that it is initialized empty and then the parameters are included only for those attributes that are not a type of “Class” and do not have the stereotype “X”. There is then a method to generate a <i>Parameter</i> for each <i>Attribute</i>.</p> <p>The code is very complex and not easy to understand.</p>
<pre> helper def : var1: UML!Lifeline = OclUndefined; helper def : counter: Integer = 0; </pre>	<p>Many times it is necessary to define something equivalent to <i>global variables</i> in order to share information between different rules that are executed in a non-deterministic order. This is considered bad practice in any programming language.</p>
<pre> helper def : getStereotype(name : String) : UML!Stereotype = UML!Stereotype.allInstances() -> select(p p.name = name)->first(); helper def : getVerdictType() : UML!Enumeration = UML!Enumeration.allInstances()-> select(p p.name = 'Verdict')->first(); </pre>	<p>It is necessary to use this kind of method in order to add specific semantics to the elements of the model. So, the types used in the transformation are given as strings.</p> <p>Something similar happens with “Verdict” which is a literal from an enumeration. This manipulation is not safe and is error prone.</p>
<pre> helper def : getCollectionFrom1to(i:Integer) : Collection(Integer) = if i>1 then thisModule.getCollectionFrom1to(i-1)- >including(i) else Sequence{1} endif; </pre>	<p>Some basic functionality like the creation of a collection of integers in a certain range is not provided by the engine. So, it was necessary to build this helper to return a collection of integers from one to the number given as a parameter.</p>

For us, these tasks are not easier than programming the code in Java or C#, storing the model in an XML file or even in a database model. Also, programming skills are required for this kind of task. Allowing the programmer to work in a popular and well-known environment is perhaps a better option.

Graphical Representation: One of the motivations for users selecting these approaches instead of starting from scratch is that there are modeling tools to edit the models graphically, and the semantic of UML models can be extended using UML Profiles. The

problem is that the time saved in these tasks is perhaps lost in dealing with the above issues.

Advantages of using UML and ATL over programming and modeling in a database:

- Graphical notation and representation (UML tools)

Advantages of programming and modeling in a database over UML and ATL:

- Well-known by programmers
- Better understanding of the code
- Better maintainability of the code
- Better control of integrity
- Easy to share the model with other tools (more implementation required, but it can work)

Some common problems: Below, some of the most common problems also addressed in the development of this thesis are listed.

- Modularized metamodel: if the metamodel is defined in different sections/modules (such as PMM) it is not possible to use ATL. Reported here <http://www.eclipse.org/forums/index.php/m/1100682/>
- It is not possible to load a UML and all its characteristics in a graphical model editor. Here are some of the places where it is reported:
 - <http://www.eclipse.org/forums/index.php/m/1064693/>
 - <http://www.eclipse.org/forums/index.php/m/1064257/>
- In certain circumstances executing a transformation from the Eclipse plug-in gives different results than executing it from Java code.
- Loading metamodels and profiles in Java code in order to execute a transformation using them does not always work properly.
- Problems using UML Profiles (specially, applying profiles). See Figure 71.

```
Java Stack:  
  
org.eclipse.m2m.atl.engine.emfvm.VMException: Exception during invocation of  
operation applyStereotype on org.eclipse.uml2.uml.internal.impl.ClassImpl@1bc1cae  
(name: Table, visibility: <unset>) (isLeaf: false, isAbstract: false) (isActive:  
false)  
...  
...
```

FIGURE 71 - ONE OF THE MOST COMMON EXCEPTIONS THROWN BY ATL USING UML PROFILES

A1.3. CONCLUSION

To summarize, the author of this thesis is now convinced that it is better to design a metamodel and its representation, and develop graphical notations and editors, than model-driven tendencies, at least, until the platform, tools and languages reach a better, and more mature state. The goal of allowing users without programming skills to prepare their transformations is still far away.

In the last twenty or thirty years, modeling in software development has been continuously evolving from non-formal graphic notations (Jackson, OMT, EER, UML), sometimes computable with author's metamodels, to the complete representations of almost any aspect of systems that the most recent versions of UML (and their extension possibilities) provide. However, the actual practice and use of MD tools is still a little tedious for the common developer: difficult configuration tasks, a continuing lack of technical documentation and user manuals, and difficulties in getting valid metamodels and problems for tool interoperability are the main obstacles for adopting this technology. However, hopefully soon all research and development efforts will be reflected in a better and more efficient way to develop software systems.

BIBLIOGRAPHY

- [1] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, and D. Roland, "Database reverse engineering: From requirements to CARE tools," *Autom. Softw. Eng.*, vol. 3, no. 1–2, pp. 9–45, 1996.
- [2] R. B. Grady, "Successfully applying software metrics," *Computer*, vol. 27, no. 9, pp. 18 – 25, Sep. 1994.
- [3] W. Suryn, A. Abran, and A. April, "ISO/IEC SQuaRE: The second generation of standards for software product quality," in *7th IASTED International Conference on Software Engineering and Applications*, 2003.
- [4] G. Myers, *The Art of Software Testing*. John Wiley & Son. Inc., 2004.
- [5] E. G. Kent Beck, "JUnit," 1997. [Online]. Available: <http://www.junit.org>.
- [6] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," 2007, pp. 31–36.
- [7] R. McTaggart, "Principles for participatory action research," *Adult Educ. Q.*, vol. 41, no. 3, pp. 168–187, 1991.
- [8] D. E. Avison, F. Lau, M. D. Myers, and P. A. Nielsen, "Action research," *Commun. ACM*, vol. 42, no. 1, pp. 94–97, 1999.
- [9] Y. Wadsworth, *What is participatory action research?* Action Research Issues Association, 1993.
- [10] OMG, "Model Driven Architecture (MDA)." [Online]. Available: <http://www.omg.org/mda/>.
- [11] OMG, "Unified Modeling Language," 1997. [Online]. Available: <http://www.uml.org/>.
- [12] OMG, "XMI." [Online]. Available: <http://www.omg.org/spec/XMI/>.
- [13] OMG, "Meta object facility (MOF)." [Online]. Available: <http://www.omg.org/mof/>.
- [14] J. Miller, J. Mukerji, and others, "MDA Guide Version 1.0. 1," *Object Manag. Group*, vol. 234, p. 51, 2003.
- [15] M. Utting and B. Legeard, *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.
- [16] OMG, "Meta Object Facility 2.0 Query/View/Transformation Specification," 2005.
- [17] OMG, "MOF Model to Text Transformation Language (MOFM2T), 1.0," 2008.
- [18] "MediniQVT." [Online]. Available: <http://projects.ikv.de/qvt>. [Accessed: 19-Jan-2013].
- [19] OBEO and AtlanMod, "ATL: ATLAS Transformation Language," 2012. [Online]. Available: <http://www.eclipse.org/atl/>.
- [20] "MOFScript." [Online]. Available: <http://www.eclipse.org/gmt/mofscript/>.
- [21] "Acceleo." [Online]. Available: <http://www.eclipse.org/acceleo/>. [Accessed: 15-Jan-2013].
- [22] OMG, "SysML specification," 2006. [Online]. Available: <http://www.omgsysml.org/>.
- [23] S. Friedenthal, A. Moore, and R. Steiner, *A practical guide to SysML: the systems modeling language*. Access Online via Elsevier, 2011.

- [24] T. Weillkiens, *Systems engineering with SysML/UML: modeling, analysis, design*. Morgan Kaufmann, 2011.
- [25] “ab/05-12-02 - Information Management Metamodel (IMM) RFP.” OMG, 2005.
- [26] A. De Lucia, C. Gravino, R. Oliveto, and G. Tortora, “An experimental comparison of ER and UML class diagrams for data modelling,” *Empir. Softw. Eng.*, vol. 15, pp. 455–492, 2010.
- [27] G. Bavota, C. Gravino, R. Oliveto, A. D. Lucia, G. Tortora, M. Genero, and J. A. Cruz-Lemus, “Identifying the Weaknesses of UML Class Diagrams during Data Model Comprehension,” presented at the MoDELS, 2011, pp. 168–182.
- [28] E. Marcos, B. Vela, and J. M. Caverio, “A methodological approach for object-relational database design using UML,” *Softw. Syst. Model.*, vol. 2, pp. 59–72, 2003.
- [29] S. W. Ambler, “A UML profile for data modeling,” 2009.
- [30] M. Polo, I. García-Rodríguez, and M. Piattini, “An MDA-based approach for database re-engineering,” *J. Softw. Maint. Evol. Res. Pract.*, vol. 19, pp. 383–417, 2007.
- [31] G. Sparks, “Database modeling in UML,” *Methods & Tools*, pp. 10–22, 2001.
- [32] D. Gornik, “UML Data Modeling Profile,” IBM, Rational Software, 2002.
- [33] T. Isakowitz, A. Kamis, and M. Koufaris, “Extending the capabilities of RMM: Russian Dolls and Hypertext,” in *System Sciences, 1997, Proceedings of the Thirtieth Hawaii International Conference on*, 1997, vol. 6, pp. 177–186.
- [34] J. Gómez, C. Cachero, and O. Pastor, “Conceptual modeling of device-independent web applications,” *Multimed. IEEE*, vol. 8, no. 2, pp. 26–39, 2001.
- [35] D. Schwabe, R. Mattos Guimaraes, and G. Rossi, “Cohesive design of personalized web applications,” *Internet Comput. IEEE*, vol. 6, no. 2, pp. 34–43, 2002.
- [36] S. Ceri, P. Fraternali, and A. Bongio, “Web Modeling Language (WebML): a modeling language for designing Web sites,” *Comput. Netw.*, vol. 33, no. 1, pp. 137–157, 2000.
- [37] J. Conallen, *Building Web applications with UML*. Addison-Wesley Longman Publishing Co., Inc., 2002.
- [38] OMG, “Object Constraint Language,” 2000. [Online]. Available: <http://www.omg.org/spec/OCL/2.0/>.
- [39] T. Halpin, “Augmenting UML with Fact-orientation,” 2001, p. 10 pp.
- [40] S. Shah, K. Anastasakis, and B. Bordbar, “From UML to Alloy and back again,” *Models Softw. Eng.*, pp. 158–171, 2010.
- [41] S. Khurshid and D. Marinov, “TestEra: Specification-based testing of Java programs using SAT,” *Autom. Softw. Eng.*, vol. 11, no. 4, pp. 403–434, 2004.
- [42] S. A. Khalek, B. Elkarablieh, Y. O. Laleye, and S. Khurshid, “Query-aware test generation using a relational constraint solver,” 2008, pp. 238–247.
- [43] D. Jackson, I. Schechter, and I. Shlyakhter, “ALCOA: The Alloy constraint analyzer,” 2000, pp. 730–733.
- [44] J. M. Spivey, *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., 1992.
- [45] OMG, “MARTE Specification,” 2004. [Online]. Available: <http://www.omgmarTE.org/>.

- [46] S. Gilmore, L. Gönczy, N. Koch, P. Mayer, M. Tribastone, and D. Varró, “Non-functional properties in the model-driven development of service-oriented systems,” *Softw. Syst. Model.*, vol. 10, no. 3, pp. 287–311, 2011.
- [47] A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta, “GLIMPSE: a generic and flexible monitoring infrastructure,” in *Proceedings of the 13th European Workshop on Dependable Computing*, 2011, pp. 73–78.
- [48] A. Di Marco, C. Pompilio, A. Bertolino, A. Calabrò, F. Lonetti, and A. Sabetta, “Yet another meta-model to specify non-functional properties,” in *Proceedings of the International Workshop on Quality Assurance for Service-Based Applications*, 2011, pp. 9–16.
- [49] A. Bertolino, A. Calabrò, F. Lonetti, A. Di Marco, and A. Sabetta, “Towards a model-driven infrastructure for runtime monitoring,” in *Software Engineering for Resilient Systems*, Springer, 2011, pp. 130–144.
- [50] A. Abran and P. Bourque, *SWEBOK: Guide to the software engineering Body of Knowledge*. IEEE Computer Society, 2004.
- [51] IEEE, “IEEE Standard Glossary of Software Engineering Terminology,” 1990.
- [52] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams, *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., 2007.
- [53] OMG, “UML 2.0 Testing Profile Specification,” 2004. [Online]. Available: <http://utp.omg.org/>.
- [54] S. Bukhari and T. Waheed, “Model driven transformation between design models to system test models using UML: a survey,” in *Proceedings of the 2010 National Software Engineering Conference*, 2010, p. 8.
- [55] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock, “An introduction to the testing and test control notation (TTCN-3),” *Comput. Netw.*, vol. 42, no. 3, pp. 375–403, 2003.
- [56] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe, “On the Design of the New Testing Language TTCN-3,” in *TestCom*, 2000, pp. 161–176.
- [57] I. Schieferdecker, Z. R. Dai, J. Grabowski, and A. Rennoch, “The UML 2.0 testing profile and its relation to TTCN-3,” in *Testing of Communicating Systems*, Springer, 2003, pp. 79–94.
- [58] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, “From U2TP models to executable tests with TTCN-3—an approach to model driven testing,” in *Testing of Communicating Systems*, Springer, 2005, pp. 289–303.
- [59] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea, *Performance testing guidance for web applications: patterns & practices*. Microsoft Press, 2007.
- [60] B. C. Kitchenham, “Guidelines for performing systematic literature reviews in software engineering,” *Sch. Softw. Eng. Group*, vol. 2, p. 1051, 2007.
- [61] L. Zhu, N. B. Bui, Y. Liu, and I. Gorton, “MDABench: Customized benchmark generation using {MDA},” *J. Syst. Softw.*, vol. 80, no. 2, pp. 265 – 282, 2007.
- [62] L. Zhu, Y. Liu, I. Gorton, and N. B. Bui, “MDABench: a tool for customized benchmark generation using MDA,” in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, 2005, pp. 171–172.

- [63] Y. Cai, J. Grundy, and J. Hosking, “Experiences Integrating and Scaling a Performance Test Bed Generator with an Open Source CASE Tool,” presented at the ASE, 2004, pp. 36–45.
- [64] A. Bertolino, G. De Angelis, L. Frantzen, and A. Polini, “Model-based generation of testbeds for web services,” in *Testing of Software and Communicating Systems*, Springer, 2008, pp. 266–282.
- [65] Z. Liu, N. Niclausse, and C. Jalpa-Villanueva, “Traffic model and performance evaluation of web servers,” *Perform. Eval.*, vol. 46, no. 2, pp. 77–100, 2001.
- [66] F. M. de Oliveira, R. da S. Menna, H. V. Vieira, and D. Ruiz, “Performance testing from UML models with resource descriptions,” in *1st Brazilian Workshop on Systematic and Automated Software Testing*, 2007.
- [67] M. da Silveira, E. Rodrigues, A. Zorzo, L. Costa, H. Vieira, and F. Oliveira, “Generation of Scripts for Performance Testing Based on UML Models,” in *SEKE*, 2011, pp. 258–263.
- [68] HP Mercury, “LoadRunner,” 2001. [Online]. Available: <http://www8.hp.com/us/en/software-solutions/software.html?compURI=1175451>.
- [69] V. Garousi, L. C. Briand, and Y. Labiche, “Traffic-aware stress testing of distributed systems based on UML models,” in *ICSE*, New York, NY, USA, 2006, pp. 391–400.
- [70] D. Krishnamurthy, J. A. Rolia, and S. Majumdar, “A synthetic workload generation technique for stress testing session-based systems,” *Softw. Eng. IEEE Trans. On*, vol. 32, no. 11, pp. 868–882, 2006.
- [71] HP, “httperf,” 2005. [Online]. Available: <http://www.hpl.hp.com/research/linux/httperf/>.
- [72] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, “MBPeT: A Model-Based Performance Testing Tool,” in *VALID 2012, The Fourth International Conference on Advances in System Testing and Validation Lifecycle*, 2012, pp. 1–8.
- [73] F. Abbors, T. Ahmad, D. Truscan, and I. Porres, “Model-based performance testing in the cloud using the mbpet tool,” in *Proceedings of the ACM/SPEC international conference on International conference on performance engineering*, 2013, pp. 423–424.
- [74] T. Ahmad, F. Abbors, D. Truscan, and I. Porres, *Model-Based Performance Testing Using the MBPeT Tool*. Turku Centre for Computer Science, 2013.
- [75] X. Guo, X. Qiu, Y. Chen, and F. Tang, “Design and implementation of performance testing model for web services,” in *Informatics in Control, Automation and Robotics (CAR), 2010 2nd International Asia Conference on*, 2010, vol. 1, pp. 353–356.
- [76] B. A. Pozin and I. V. Galakhov, “Models in performance testing,” *Program. Comput. Softw.*, vol. 37, no. 1, pp. 15–25, 2011.
- [77] A. J. Bennett and A. J. Field, “Performance engineering with the UML profile for schedulability, performance and time: a case study,” in *Modeling, Analysis, and Simulation of Computer and Telecommunications Systems, 2004.(MASCOTS 2004). Proceedings. The IEEE Computer Society’s 12th Annual International Symposium on*, 2004, pp. 67–75.
- [78] C. U. Smith and L. G. Williams, “Performance engineering evaluation of object-oriented systems with SPE· ED TM,” in *Computer Performance Evaluation Modelling Techniques and Tools*, Springer, 1997, pp. 135–154.

- [79] C. U. Smith, C. M. Lladó, and R. Puigjaner, “Performance Model Interchange Format (PMIF 2): A comprehensive approach to Queueing Network Model interoperability,” *Perform. Eval.*, vol. 67, no. 7, pp. 548–568, 2010.
- [80] C. U. Smith and C. M. Llado, “Performance model interchange format (PMIF 2.0): XML definition and implementation,” in *Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings. First International Conference on the*, 2004, pp. 38–47.
- [81] C. U. Smith and L. G. Williams, “A performance model interchange format,” *J. Syst. Softw.*, vol. 49, no. 1, pp. 63–80, 1999.
- [82] S. Balsamo, A. Di Marco, P. Inverardi, and M. Simeoni, “Model-based performance prediction in software development: A survey,” *Softw. Eng. IEEE Trans. On*, vol. 30, no. 5, pp. 295–310, 2004.
- [83] H. Koziolok, “Performance evaluation of component-based software systems: A survey,” *Perform. Eval.*, vol. 67, no. 8, pp. 634–658, 2010.
- [84] W. Afzal, R. Torkar, and R. Feldt, “A systematic review of search-based testing for non-functional system properties,” *Inf. Softw. Technol.*, vol. 51, no. 6, pp. 957–976, 2009.
- [85] P. McMinn, “Search based software test data generation: a survey,” *Softw. Test. Verification Reliab.*, vol. 14, pp. 105–156, 2004.
- [86] F. Toledo, F. Lonetti, A. Bertolino, M. Polo Usaola, and B. Pérez Lamanca, “Extending the Non-Functional Modeling of UML-TP,” Pisa, Italy, Dec. 2013.
- [87] Apache, “JMeter,” 2001. [Online]. Available: <http://jmeter.apache.org/>. [Accessed: 06-Mar-2013].
- [88] K. H. Davis and P. H. Aiken, “Data Reverse Engineering: A Historical Survey,” in *Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE’00)*, Washington, DC, USA, 2000, p. 70–.
- [89] G. CanforaHarman and M. Di Penta, “New frontiers of reverse engineering,” in *2007 Future of Software Engineering*, 2007, pp. 326–341.
- [90] E. J. Chikofsky and J. H. Cross, “Reverse engineering and design recovery: A taxonomy,” *Softw. IEEE*, vol. 7, no. 1, pp. 13–17, 1990.
- [91] A. Memon, I. Banerjee, and A. Nagarajan, “GUI ripping: Reverse engineering of graphical user interfaces for testing,” in *Proceedings of the 10th Working Conference on Reverse Engineering*, 2003, pp. 260–269.
- [92] C. Bellettini, A. Marchetto, and A. Trentini, “WebUml: reverse engineering of web applications,” in *Proceedings of the 2004 ACM symposium on Applied computing*, 2004, pp. 1662–1669.
- [93] F. Ricca and P. Tonella, “Building a tool for the analysis and testing of web applications: Problems and solutions,” in *Tools and Algorithms for the Construction and Analysis of Systems*, Springer, 2001, pp. 373–388.
- [94] D. C. Kung, C.-H. Liu, and P. Hsia, “An object-oriented web test model for testing web applications,” in *Quality Software, 2000. Proceedings. First Asia-Pacific Conference on*, 2000, pp. 111–120.
- [95] G. A. Di Lucca, A. R. Fasolino, F. Faralli, and U. De Carlini, “Testing web applications,” in *Software Maintenance, 2002. Proceedings. International Conference on*, 2002, pp. 310–319.

- [96] I. Jacobson, G. Booch, and J. E. Rumbaugh, *The unified software development process—the complete guide to the unified process from the original designers*. Addison-Wesley, 1999.
- [97] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. E. Williams, “The UML 2.0 testing profile,” 2004, pp. 181–189.
- [98] R. S. Pressman, *Software Engineering: A Practitioner’s Approach*, 7/e. 2009.
- [99] M. Fewster and D. Graham, *Software test automation: effective use of test execution tools*. ACM Press/Addison-Wesley Publishing Co., 1999.
- [100] M. Polo, S. Tendero, and M. Piattini, “Integrating techniques and tools for testing automation,” *Softw. Test. Verification Reliab.*, vol. 17, pp. 3–39, 2007.
- [101] C. Poole, “Nunit,” 2002. [Online]. Available: <http://www.nunit.org>. [Accessed: 01-Jan-2003].
- [102] J. Huggins, “Selenium,” 2004. [Online]. Available: <http://seleniumhq.org/>. [Accessed: 06-Mar-2013].
- [103] T. Koomen, L. van der Aalst, B. Broekman, and M. Vroon, *TMap Next, for result-driven testing*. UTN Publishers, 2006.
- [104] M. Grindal, J. Offutt, and S. F. Andler, “Combination testing strategies: A survey,” *Softw. Test. Verification Reliab.*, vol. 15, pp. 167–199, 2005.
- [105] M. Polo Usaola and B. Pérez Lamancha, “A framework and a web implementation for combinatorial testing.”
- [106] G. Vázquez, M. Reina, F. Toledo, S. de Uvarow, E. Greisin, and H. López, “Metodología de Pruebas de Performance,” presented at the JCC, 2008.
- [107] J. Offutt, S. Liu, A. Abdurazik, and P. Ammann, “Generating test data from state-based specifications,” *Softw. Test. Verification Reliab.*, vol. 13, pp. 25–53, 2003.
- [108] Cornett, “Code Coverage Analysis,” 2004. [Online]. Available: www.bullseye.com/coverage.html. [Accessed: 01-Jan-2012].
- [109] D. Cohen, I. C. Society, S. R. Dalal, M. L. Fredman, and G. C. Patton, “The AETG System: An Approach to Testing Based on Combinatorial Design,” *IEEE Trans. Softw. Eng.*, vol. 23, pp. 437–444, 1997.
- [110] Y. Lei and K. C. Tai, “In-parameter-order: a test generation strategy for pairwise testing,” in *High-Assurance Systems Engineering Symposium, 1998. Proceedings. Third IEEE International*, 1998, pp. 254–261.
- [111] A. Andrews, R. France, S. Ghosh, and G. Craig, “Test adequacy criteria for UML design models,” *Softw. Test. Verification Reliab.*, vol. 13, pp. 95–127, 2003.
- [112] J. Tuya, M. J. Suárez-Cabal, and C. De La Riva, “Full predicate coverage for testing SQL database queries,” *Softw. Test. Verification Reliab.*, vol. 20, pp. 237–288, 2010.
- [113] K. Haller, “White-box testing for database-driven applications: A requirements analysis,” 2009, p. 13.
- [114] M. Emmi, R. Majumdar, and K. Sen, “Dynamic test input generation for database applications,” presented at the ISSTA’07: Software Testing and Analysis, 2007, pp. 151–162.
- [115] A. Arasu, R. Kaushik, and J. Li, “Data generation using declarative constraints,” presented at the International conference on Management of data, 2011, pp. 685–696.

- [116] D. Chays and Y. Deng, “Demonstration of AGENDA tool set for testing relational database applications,” 2003, pp. 802–803.
- [117] A. Neufeld, G. Moerkotte, and P. C. Loekemann, “Generating consistent test data: Restricting the search space by a generator formula,” *VLDB J.*, vol. 2, pp. 173–213, 1993.
- [118] E. Song, S. Yin, and I. Ray, “Using UML to model relational database operations,” *Comput. Stand. Interfaces*, vol. 29, pp. 343–354, 2007.
- [119] E. J. Naiburg and R. A. Maksimchuck, *UML for database design*. Addison-Wesley Professional, 2001.
- [120] K. Zieliński and T. Szmuc, *Software engineering: evolution and emerging technologies*, vol. 130. IOS Press, 2005.
- [121] “UML SDK.” [Online]. Available: <http://www.eclipse.org/modeling/mdt/?project=uml2>.
- [122] J. F. Terwilliger, L. M. Delcambre, and J. Logan, “Querying through a user interface,” *Data Knowl. Eng.*, vol. 63, no. 3, pp. 774–794, 2007.
- [123] P. Santos-Neto, R. Resende, and C. Pádua, “Requirements for information systems model-based testing,” in *Proceedings of the 2007 ACM symposium on Applied computing*, 2007, pp. 1409–1415.
- [124] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, “Hints on test data selection: Help for the practicing programmer,” *Computer*, vol. 11, pp. 34–41, 1978.
- [125] A. Bertolino, “Software Testing Research: Achievements, Challenges, Dreams,” presented at the Future of Software Engineering, 2007. FOSE '07, 23, pp. 85–103.
- [126] G. M. Kapfhammer and M. L. Soffa, “A family of test adequacy criteria for database-driven applications,” presented at the ESEC/FSE-11: European Software Engineering Conference, 2003, vol. 28, pp. 98–107.
- [127] B. Pérez Lamancha, P. Reales Mateo, M. Polo Usaola, and D. Caivano, “Model-driven Testing - Transformations from Test Models to Test Code,” presented at the ENASE, conf/enase/LamanchaMPC11, 2011, pp. 121–130.
- [128] D. Xu, “A tool for automated test code generation from high-level Petri nets,” *Appl. Theory Petri Nets*, pp. 308–317, 2011.
- [129] M. J. Suárez-Cabal, C. De La Riva, and J. Tuya, “Populating test databases for testing SQL queries,” *IEEE Lat. Am. Trans.*, vol. 8, pp. 164–171, 2010.
- [130] S. A. Khalek and S. Khurshid, “Systematic testing of database engines using a relational constraint solver,” 2011, pp. 50–59.
- [131] C. Tadros and L. Wiese, *Using SAT-solvers to compute inference-proof database instances*, vol. 5939 LNCS. 2010.
- [132] D. Graham and M. Fewster, *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison-Wesley Professional, 2012.
- [133] F. Toledo, M. Reina, F. Baptista, M. Polo Usaola, and B. Pérez Lamancha, “From Functional Test Scripts to Performance Test Scripts for Web Systems,” in *SEM 2013*, Angers, France, 2013, pp. 12–20.
- [134] “Papyrus,” 2010. [Online]. Available: <http://www.eclipse.org/papyrus/>.
- [135] IBM, “Rational Software Architect,” 1990. [Online]. Available: <http://www-03.ibm.com/software/products/es/es/ratisoftarch/>.

- [136] F. Toledo, M. Reina, F. Baptista, and S. Grattarola, "GXtest online user manual." [Online]. Available: <http://gxtest.abstracta.com.uy>.
- [137] I. de S. Santos, A. R. Santos, and P. de A. dos S. Neto, "Reusing Functional Testing in order to Decrease Performance and Stress Testing Costs," in *SEKE*, 2011, pp. 470–474.
- [138] M. Polo Usaola, F. Toledo, and B. Pérez Lamancha, "A language for the automated addition of oracle to combinatorial test cases," *IEEE Software (sent)*.
- [139] F. Toledo, M. Reina, F. Baptista, S. Grattarola, M. Polo Usaola, and B. Pérez Lamancha, "Generación de Pruebas a Partir de Modelos de Datos en Entornos Generadores de Código," *submitted*.
- [140] F. Toledo, B. Pérez Lamancha, and M. Polo Usaola, "Utilización de MDE para la Prueba de Sistemas de Información Web," *Novatica Rev. Asoc. Téc. Informática*, vol. 224, pp. 33–39, Jul. 2013.
- [141] F. Toledo, M. Reina, F. Baptista, B. Pérez Lamancha, and M. Polo Usaola, "Automated Generation of Performance Test Cases from Functional Tests for Web Applications," in *Evaluation of Novel Approaches to Software Engineering*, Springer-Verlag, 2013.
- [142] F. Toledo, F. Lonetti, A. Bertolino, M. Polo Usaola, and B. Pérez Lamancha, "Extended UML Testing Profile for Improving Non-Functional Test Modeling," presented at the 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD'14), Lisbon, Portugal, 2014.
- [143] F. Toledo, B. Pérez Lamancha, and M. Polo Usaola, "Towards a Framework for Information System Testing - A model-driven testing approach," presented at the ICSOFT, 2012.
- [144] F. Toledo, B. Pérez Lamancha, and M. Polo Usaola, "Data model centered test case desing. A model-driven approach," presented at the VALID, 2012.
- [145] F. Toledo, B. Pérez Lamancha, and M. Polo Usaola, "Test case generation for information systems using reverse engineering techniques," presented at the Information Systems and Technologies (CISTI), 2012 7th Iberian Conference on, 20, pp. 354–359.
- [146] G. Vázquez, M. Reina, F. Toledo, S. de Uvarow, E. Greisin, and H. López, "Metodología de Testing de Performance," presented at the 5ta Edición del SEPGLA (Software Engineering Process Group Latin America), Mar del Plata, Argentina, 2008.
- [147] F. Toledo, M. Reina, F. Baptista, M. Polo Usaola, and B. Pérez Lamancha, "Generación de Pruebas de Rendimiento a partir de Pruebas Funcionales para Sistemas Web," presented at the Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Madrid, Spain, 2013.
- [148] F. Toledo, M. Polo Usaola, and B. Pérez Lamancha, "Enfoque dirigido por modelos para probar Sistemas de Información con Bases de Datos," presented at the Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Almería, Spain, 2012, pp. 315–328.
- [149] F. Toledo, B. Pérez Lamancha, and M. Polo Usaola, "Técnicas de prueba basadas en modelos para Procesos de Negocio," 2011.
- [150] F. Toledo, M. Reina, S. de Uvarow, H. López, G. Vazquez, and E. Greisin, "Metodología para Pruebas de Performance (Reporte Técnico)," Universidad de la República, Montevideo, Uruguay, 0797–6410-RT 08-20.
- [151] F. Toledo, M. Polo Usaola, and B. Pérez Lamancha, "Tutorial de Pruebas de Rendimiento," presented at the Jornadas de Ingeniería del Software y Bases de Datos (JISBD), Almería, Spain, 17-Sep-2012.

- [152] F. Toledo, “Curso de Pruebas de Rendimiento,” presented at the 8 th Jornadas de Calidad y Testing de Software, expoQA., Madrid, Spain, 04-Jun-2012.
- [153] F. Toledo, M. Reina, and F. Baptista, “Generating test cases with the same abstraction level of 4th generation environments,” presented at the User Conference on Advanced Automated Testing, Paris, France, 22-Oct-2013.
- [154] S. Bauersfeld and T. E. Vos, “Advanced Monkey Testing for Real-World Applications.”
- [155] M. Petre, “UML in practice,” in *Proceedings of the 2013 International Conference on Software Engineering*, 2013, pp. 722–731.