Towards a Framework for Information System Testing
A Model-driven Testing Approach

Federico Toledo Rodríguez[1], Beatriz Pérez Lamancha[2] and Macario Polo Usaola[3]
[1]*Abstracta, Montevideo, Uruguay*
[2]*Centro de Ensayos de Software, Universidad de la República, Montevideo, Uruguay*
[3]*Alarcos Research Group, University of Castilla-La Mancha, Ciudad Real, Spain*

Keywords: Software Testing, Model-based Testing, Information System Testing, Test Data.

Abstract: Testing has an important role within the development of a software system; the automation of testing tasks has been largely used with the goal of minimizing costs and increasing productivity. For some of those tasks –as it is the execution of test cases– well-known solutions already exist as the industry adopted them many years ago. This is not the case with test data generation, and even less for software that uses databases, where this task is particularly complex. In the present work we propose to generate test cases automatically to test information systems that use relational databases. We represent the data model with UML Data Modeling Profile, automatically extracted from the database with reverse engineering techniques. Then, applying model-driven testing, test cases are generated from the data model, represented with the standard UML Testing Profile. The test case generation also includes test data inputs, in order to accomplish certain coverage criteria defined on the schema.

1 INTRODUCTION

An Information System (IS) is a software system that allows the manipulation of structured data for a specific business goal. The importance of testing in the IS development process has being growing lately, looking for better quality, but it is still one of the most time consuming tasks. Commonly, IS's consist of applications which deal with the information saved in relational databases (DB), storing data of different entities on the base of particular business rules. Thus, there is a correspondence between the visual components (e.g. web forms), the data structures (generally in relational DB) 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 operation of type *update* on the DB.

Considering this, the DB is one of the essential components for an IS. To manage it, the IS could provide several applications with different nature (according to the type of user, operating system, environment, kind of device, etc.) and there is often a correspondence between the DB structure and the logic layer of those applications accessing it. In fact, there are many proposals which intend to apply reverse engineering on the DB to obtain the corresponding data model: this is sometimes used to restructure the DB itself, and sometimes as a business layer model to generate new applications for the same DB (Alalfi et al., 2008, García Rodríguez de Guzmán, 2007, Pérez-Castillo et al., 2012).

Therefore, taking into account that the DB is the main common element between those applications, we can use its structure as a starting point for the construction of test cases which permit to guarantee the quality of the IS accessing it.

Model-Driven Testing (MDT) refers to model-based testing where the test cases are automatically generated from software artifacts through model transformation. In this work we propose a model-driven testing methodology to automatically generate test cases from the DB metadata.
The goal is to minimize the development effort of the required tools for the proposal, using (as much as possible) metamodels according to the standards offered by the *ObjectManagment Group (OMG)*, for which there are tools already available in the market.

Toledo Rodríguez F., Perez B. and Polo Usaola M..
Towards a Framework for Information System Testing - A Model-driven Testing Approach.
DOI: 10.5220/0004067801720177
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 172-177
ISBN: 978-989-8565-19-8

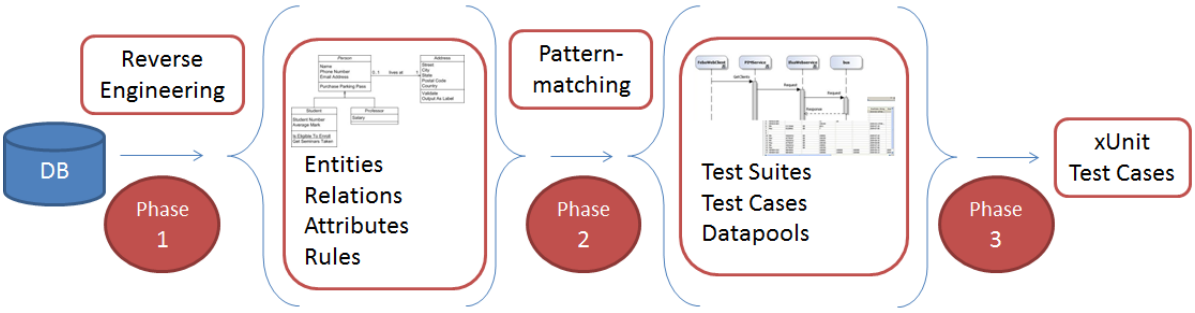

Figure 1: General Schema.

The methodology consists mainly in three phases described below and shown in Figure 1:

- **Phase 1: Reverse Engineering to Data Model Extraction.** Initially some reverse engineering techniques and tools are used in order to obtain, from the physical schema of the DB, its corresponding data model.
- **Phase 2: Model-driven Test Cases Generation.** The data model is processed using pattern-matching (García-Rodríguez et al., 2007), then the test cases that test each pattern are automatically generated through model transformations. As a result, test cases for the data structures are generated, thus obtaining a test model.
- **Phase 3: Executable Test Cases Generation.** Last but not least, the test models are transformed into test code, obtaining executable test cases.

Except for the reading of the physical DB scheme, models and transformations comply with the standards of the *OMG*.

In the following section we present some background concepts. In section 3, the general framework is explained. Then, in section 4, we show the related work. Finally, in section 5, we explain our conclusions and future work.

2 BACKGROUND

This section presents the background of this paper.

2.1 Metamodels

Our metamodels are UML Profiles, which is the standard mechanism that UML offers for its extension, based on the use of stereotypes and tagged values.

The UML Data Modeling Profile (**UDMP)** (Gornik, 2002) is an UML class diagram extension developed by IBM to design DB using UML, with the expressive power of an entity-relationship model, and it is used in their tool *IBM Rational Rose Data Modeler*. It defines concepts at a physical level and architecture (*Node, Tablespace, Database,* etc.), and the ones required for the DB design (*Table, Column,* etc.). Several proposals use this profile to model the DB structure (Yin and Ray, 2005; Zielinski and Szmuc, 2005; Sparks, 2001).

The **UML Testing Profile** (UTP) extends UML with test specific concepts for testing, grouping them in test architecture, test data, test behavior and test time. The *test case* is the main concept and its behavior can be described by sequence diagrams, state machines or activity diagrams. In this profile, the test case is an operation of a test context that specifies how a set of components cooperates with the system under test to achieve the test goal, and giving a verdict. Being a profile, the UTP seamlessly integrates into UML.

The OMG has adopted and published the transformation language between models, called **QVT** (*query/view/ transformation*) (OMG, 2005), which is defined at a metamodel level. Using QVT, it is possible to throw queries against models, and of course, to perform model transformations within models.

Besides, the OMG published a model-to-text transformation language, called **MOFM2T** (OMG, 2008). Its goal is to define a language to facilitate the generation of code or documentation from models.

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 which have quickly reached high popularity (Polo et al., 2007b). 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 perhaps are Junit

(www.junit.org) for Java and Nunit (www.nunit.org) for Microsoft .NET.

2.2 Coverage Criteria

The coverage criteria are 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, achieving a predefined coverage without finding new errors could be used as a stop testing criteria (Cornett, 2004). Given that in our case we generate test cases from a UDMP class diagram corresponding to the DB's data model, we will consider some coverage criteria as adequate to those artifacts:

2.2.1 Coverage Criteria for Class Diagrams

Andrews et al. proposed different coverage criteria for testing UML diagrams (Andrews et al., 2003). For class diagrams, they propose the following:

- *AEM (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 which 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)*.
- *GN (Generatlization)*: the test set must cover every generalization relation of the model.
- *CA (Class attribute)*: the test set must instantiate representative data sets for the different attributes of each class.

2.2.2 Coverage Criteria for CRUD

The data instances on a system have a life cycle, since they are created until they are deleted, and they are updated in between. Because of that, it is useful to consider CRUD operations as coverage criteria.

This criterion (Koomen et al., 2006) considers that the whole life cycle of an entity should be tested, therefore it defines the test cases starting with a *C*, followed by a *R*, followed by every operation that performs a *U* (with a *R* after that to validate the result) and finally a *D* and another *R* (to validate the deletion). Then, representing with *Ui* each operation that updates data (over different attributes for example), the criteria could be represented with the following regular expression:

$$C \cdot R \cdot (Ui \cdot Ri)^* \cdot D \cdot R \quad (1)$$

3 FRAMEWORK FOR TESTCASE GENERATION

In this section we describe the details of the proposed framework, going through the different phases, pointing to which metamodels are used, what is going to be generated and how. *Figure 2* complements the already presented **Error! Reference source not found.**, showing the metamodels involved in the process.

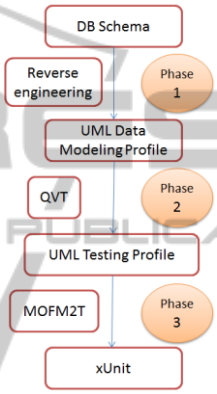

Figure 2: Metamodels used for the test case generation.

Nowadays, we are developing the proposal to its experimentation and validation. Bellow, the implementation details also are described.

3.1 Phase 1: Data Model Extraction

From the DB of the IS under test we extract a data model which represents the entities and their relationships, attributes, and constraints. This model is based on the UDMP metamodel. This approach allows representing the information necessary to generate test cases in a platform independent way.

This is the only step for which there is not a standard tool available. We are extending *RelationalWeb* (Polo et al., 2007a), a reengineering tool developed in ALARCOS research group, in order to make it generate models according to the UDMP metamodel.

3.2 Phase 2: Test Cases Generation

We want to generate test cases for every occurrence of certain structures in the data model. This can be made by using model transformation, defining

patterns which indicate which structures to look for in the data model in order to generate test cases from them. The patterns are expressed as QVT rules which explore the data model looking for occurrences of the defined substructure. The target metamodel in the transformation is UTP: for each occurrence matched by the QVT rule, the transformation will generate different elements of the UTP.

We can define, for example, a pattern to match every relation of two entities from 1 to 0:N, indicating to generate test cases in order to cover the *AEM criterion* for the *create* operation of those entities. The matching rule is easy to represent and it can be seen as a generic model; this is shown in *Figure 3*. The QVT rule will look for this kind of substructure in the data model to apply the corresponding transformations.

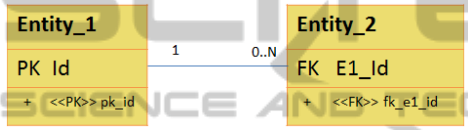

Figure 3: Example of pattern model.

First of all, the transformation will create the test architecture (according to UTP), as shown in *Figure 4*, which includes, among other components:

- a *Test Context*, which contains the generated *test cases* as methods;
- a *Test Component*, responsible for initiating the test cases and interacting with the *SUT (system under test)*;
- one *datapool* per each entity; each *datapool* has one *data selector* for each *test case* in order to provide specific data for each test.

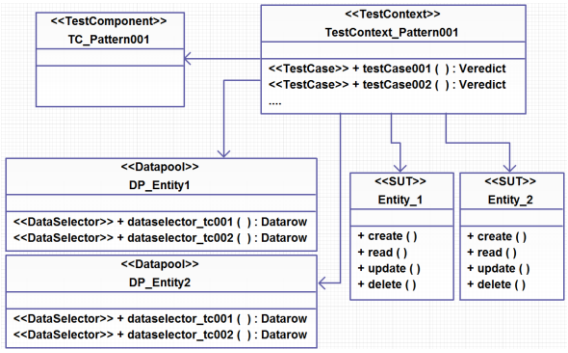

Figure 4: Example of generated test architecture.

Then, each test case behavior is represented as a sequence diagram as the one represented in *Figure 5*. Note that for the same matching rule several test cases can be generated, in order to reach certain coverage. For this example, with this test case, we reach *AEM criterion* for the *create* operation of the *Entity_1* and *Entity_2*. Considering the relationship 1 to 0:N, we should test these cases:

- *case 1*: 1 at the left extreme,
- *case 2*: 0 at the right extreme,
- *case 3*: 1 at the right extreme,
- *case 4*: multiple instances at the right extreme (there is no a limit, so we consider this situation covered with at least 2).

Furthermore, we should even test 0 at the left extreme (*case 5*), in order to verify if the system can manage the unexpected situation (it should fail because of the foreign key).

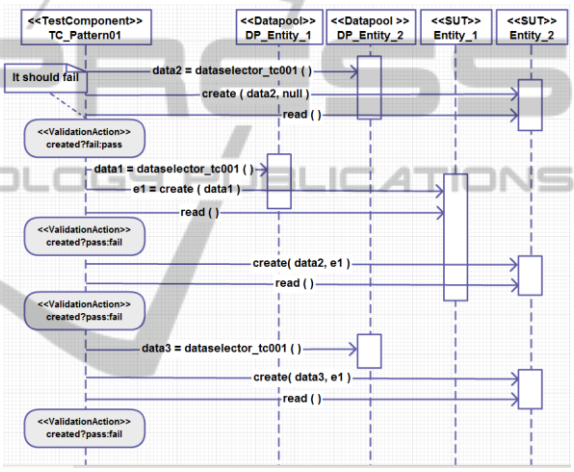

Figure 5: Example of a generated test case.

In *Figure 5* we can see that these situations are covered in the different calls:

- the first call tries to create an instance of *Entity_2* without an instance of *Entity_1*, which covers *case 5* (it should fail);
- then the test case creates an instance of *Entity_1* (*case 1* and *case 2*)
- then tries to create an instance of *Entity_2* associated to one of *Entity_1* (*case 3*),
- and two different instances of *Entity_2* associated with one of *Entity_1* (*case 4*).

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

Moreover, we generate *datapools* and *data selector* methods for each test case. In the testing model we indicate if the data is valid or invalid, and the test case works considering this meta-information, which is useful for the oracle, to define

if it has to verify failure or success. For example, if *Entity_2* has three columns, one for the foreign key (*E1_Id*) and two integer values (*A* and *B*), the generated *data selector* could return data as it is shown in *Table 1*. So, test cases do not have specific data, they are tagged as *valid* or *invalid*, and in the following phase, taking this categorization and the data types from the data model, they are instantiated with corresponding values.

Table 1: Example datapool.

A	B	E1_Id	Expected result
Any	Any	Null	Fail
Valid	Invalid	Valid	Fail
Valid	Valid	Valid	Pass
…	…	…	…

3.3 Phase 3: Executable Test Cases Generation

Finally, from the generated test cases in the UTP, we obtain test code. To transform those models into executable code, we continue the work presented in (Pérez Lamancha et al., 2011), which uses UTP test cases automatically generated from UML sequence diagrams taken from the design of the SUT. The UTP test cases are transformed into JUnit and Nunit code using MOFM2T.

Instead, in our proposal, we do not have the specification of the operations that are being tested (mainly the CRUD operations). In (Pérez Lamancha et al., 2011) the operations under test are given by the sequence diagrams. As we do not ask for this specification as input, the test cases that we generate have invocations to stub methods, belonging to an adaptation layer that has to be developed later on.

One of the main benefits of this part of the proposal is that we can follow a *Keyword-driven testing* approach (Fewster and Graham, 1999), to have the possibility to test different levels of our system, or different components that manage the same data model (i.e., a web component and another for mobile). For example, we could use the generated xUnit test cases for unit testing (performing the invocations at a persistency level, accessing the classes that manage the access to the DB), or for integration testing (exercising the classes of the logic layer of the system), or even for system testing (invocating automated scripts at an user interface level, for example using Selenium, seleniumhq.org).

This idea can be better understood paying attention to the example pseudo code shown in *Figure 6*. This is part of the result of the last phase of the generation process, considering the same example of *Figure 5*. To make this xUnit completely executable, the user has to develop some operations for each entity; those are the CRUD operations (in the example the *create* operation is used for *Entity_1* and *Entity_2*) and methods to validate absence or existence of an instance (verifying the values of each attribute). The same test can be executed with different adaptation layers, one for unit testing, another for system testing, invoking Selenium scripts, etc.

```
@Test
voidtest_001() throws Exception{
  …
  data2 = dpE2.dataSelectorTest001();
  //create instance with datapool
  //data and without Entity_1
  al_E2.create(data2,null);
  //verify that it was NOT inserted
  al_E2.verifyNotExists(data2,null);
  data1 = dpE1.dataSelectorTest001();
  e1 = al_E1.create(data1);
  al_E2.create(data2, e1);
  al_E2.verifyExists(data2, e1);
  …
}
```

Figure 6: Pseudocode of the generated xUnit.

In this way, we are obtaining a set of test cases that can be executed against any IS managing the data model from which we generated the test cases, independently of the different platforms over which those IS were developed.

4 RELATED WORKS

Regarding test data generation, (Tuya et al., 2010) define a coverage criteria based on SQL queries, applying a criteria similar to MCDC (Chilenski and Miller, 1994) but considering the conditions of FROM, WHERE and JOIN sentences, generating test data to cover this criterion. There is an approach where the code coverage criteria are extended in order to consider the embedded SQL sentences (Haller, 2009, Emmi et al., 2007), generating DB instances to cover the different scenarios proposed as interesting. (Arasu et al., 2011) propose to specify in some way the expected results of each SQL included in the test, and then they can generate test data to satisfy this specification. The proposal from (Chays and Deng, 2003), called AGENDA, takes as input the DB schema and categorized test data given by the user, whereby generates test cases and initial

DB states, and validating after the test case execution the outputs and the final DB state. (Neufeld et al., 1993) generate DB states according to the integrity restrictions of the relational schema, using a constraint solver. As far as we know, many proposals for test data generation exist, but none of them focuses on automated test model generation using model transformations.

5 CONCLUSIONS

In this article a novel approach to test IS with DB was presented, with focus in the coverage of the structures found by test patterns in the data model. This test generation methodology takes into account the fact that, in this kind of systems, one of the most important things is the correctness of the data, which implies testing the operations over the data structures. As the framework is almost completely based on standards, it can be adopted with almost any UML-compliant tool. Therefore no tools are needed to be developed to support the methodology.

This is the first step towards the construction of a test generation environment specifically for IS that uses DB, which will facilitate the empirical validation of the proposed ideas.

ACKNOWLEDGEMENTS

This work has been partially funded by *ANII*, Uruguay, and by *DIMITRI* (TRA2009_0131) and *MAGO/Pegaso* (TIN2009-13718-C0201) Spanish projects.

REFERENCES

Alalfi, M. H., Cordy, J. R. & Dean, T. R. 2008. SQL2XMI: Reverse Engineering of UML-ER Diagrams from Relational Database Schemas. *Working Conference on Reverse Engineering.* IEEE Computer Society.

Andrews, A., France, R., Ghosh, S. & Craig, G. 2003. Test adequacy criteria for UML design models. *Software Testing, Verification and Reliability,* 13**,** 95-127.

Arasu, A., Kaushik, R. & Li, J. 2011. Data generation using declarative constraints. *International conference on Management of data.* ACM.

Cornett. 2004. *Code Coverage Analysis* [Online]. Available: www.bullseye.com/coverage.html [Accessed 2012].

Chays, D. & Deng, Y. 2003. Demonstration of AGENDA tool set for testing relational database applications. IEEE Computer Society.

Chilenski, J. J. & Miller, S. P. 1994. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal,* 9**,** 193-200.

Emmi, M., Majumdar, R. & Sen, K. 2007. Dynamic test input generation for database applications. *ISSTA'07: Software Testing and Analysis.*

Fewster, M. & Graham, D. 1999. *Software test automation: effective use of test execution tools*, ACM Press/Addison-Wesley Publishing Co.

García-Rodríguez, I., Polo, M. & Piattini, M. 2007. Using Model-Driven Pattern Matching to derive functionalities in Models. *SEKE - Software Engineering and Knowledge Engineering*

García Rodríguez De Guzmán, I. 2007. *Pressweb: un proceso para la reingeniería de sistemas heredados hacia servicios web.* UCLM.

Gornik, D. 2002. UML Data Modeling Profile. IBM, Rational Software.

Haller, K. 2009. White-box testing for database-driven applications: A requirements analysis. ACM.

Koomen, T., Van Der Aalst, L., Broekman, B. & Vroon, M. 2006. *TMap Next, for result-driven testing*, UTN Publishers.

Neufeld, A., Moerkotte, G. & Loekemann, P. C. 1993. Generating consistent test data: Restricting the search space by a generator formula. *The VLDB Journal,* 2**,** 173-213.

Omg 2005. Meta Object Facility 2.0 Query/View/Transformation Specification.

Omg 2008. MOF Model to Text Transformation Language (MOFM2T), 1.0.

Pérez-Castillo, R., García-Rodríguez De Guzmán, I., Caballero, I. & Piattini, M. 2012. Software Modernization by Recovering Web Services from Legacy Databases. *Journal of Software: Evolution and Process***,** In Press.

Pérez Lamancha, B., Mateo, P. R., Polo Usaola, M. & Caivano, D. 2011. Model-driven Testing - Transformations from Test Models to Test Code. *ENASE.* SciTePress.

Polo, M., García-Rodríguez, I. & Piattini, M. 2007a. An MDA-based approach for database re-engineering. *Journal of Software Maintenance and Evolution: Research and Practice,* 19**,** 383-417.

Polo, M., Tendero, S. & Piattini, M. 2007b. Integrating techniques and tools for testing automation. *Software Testing Verification and Reliability,* 17**,** 3-39.

Sparks, G. 2001. Database modeling in UML. *Methods & Tools.*

Tuya, J., Suárez-Cabal, M. J. & De La Riva, C. 2010. Full predicate coverage for testing SQL database queries. *Software Testing Verification and Reliability,* 20**,** 237-288.

Yin, S. & Ray, I. 2005. Relational database operations modeling with UML. *AINA'05: Advanced Information Networking and Applications.*

Zielinski, K. & Szmuc, T. 2005. Data Modeling with UML 2.0. *Frontiers in Artificial Intelligence and Applications***,** 63.