A Test Model for Graph Database Applications: An MDA-Based Approach

Raquel Blanco
Department of Computing
University of Oviedo
Gijón, Spain
rblanco@uniovi.es

Javier Tuya
Department of Computing
University of Oviedo
Gijón, Spain
tuya@uniovi.es

ABSTRACT

NoSQL databases have given rise to new testing challenges due to the fact that they use data models and access modes to the data that differ from the relational databases. Testing relational database applications has attracted the interest of many researchers; but this is still not the case with NoSQL database applications. The approach presented in this paper defines a test model for graph database applications that takes into account the data model of this technology and the system specification. To automate the derivation of the test cases and the evaluation of their adequacy we propose a framework that places model-based testing into the model-driven architecture context.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification

Keywords

Graph database testing, model-based testing, MDA, specification-based testing.

1. INTRODUCTION

Databases are probably the most important asset of an organization, and constitute the core of most software systems. Nowadays, many organizations need to store a vast amount of information, and they are increasingly turning to NoSQL databases to manipulate this large amount of data with higher performance [21].

There are numerous NoSQL technologies (currently 150) [28], which are classified into four popular types according to their data model [26]: key-value, document-based, column-family and graph databases. These types of database have something in common: they do not require a schema that restricts the data that can be stored.

Testing NoSQL database applications is a crucial and a challenging process for several reasons. On the one hand, NoSQL technologies do not work with SQL and each one uses its own APIs and tailored query languages, which are not as widely known as SQL by the developers. Moreover, the programming of complex queries can be difficult [21]. In particular, queries of graph database technologies can be especially verbose and difficult to write, understand and maintain [2]. Due to these difficulties, faults can appear in the code that accesses the database.

On the other hand, despite the fact that NoSQL databases do not require a schema, the applications usually have an underlying conceptual model that represents the data stored (henceforth *conceptual data model*). As there are no constraints that restrict their storage, the physical database could contain data that do not satisfy the conceptual data model. These data can produce application malfunctions and/or incorrect outputs to the user.

To test database applications, many approaches have been developed, such as [7], [9], [13], [15], [24]. However, as these works rely on SQL statements and/or the existence of an explicit database schema, they cannot be applied to testing NoSQL database applications. So, it is necessary to develop new testing approaches for this type of applications, which take into account the new data models and specific characteristics of each NoSQL technology.

The scope of this paper is the development of an approach to test graph database applications that considers the data model characteristics of this technology. Data are stored in nodes and relationships among nodes, and both nodes and relationships can contain properties. The graph databases are gaining in popularity and thousands of organizations use them in applications such as social recommendations, logistics, fraud detection, identity and access management, etc. [27]. To achieve this goal, we propose a model-based testing approach in the context of model-driven architecture, so that we benefit from the support of automation of both paradigms.

The main contributions of this work are:

- The definition of a framework that integrates model-based testing (MBT) into the model-driven architecture (MDA) paradigm.
- The definition of a test model for graph database applications that relies on both the underlying conceptual data model and the system specification.

The remainder of this paper is organized as follow: Section 2 presents the related work. Sections 3 and 4 describe the architecture of our MBT/MDA framework and the test model, respectively. Section 5 presents a case study. The paper ends with conclusions and future work.

A-TEST'15, August 30-31, 2015, Bergamo, Italy
© 2015 ACM. 978-1-4503-3813-4/15/08...
http://dx.doi.org/10.1145/2804322.2804324

2. RELATED WORK

2.1 Testing Database Applications

Several approaches in the literature address the problem of testing database applications. To guide the generation of test inputs and evaluate their adequacy, several criteria have been developed. Works that define program-based adequacy criteria range from criteria for procedural code that take into account the SQL queries [10], to criteria specially designed to deal with the SQL statements [14], [15], [32], [33], [35] and tools to automate the criteria [16], [31], [39]. Other works define specification-based adequacy criteria, such as [4]. The generation of test inputs has been addressed in several works: [3], [23], [36] generate test databases and [7], [24] both test database and program inputs.

With regard to testing the database schema, works are focused on defining adequacy criteria [25], [37], generating data to test the schema constraints [17] or prioritizing the test cases when the database schema changes [12], [13].

As stated before, these works depend on SQL code and/or explicit relational database schemas, while our approach is totally independent. The closest works to ours are those of [17], [37] as they use the database schema to generate test cases. These works are focused on testing the database schema. However, our approach uses a conceptual data model as the basis for designing the test model according to the system specification.

2.2 Model-Based Testing and Model-Driven Architecture

Model-based testing (MBT) has been used in several database testing works, such as [4], [9], [11], [18]. In MBT, the system is modelled to identify the important aspect to be tested regarding the expected system behaviour, obtaining a test model. Next, a test selection criterion is chosen to derive the abstract test cases, which are then concretized by means of a test generation technology and translated into executable test cases that can be run against the software under test (SUT) [34].

On the other hand, MBT can be placed into the MDA context, obtaining the abstraction levels PIT (Platform Independent Test) and PST (Platform Specific Test) [8]. The PIT level contains the test models that are platform independent, whereas at the PST level the test models contain information about the specific underlying platform.

Works in the MBT/MDA context are mainly focused on transforming the system model at the PIM level into the test model at PIT level [1], [5], [6], [19], [22], and defining transformations from the PIT level to the PST level and/or the test code [1], [20], [22], [38]. However, it is important to have some independence between the system models and the test models, because mistakes in the system models can be propagated to the code and the tests and, therefore, they are impossible to detect [30], [34]. In our approach the test model is designed by the testers, instead of being generated from a system model.

3. THE MBT/MBA FRAMEWORK

The architecture of the MBT/MDA framework we propose is depicted in Figure 1. At the PIT level, we have identified two important viewpoints: PITM (*Platform Independent Test Model*) and PITGM (*Platform Independent Test Generation Model*).

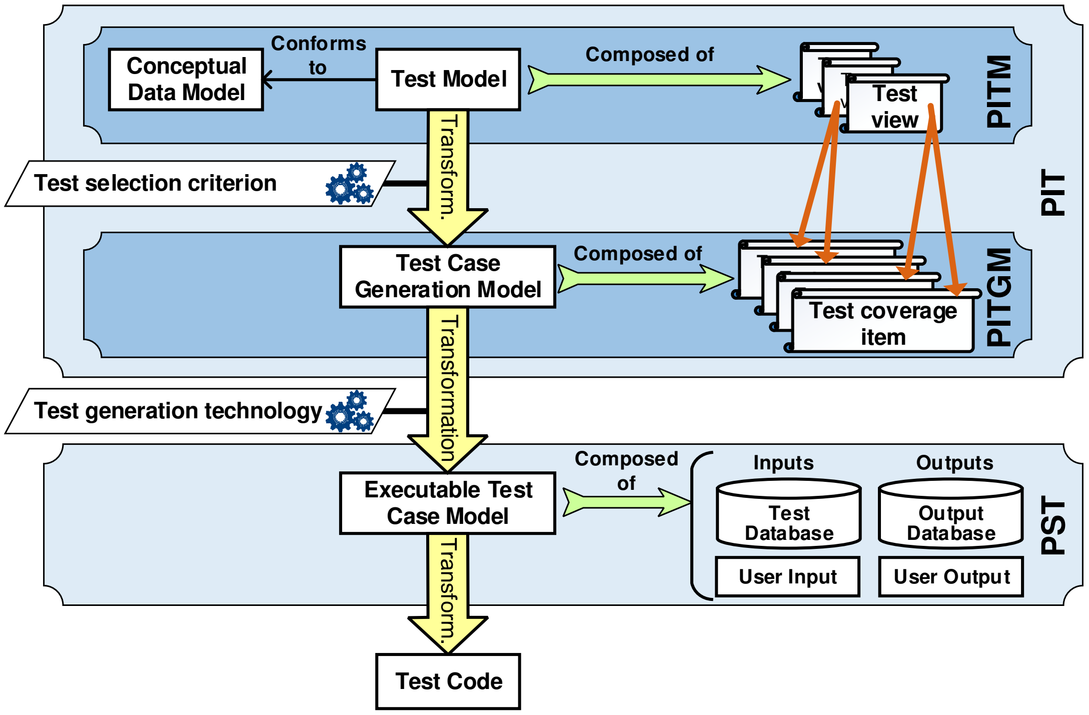


Figure 1. Architecture of the MBT/MDA framework

The PITM level is focused on the definition of the testing objectives, according to the system specification of the SUT. Here, the test model is designed as a composition of one or more important features of the SUT to be tested, called *test views*. Related to the scope of testing database applications, the conceptual data model of the database plays an important role, so the test model must conform to it in order to specify the test views correctly. If the conceptual data model is not explicitly stated (the NoSQL databases are schema-optional), the tester prepares this model as part of the testing process.

The PITGM level is centred on the definition of the test case generation model that is formed by the specific items that must be tested, which are called *test coverage items*. In the context of MBT, the test generation model represents a model of the abstract test cases. The mapping between the test model and the test generation model is performed by transformations that are guided by the test selection criterion chosen, which leads the test coverage items. In this mapping, a test view can give rise to several test coverage items. From a PITM, several PITGM can be automatically derived by appropriate transformations.

The PST level contains the executable test case model, which is obtained by means of transformations from the test case generation model and depends on the specific graph database management system used. These transformations are guided by some test generation technology that concretizes the test inputs, formed by the state of the database before the execution of the test case (test database) and the values supplied by the user (user input); and the expected outputs, formed by the state of the database after the execution of the test case (output database) and the values shown to the user (user output). Finally, the executable test cases can be transformed into an executable test code.

An important benefit of the MDA paradigm consists in reaching a high level of automation by defining transformations among models. In our framework, the tester specifies the test model and, after that, the processes of deriving the test case generation model, the executable test cases and the test code can be carried out automatically.

The elaboration of a test database with meaningful data is a determining factor, as these data are transformed to produce the test output and the test database has to represent the situations of interest to be tested, so the SUT can exercise them. This paper is focused on the definition of test views for unit testing, which are specially tailored for managing the database of graph database applications.

4. TEST VIEWS FOR GRAPH DATABASE APPLICATIONS

Consider, for example, a database application ("illness risk") which determines the level of risk of suffering an illness according to different factors such as the severity of previous episodes suffered by the person (which is classified in three levels), the existence of previous episodes of the illness in his/her family, etc.. The conceptual data model of the database is depicted in Figure 2.

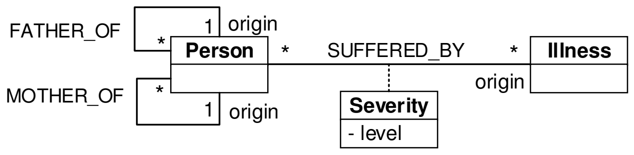
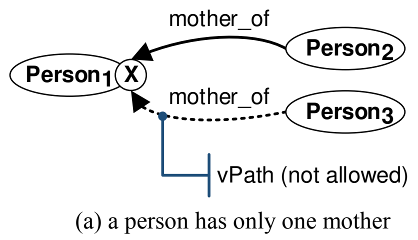


Figure 2. Conceptual data model of the "illness risk" application

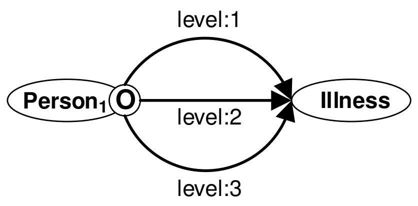
Some interesting features to test are situations in which: (a) a person has only one mother; (b) a person can suffer several episodes of the same illness with different severity levels; (c) an illness can be suffered by several people of the same family.

Our approach allows the tester to define test views based on the system specification, which indicate interesting nodes and relationships of the test database to test the application behaviour. Figure 3 depicts the test views that correspond to the aforementioned features to test. The elements that compose a test view are also identified:

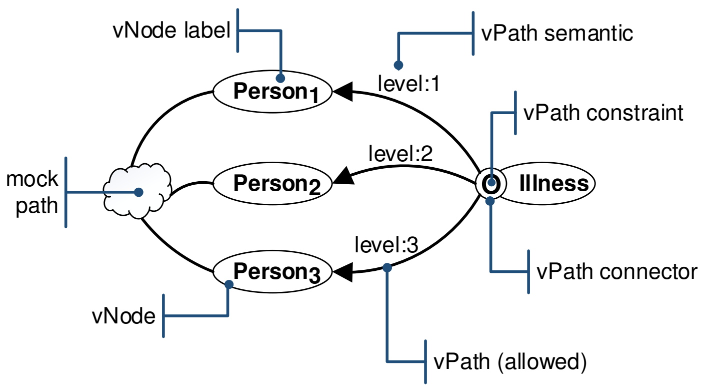
- *View node or vNode*: a type of node of the database. The *vNode label* indicates the class that represents the vNode in the conceptual data model. A type of node can be unique in a test view, generating only one vNode (like the vNode "Illness"), or have several instances, giving rise to several vNodes denoted by *class$_i$*, (the subscript represents the number of the instance of this vNode). For example, the vNodes "Person$_1$", "Person$_2$" and "Person$_3$" are three different instances of the same type of node "Person".
- *View path* or *vPath*: a directed path that relates two vNodes according to a specific semantic derived from the relationships of the conceptual data model, which is indicated by the label *vPath semantic*. There are two types of vPaths: allowed and not allowed, which specify that a vPath can appear or cannot appear in a database, respectively.
- *Mock path*: a not completely defined path that relates two or more vNodes. The testing objective is not focused on any specific path that relates these vNodes, but it is focused on its existence.
- *vPath constraint*: a restriction over a group of vPaths, which constraints whether each one can, cannot or must appear at the same time in the database. There are three types of vPath constraints: *XOR* (represented by "X") indicates that only one allowed vPath must appear in the database; *OR* (represented by "O") indicates that several allowed vPaths can appear at the same time in the database; *AND* (represented by "+") indicates that all allowed vPaths constrained must appear at the same time in the database.
- *vPath connector* (*connector*, for short): joins a group of vPaths that are restricted by the same vPath constraint. A connector can join vPaths that start in the same vNode or vPaths that end in the same vNode.



(a) a person has only one mother



(b) a person suffers several episodes of the same illness



(c) an illness can be suffered by several people of the same family

Figure 3. Examples of test views

The test view of Figure 3(a) indicates that the vPath between "Person$_2$" and "Person$_1$" must appear in the database, whereas the vPath between "Person$_3$" and "Person$_1$" cannot appear. Figure 3(b) indicates several vPaths that represent different severity levels of an illness. One or more of these vPaths can appear in the database between an instance of "Person" and an instance of "Illness". Finally, the test view of Figure 3(c) indicates that three different people have suffered an illness with different severity levels (vPaths from "Illness" to "Person$_1$", "Person$_2$" and "Person$_3$"). One or more of these vPaths can appear in the database. The mock path indicates that there can be family relationships between "Person$_1$", "Person$_2$" and "Person$_3$", but these relationships are not exactly defined.

After defining the test views, transformations guided by some test selection criterion can derive automatically the test coverage items. These test coverage items can be automatically mapped to executable test cases by means of transformations guided by a test generation technique.

Our approach allows the tester to define several types of test views, however, due to the lack of space we only present three examples.

5. CASE STUDY

To illustrate how our approach can be applied, a real-world example of a graph database application, called "authorization and access control" [29], has been used. This application represents the business of an international communications services company, which offers its customer organizations the ability of self-service their accounts. Organization administrators can add and remove services on behalf of their employees. To ensure that resources are only seen and changed by the entitled users, a complex access control system has been designed, considering different types of permissions and hierarchy structures among organizations. The conceptual data model of the database is depicted in Figure 4.

Administrators are assigned to one or several groups, and these groups have several permissions against the organizational structure. Each organization can be the parent of several organizations, with their own employees and accounts to manage. The permissions defined among groups and organizations are: (1) *allowed_inherit* allows administrators within the group to manage the accounts of both the organization and its children; (2) *allowed_do_not_inherit* allows the administrator with the group to manage the organization, but not its children; (3) *denied* forbids administrators with a group to manage the organization and its children. The access control system also establishes a permission precedence, because an administrator can be a member of two groups within different permissions against the same organizations. So, the permission *denied* takes precedence over *allowed_inherit*, and *allowed_do_not_inherit* prevails over *denied*.

The system specification defines three queries to find all accessible accounts for an administrator (shown in Figure 5), to determine whether an administrator has access to an account and to find all administrators for an account.

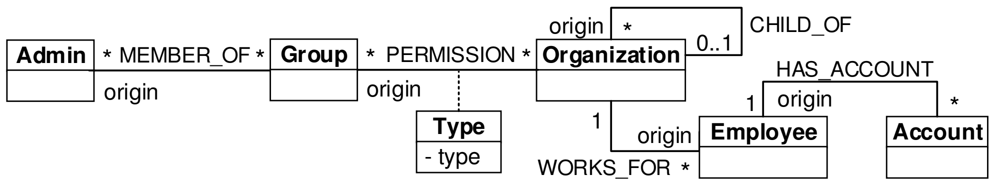


Figure 4. Conceptual data model of the "authorization and access control" application

```
START admin=node:administrator(name={administratorName})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_INHERIT]->()
        <-[:CHILD_OF*0..3]-(company)<-[:WORKS_FOR]-(employee)
        -[:HAS_ACCOUNT]->(account)
WHERE NOT ((admin)-[:MEMBER_OF]->()-[:DENIED]->()<-[:CHILD_OF*0..3]-(company))
RETURN employee.name AS employee, account.name AS account
UNION
START admin=node:administrator(name={administratorName})
MATCH paths=(admin)-[:MEMBER_OF]->()-[:ALLOWED_DO_NOT_INHERIT]->()
        <-[:WORKS_FOR]-(employee)-[:HAS_ACCOUNT]->(account)
RETURN employee.name AS employee, account.name AS account
```

Figure 5. Cypher query for finding all accessible accounts for an administrator

First, we designed several test views, according to the system specification. One of them can be seen in Figure 6: a group can have different permissions against different organizations, which have a hierarchical structure. The "void" permission indicates that the group does not have an explicit permission against "Organization$_4$". The objective of this test view is to test the inheritance of the different types of permissions.

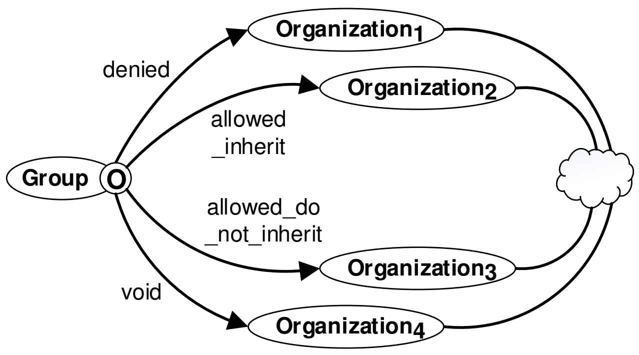


Figure 6. Test view of the "authorization and access control" application

Then, we transformed the test views into the test coverage items using a script that implements a combinatorial technique based on permutations without repetition. For example, for the test view of Figure 6 the script carried out permutations without repetition over the vNodes related by the mock path to generate different hierarchical orders between them. As a result, 24 test coverage items were generated automatically. Two of these test coverage items are shown in Figure 7. Note that the mock paths are now directed paths to indicate the particular hierarchical structure represented by the test coverage item.

From the test coverage items, we generated the test database, considering the specific graph database (Neo4j in our case [27]).

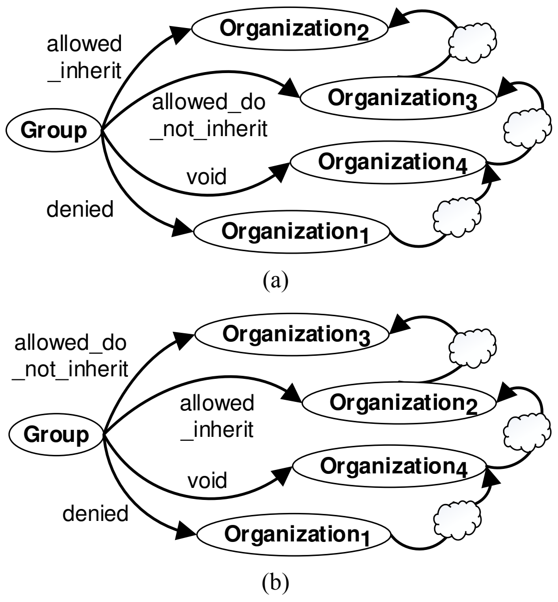


Figure 7. Test coverage items of the "authorization and access control" application

Figure 8 shows the nodes and relationships that were introduced into the test database to cover the test coverage items of Figure 7. The nodes "G1", "O1", "O2", "O3" and "O4" (and their relationships) cover the test coverage item of Figure 7(a), while the nodes "G1", "O5", "O6", "O7" and "O8" (and their relationships) cover the test coverage item of Figure 7(b). The other nodes and relationships were used to conform to the conceptual data model. Finally, we generated the test code that was executed against the SUT using the languages Cypher and Java. At present, the test database and the test code are generated by hand, however both tasks will be automated in the future.

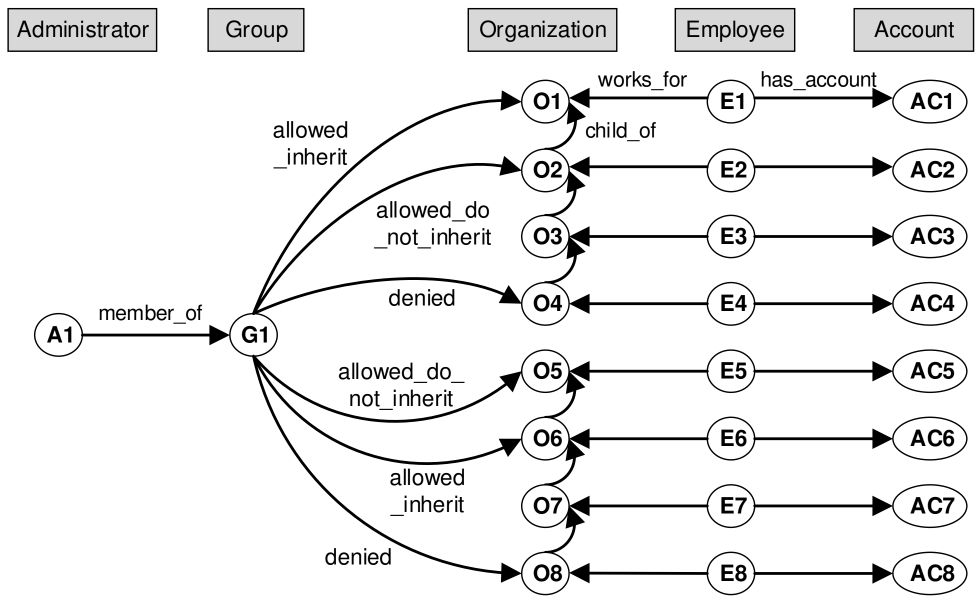


Figure 8. Extract of the test database of the "authorization and access control" application

The execution of the test cases, which take as input the test database generated, reported that "A1" has access to the accounts "AC1", "AC2", AC3", "AC5", "AC6" and "AC7", but should "A1" have access to the accounts "AC3", "AC6" and "AC7"? We do not know because the system specification does not indicate the preference between the *allowed_inherit* and the *allowed_do_not_inherit* permissions. So, the test cases detected a fault. If the observed output is equal to the expected output, the specification has a fault because it is incomplete. If the observed output is not equal to the expected output, both the specification and the implementation have a fault.

6. CONCLUSIONS AND FUTURE WORK

We have presented an approach to test graph database applications. This approach defines a test model taking into account the conceptual data model of the SUT and the system specification. The test model is composed of several test views that represent the important features of the SUT to be tested. To automate the generation of test cases from the test model we have proposed a framework that places MBT in the MDA context.

The results of the case study show that the test cases obtained from the test model reported that an administrator had access to some resources that could be forbidden (the system specification is not complete). An incomplete specification can cause defects in the applications, as developers could make erroneous assumptions about what the system must do; the increase of costs, since new code could be developed, and of course tested, when the omission is detected; and even the dissatisfaction of the customers, as the system does not meet their needs.

Future work includes several avenues. On the one hand, the definition of test selection criteria that consider the characteristics of the test views to derive the test coverage items and the development of techniques to generate executable test cases for graph database applications. Furthermore, the elaboration of the test views could be partially automated to represent different strategies and patterns of features that should be tested. At present, the generation of test coverage items has been automated, however other aspects can be automated, such as the transformations between the other models. As part of future work, we will define transformations between models that allow automating the process and we will develop a tool implementing the framework proposed.

7. ACKNOWLEDGMENTS

This work was supported by projects TIN2013-46928-C3-1-R, funded by the Spanish Ministry of Science and Technology, and GRUPIN14-007, funded by the Principality of Asturias (Spain) and ERDF funds.

8. REFERENCES

[1] Alves, E.L.G., Machado, P.D.L., Ramalho, F. 2014. Automatic generation of built-in contract test drivers. *Software and Systems Modelling*, 13(3), 1141-1165.

[2] Barmpis, K., Kolovos, D.S. 2014. Evaluation of Contemporary Graph Databases for Efficient Persistence of Large-Scale Models. *Journal of Object Technology*, 13(2), pp 3:1-26.

[3] Binnig, C., Kossmann, D., Lo, E. 2008. MultiRQP - Generating test databases for the functional testing of OLTP applications. In *Proceedings of the 1st International Workshop on Testing Database Systems*.

[4] Blanco, R., Tuya, J., Seco, R.V. 2012. Test adequacy evaluation for the user-database interaction: a specification-based approach. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 71-80.

[5] Busch, M., Chaparadza, R., Dai, Z., Hoffmann, A., Lacmene, L, Ngwangwen, T., Ndem, G., Ogawa, H., Serbanescu, D., Schieferdecker, I., Zander-Nowicka,J. 2006. *Model transformers for test generation from system models*. Technical report, Fraunhofer FOKUS,Germany and Hitachi Central Research Laboratory Ltd., Japan.

[6] Ciccozzi, F., Cicchetti, A., Siljamäki, T, Kavadiya, J. 2010. Automating test cases generation: from xtuml system models to qml test models. In *Proceedings of International Workshop on Model-Based Methodologies for Pervasive and Embedded Software*, pp. 9–16.

[7] Chays, D., Deng, Y., Frankl, P.G., Dan, S., Vokolos, F.I., Weyuker, E.J. 2004. An AGENDA for testing relational database applications. *Software Testing, Verification and Reliability,* 14(1), 17-44.

[8] Dai, Z.R. 2004. Model-driven testing with UML 2.0. In *Proceedings of the Second European Workshop on Model Driven Architecture*, pp. 179-187.

[9] de la Riva, C., Suárez-Cabal, M.J., Tuya, J. 2010. Constraint-based test database generation for SQL queries. In *Proceedings of the 5th International Workshop on Automation of Software Test*, pp. 67-74.

[10] Emmi, M., Majumdar, R., Sen, K. 2007. Dynamic Test input generation of database applications. In *Proceedings of the International Symposium on Software Testing and Analysis*, pp. 151-162.

[11] Fujiwara, S., Munakata, K., Maeda, Y., Katayama, A., Uehara, T. 2011. Test data generation for web application using a UML class diagram with OCL constraints. *Innovations in Systems and Software Engineering*, 7(4), 275-282.

[12] Gardikiotis, S.K., Malevris, N. 2009. A Two-folded Impact Analysis of Schema Changes on Database Applications. *International Journal of Automation and Computing*, 6(2) 109-123.

[13] Garg, D., Datta A. 2012. Test Case Priorization due to Database Changes in Web Applications. In *Proceedings of the 5th International Conference on Software Testing, Verification and Validation*, pp. 726-730.

[14] Halfond, W.G.J., Orso, A. 2006. Command-form coverage for testing database applications. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, pp. 69-80.

[15] Kapfhammer, G.M., Soffa, M.L. 2003. A family of test adequacy criteria for database-driven applications. In *Proceedings of the 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT Int'l Symposium on the Foundations of Software Engineering*, pp. 98-107.

[16] Kapfhammer, G.M., Soffa M.L. 2008. Database-aware test coverage monitoring. In *Proceedings of the 1st India Software Engineering Conference*, pp. 77-86

[17] Kapfhammer, G.M., McMinn, P., Wright, C.J. 2013. Search-Based Testing of Relational Schema Integrity Constraints Across Multiple Database Management Systems. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation*, pp. 31-40.

[18] Khalek, S.A., Elkarablieh, B., Laleye, Y.O., Khurshid, A. 2008. Query-aware test Generation using a relational constraint solver. In *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pp. 238-247.

[19] Lamancha, B.P., Reales, P., García, I., M. Polo, Piattini, M. 2009. Automated Model-based Testing using the UML Testing Profile and QVT. In *Proceedings of the 6th International Workshop on Model-Driven Engineering, Verification and Validation*, pp. 1-10.

[20] Lamancha, B.P, Reales P., Polo M., Caivano, D. 2011. Model-driven testing: transformations from test models to test code. In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering*, pp. 121-130.

[21] Leavitt, N. 2010. Will NoSQL databases live up to their promise? *IEEE Computer*, 43(2) 12-14.

[22] Liu, Y., Li, Y., Wang, P. 2010. Design and implementation of automatic generation of test cases based on model driven architecture. In *Proceedings of the 2nd International Conference on Information Technology and Computer Science*, pp. 344-347.

[23] Lo, E., Binnig, C., Kossmann, D., Özsu, M.T., Hon, W.K. 2010. A framework for testing DBMS features. *The VLDB Journal*, 19(2), pp. 203-230.

[24] Marcozzi, M., Vanhoof, W., Hainaut, J.L. 2013. A relational symbolic execution algorithm for constraint-based testing of database programs. In *Proceedings of the 13th International Working Conference on Source Code Analysis and Manipulation*, pp. 179-188.

[25] McMinn, P., Wright, C.J., Kapfhammer, G.M. 2015. An Analysis of the Effectiveness of Different Coverage Criteria for Testing Relational Database Schema Integrity Constraints. Technical Report CS-15-01, University of Sheffield.

[26] Moniruzzaman, A.B.M., Hossain, S.H. 2013. NoSQL Database: New Era of Databases for Big data Analytics-Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4) 1-14.

[27] Neo4J, http://neo4j.com

[28] NoSQL databases, http://nosql-database.org

[29] Robinson, I., Webber, J., Eifrem, E. 2013. *Graph databases*. O'Reilly.

[30] Schieferdecker, I. 2012. Model-based testing. *IEEE Software* 29, 14-18.

[31] Tuya, J., Suárez-Cabal M.J., de la Riva, C. 2006. SQLMutation: a tool to generate mutants of SQL database queries. In *Proceedings of the Second Workshop on Mutation Analysis*.

[32] Tuya, T., Suárez-Cabal, M.J., de la Riva, C.2007. Mutating database queries. *Information and Software Technology*, 49(4) 398-417.

[33] Tuya, T., Suárez-Cabal, M.J., de la Riva, C. 2010. Full predicate coverage for testing SQL database queries. *Software Testing Verification and Reliability*, 20(3) 237-288.

[34] Utting, M., Pretschner, A., Legeard, B. 2012. A taxonomy of model-based testing approach. *Software Testing, Verification and Reliability*, 22(5) 297-312.

[35] Willmor, D., Embury, S.M. 2005. Exploring test adequacy for database systems. In *Proceedings of the 3rd UK Software Testing Research Group*, pp. 123-133.

[36] Willmor, D., Embury, S.M. 2006. Testing the implementation of business rules using intensional database

tests. In *Proceedings of Testing: Academic & Industrial Conference on Practice and Research Techniques*, pp. 115-126.

[37] Wright, C.J., Kapfhammer, G.M., McMinn, P. 2013. Efficient Mutation Analysis Of Relational Database Structure Using Mutant Schemata And Parallelisation. In *Proceedings of the 6th International Conference on Software Testing, Verification and Validation Workshops*, pp. 63-72.

[38] Zander, J., Dai, Z.R., Schieferdecker, I., Din, G. 2005. From U2TP models to executable tests with TTCN-3 - an approach to model driven testing. In *Testing of Communicating Systems*. LNCS 3502, pp.289-303.

[39] Zhou, C., Frankl, P. 2011. JDAMA: Java Database Application Mutation Analyzer. *Software Testing, Verification and Reliability*, 21(3), 241-263.