Automated Model-based Testing Based on an Agnostic-platform Modeling Language

Concepción Sanz[1], Alejandro Salas[1], Miguel de Miguel[1,2], Alejandro Alonso[1,2], Juan Antonio de la Puente[1,2] and Clara Benac[1,3]

[1]*Center for Open Middleware, Universidad Politécnica de Madrid (UPM), Campus de Montegancedo, Pozuelo de Alarcón, Madrid, Spain*

[2]*DIT, Universidad Politécnica de Madrid (UPM), Madrid, Spain*

[3]*LSIIS, Universidad Politécnica de Madrid (UPM), Madrid, Spain*

Keywords: Model-based Testing, Automated Testing, Agile Development.

Abstract: Currently multiple Domain Specific Languages (DSLs) are used for model-driven software development, in some specific domains. Software development methods, such as agile development, are test-centered, and their application in model-based frameworks requires model support for test development. We introduce a specific language to define generic test models, which can be automatically transformed into executable tests for particular testing platforms. The resulting test models represent the test plan for applications also built according to a model-based approach. The approach presented here includes some customisations for the application of the developed languages and transformation tools for some specific testing platforms. These languages and tools have been integrated with some specific DSL designed for software development.

1 INTRODUCTION

Along time, different methodologies have been applied to software development in order to improve the product quality and reduce the time-to-market with tight budgets. From the most traditional approaches -such as waterfall development-, to the latest variations of agile development, testing has been a key phase to perform due to the need for detection and correction of defects as soon as possible; but it is also costly and highly time consuming. As companies tend to apply methodologies which shorten the software development lifecycle, it is necessary to adapt testing to more demanding scenarios where time and resources are limited. For instance, in agile methodologies testing activities are performed during the whole lifecycle. This means that it is possible to start the design of a test plan as early in the development process as the requirements specification phase, and evolve it in parallel to software. In this scenario, an adaptive and responsive framework to design tests and allow their automation increases productivity, specially when regression and integration tasks need to be carried out. Other challenges linked to testing come from the different target platforms and development technologies that can be involved in the process, which are usually known only by testers. This complexity prevents other prospective users, such as application developers, from doing their own tests. Moreover, testing can be so coupled to specific management tools that any migration could have a quite significant impact.

This work proposes a testing framework for testing model-based software applications. The testing framework is based on a model-driven approach, having developed a modelling language that allows to build test models which are agnostic about the final testing platform. In this way, the complexity is hidden to users, who only manage high-level concepts, easing the integration of test frameworks into software development frameworks. The reuse and customization of models are allowed, avoiding unnecessary repetition along test design. Once test plans are built as agnostic platform models, the framework is also responsible for transforming them, in an automatic way, into executable tests for specific target testing platforms. Some examples of testing platforms and frameworks are JUnit (Tahchiev et al., 2010), Selenium (Holmes and Kellogg, 2006) and QTP-HP (Rao, 2011). The automation of the transformation process eases regression tasks, as well as maintenance, since

Sanz C., Salas A., de Miguel M., Alonso A., Antonio de la Puente J. and Benac C..
Automated Model-based Testing Based on an Agnostic-platform Modeling Language.
DOI: 10.5220/0005237802390246
In *Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development* (MODELSWARD-2015), pages 239-246
ISBN: 978-989-758-083-3

only abstract test models need to be changed.

The approach presented here has been applied to a specific modelling framework, BankSphere, a tool suite created by the Santander Group which integrates model-based design tools in a deployment and runtime environment. This suite, used by the large community of developers existing in Santander Group, reduces significantly the development time of new banking software.

The rest of the paper is structured as follows. Section 2 summarizes different approaches linked to testing. Section 3 gives an overview of the test architecture proposed in this work. Section 4 depicts the main concepts contained in the developed language, providing a case study for a better comprehension. Section 5 shows a case where the presented approach has been applied to generate executable tests. Finally, 6 summarizes the main results of this work and sketches the lines for future work.

2 RELATED WORK

Model-based testing is the paradigm by which test cases are automatically generated using models that describe the functional behaviour of the system under test. This approach allows testers focusing on the design of better test plans instead of wasting effort in coding tasks, increasing productivity and product quality due to the automation. Models are designed using from formal specifications (e.g. B, Z) (Utting and Legeard, 2007; Cristiá and Monetti, 2009) to a large variety of diagrams, such as state charts, use case, sequence diagrams (Briand and Labiche, 2001), (extended) finite state machines (Pedrosa et al., 2013; Karl, 2013) or graphs. Most of research in models for testing is focused on UML modeling language (Fowler, 2003). A common practice is using sequence diagrams to represent the system behaviour for use case scenarios. Diagrams are then transformed into test case models -such as xUnit (Meszaros, 2006)- by means of model-to-model transformations, before being transformed -model-to-text - into executable tests for a specific platform (Javed et al., 2007).

The widespread use of UML in testing has led to the development of a specific standard profile, UML 2.0 Testing Profile -U2TP- (Baker et al., 2004), an extension to define and build test cases for software systems. Different authors have used it to derive test artefacts from systems previously described in UML (Lamancha et al., 2009; Wendland et al., 2013; Baker et al., 2007). Although U2TP bridges the gap between system and test development by using the same modelling notation, public implementations are incomplete (Wendland et al., 2013; Eclipse(a), 2011). Implementations are focused mainly on the Test Architecture section and partially on Test Behaviour, where not all the defined concepts are covered. Other sections, such as Test Data, are usually also skipped.

UML is the most widespread modelling language, but some other DSLs are also used for software development (Fowler, 2010). UML profiles are UML dependent and they are not applicable on DSL. The languages introduced in this paper are independent of development languages.

In contrast to most of literature on model-based testing, the proposal described here does not derive test models from other models which represent the system behaviour (Javed et al., 2007; Lamancha et al., 2009). The executable tests are neither derived from code or system requirements (Wendland et al., 2013). Instead, this work proposes a UML-independent approach, where test models are designed by testers and automatically translated to be executable.

3 TEST ARCHITECTURE

In this section, the architecture proposed is briefly depicted before being explained more deeply next. The aim of the architecture is the building of generic test models, agnostic about any testing platform.

3.1 Architecture Overview

The architecture proposed in this work is shown in Figure 1, as well as the role played by users. The core of the architecture is the testing models, whose main concepts will be described in Section 4. The testing modelling language is supported by an editor specially developed to help users to design test models graphically, improving the user experience and hiding

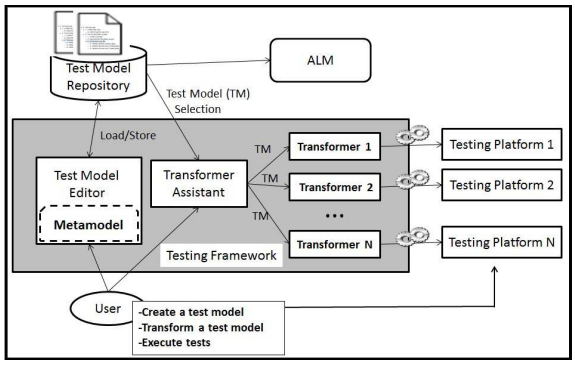

Figure 1: The proposed architecture allows users to be agnostic about the final target platform, focusing the effort on building of test models.

the complexity of the language. The resulting models conform the developed modelling language and its constraints. These models are then stored in a regular repository in order to be accessible by different elements in the architecture and from outside. Inside the architecture, the stored models are accessed by: (1) the editor, in case of performing modifications in the test models; (2) the transformer assistant, when users select models to be transformed into executable tests. From outside, the repository can be accessed by application lifecycle management tools (ALM) for lifecycle supervision activities. From these three actors, only the editor can update the repository, being used in a read-only mode by the others.

A test plan based on the proposed language can be transformed in executable tests for specific testing platforms. For this task, a transformer assistant helps users to select a model from the repository, and the target platform where tests will be finally executed. Then, the assistant delegates the test model to the right engine, which will apply specific transformation rules to automatically translate the agnostic model into a collection of tests executable in the chosen platform. Due to the agnostic nature of test models and the existence of specific transformation engines, the same model can be executed in more than one testing platform. The engines are responsible for the appropriate adaptation of models to each platform. When the transformation process ends, the resulting test collection is available in the final testing platform for execution, which in case of being also automated, hides the potential complexity of the process. Information obtained as a result of the executions should be retrieved to be included in the repository, being associated to its corresponding test model and target platform.

The advantages of agnostic testing models are numerous. Firstly, since models are independent from the management solution (ALM) adopted by the company, migration from one tool to another due to changes in the business strategy just requires an adaptor between the model repository and the new tool to populate it with the required information in each case. Models are not altered in this process. Secondly, once a test model has been built, it can be executed as many times as needed due to the automation of the transformation process, providing time saving when regression tests are needed. This automation hides the complexity of the target testing platform and reduces failures due to human action. Thirdly, when application functionalities are modified, it is only necessary to adapt the associated model, instead of the executable tests. Moreover, any modification in the testing platform only affects to the transformation engines. Finally, models can be built and used by people who are not necessarily test experts, broadening the use of tests in the development process and focused on each user's expertise area.

3.2 Testing Modelling Language Overview

The abstract syntax has been implemented using Eclipse Modeling Framework (EMF) (Eclipse(c), 2013). The number and nature of the concepts managed make necessary to build the main specification as a group of smaller packages, each of them focused on a group of concepts closely related among them. The core of the specification are four packages which covers concepts regarding structure, behaviour, context and results respectively. There is also a last package to manage common concepts such as data or filters. The three first packages are the most evolved at the moment, being deeply described in Section 4.

3.3 Test Model Editor

A graphical editor has been developed using Graphical Modeling Framework -GMF- (Eclipse(b), 2010). Depending on the test plan to build, the design can be complex enough to be difficult to visualize and manage in a single diagram. For this reason, the developed editor distinguishes among different diagrams, each of them focused on a portion of the proposed language. Thus, different diagrams allow to design tests at different levels, providing from a general overview of the test plan to a lower granularity focused on the test case level. There are three kind of diagrams to visualize different aspects of the structural design, whereas there is only one specific diagram to describe contextual information. The combination of several of these diagrams gives rise to a whole test model.

3.4 Test Model Transformation

The transformation from agnostic test models to collections of specific tests for particular testing platforms is performed currently by using the operational QVT language (OMG, 2011), the OMG's standard to perform model-to-model transformations between EMF models. In case that test plans in the target platform could not be described by its own metamodel, model-to-text transformations would be required instead.

4 TESTING MODELLING LANGUAGE

The core of the proposed architecture is the test models mentioned previously. For the sake of clarity, only the main concepts related to structure, behaviour and context will be depicted in this section. For a better explanation of the concepts managed, a case study will be used to give some examples about the potential of the developed modelling language. The examples will be based on a banking application described next.

4.1 *PiggyBank*: A Case Study

PiggyBank is the experimental use of case chosen to show the concepts managed in the developed language. It is a simplified application to access a banking system, frequently used to study the testing approaches in model-based software development. The application has been created using a fully model-driven approach, where every functionality has been built with the modelling framework BankSphere based on extended state diagrams. The framework translates automatically each of the models into executable code for web applications. *PiggyBank*'s main functionalities are: (1) a login process, mandatory to gain access to banking tasks; (2) balance visualization; (3) money transfer; and (4) system disconnection. Having this application as case study, the aim is to build a complete test plan for it using the designed language, a test plan agnostic about any testing platform.

4.2 Structure Package

This package groups all the elements which provide a hierarchical structure to the test plan. The concepts managed are flexible enough to create a large variety of hierarchies, from the most simple ones to the most complex, being able to build nested structures and reuse elements from other test plans already designed. A summary of this package can be seen in Figure 2. The most important relations with concepts from other packages are also shown.

The main concepts managed in this package are *TestProject*, *TestSuite* and *TestCaseBase*. *TestProject* is the root for any test plan, identifying the System-Under-Test (SUT) to consider and grouping all the functionalities to test in the application. This element contains an undefined number of *TestSuites*, each of them can be described as a set of test cases which are functionally related. This *TestSuite* concept provides complexity to the test plan since it can contain more *TestSuites* and point to other structural elements, allowing the reuse of testing structures. Finally, a *TestSuite* - which represents a particular functionality- contains the *TestCaseBase* concept, this is each individual test that need to be considered in the test plan. These three elements have a common superclass (*StructuralTestElement*) which, among other attributes, owns: information about the result of the execution of that element, annotations to customize attributes when an element is being reused, and a *Context* attribute to provide behavioural information as well as other kind of parameters necessary at runtime. Another attribute contains the set of requirements covered by a structural element, easing the gathering of elements affected by changes in a requirement. *TestCaseBase* elements are slightly different from the other structural elements since they are the ones that will provide a unique behaviour to each test. In such a way, *TestCaseBase* is similar to an activity diagram in a traditional UML approach. This element, apart from the context with certain behavioural information, it owns another attribute to complete the expected behaviour and validations for the test (link to *BehaviouralObject* in Figure 2).

A novelty in this package is the concept of *TestScenario*, which is only contained by *TestCaseBase* elements. This concept allows to set the data that will be executed at runtime for a particular *TestCaseBase*, and represents a possible extension to the set of validations already defined in the *TestCaseBase* element. The set of validations at *TestCaseBase* level are common to all the *TestScenarios* existing in the structural element. However, the extra validations at *TestScenario* level indicates specific checkers that need to be added for execution, which will probable depends of the specific data set in the *TestScenario*. Thus, *TestScenarios* represent variations of a *TestCaseBase* depending on the input data selected. Data collections set at *TestScenario* level are defined at pools, which act as repositories to reuse information. Data own a specific structure that allows to use them as a collection (valid/invalid customers), or as a kind of registers, being able to reference specific fields (e.g. name, surname, etc).

Considering our experimental application, *PiggyBank*, the concepts described here can be used to design the core of the test plan as follows. Since *PiggyBank* has a reduced number of functionalities, the test plan for the whole application can be built using a unique *TestProject*. This project will contain a *TestSuite* element for each functionality, i.e. Login, Money Transfer, Display Balance and Disconnect. Suppose the login access is a common task in most of the applications, not only *PiggyBank*, and it has been established a fixed building pattern to cod-

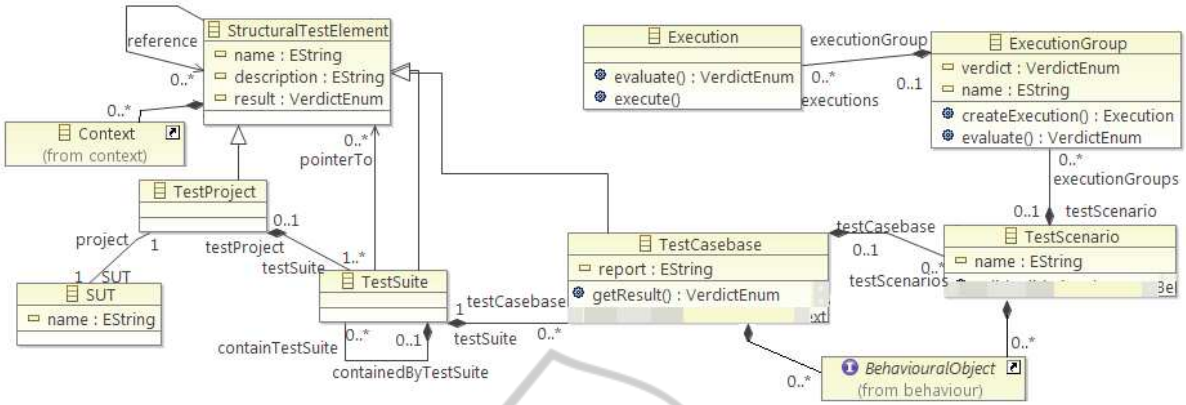

Figure 2: Structure package.

ify this task. In this situation it could make sense that the login access owns its own test plan in a specific *TestProject*, independent from the rest of the applications. This test plan could be reused in the *TestSuite* Login for *PiggyBank* just pointing to it with the appropriate attribute. Thus, it would be only necessary to use annotations to rename some data if necessary. Neither of the remaining suites would require further complexity except being containers for *TestCaseBase* elements. For instance, considering the *TestSuite* Display Balance, there would be a *TestCaseBase* to check the existence of a return button in the page (testCaseBase_1) and another one to check the existence of a message with the customer name and the balance of the bank account (testCaseBase_2). At *TestCaseBase* level there would be defined validations to check the appropriate information in each case. The data used at runtime would be set on *TestScenarios*. As an example, for testCaseBase_1 any set of valid users would be possible. However, for testCaseBase_2 it would be necessary, for instance, to divide valid users depending on their balance, positive or negative. In this case, testCaseBase_2 would require two *TestScenarios*, one to point to a set of registers of customers with positive balance and a second scenario pointing to registers of customers with negative balance. Probably, in case of negative balance, extra validations would be required, such as checking that a special message also appears on the web page. This extra validations would be indicated at *TestScenario* level, not at *TestCaseBase* level since it only affects a specific group of users.

To conclude with the Structure package, and in order to keep information about the execution of a particular test plan, at *TestScenario* level there are structures to save a number of basic details about each execution performed at the testing platform. These structures are *ExecutionGroup* and *Execution*. Each individual execution would keep a report of the data used for that particular execution and its result. These data could be the name of the browser, the user and password used in the login process, etc. *Executions* are grouped on *ExecutionGroups*, giving a unique verdict for the whole set of executions. These structures collect results from the target testing platform and provide verdicts which are moved up in the test structure until the *TestProject* level is reached and a unique verdict for the whole test plan can be provided.

4.3 Behaviour Package

This package groups the elements that provide the workflow of a test. A summary of this package can be seen in Figure 3.

The most important concept is the *BehaviouralElement*, which contains the smallest elements with behaviour in a test, *ActionElements* and *ValidationElements*. These elements will have a direct translation to one or more basic execution units (functions, services, etc.) in each target testing platform, being the transformation engines the responsible for providing the right translation in each case. *ValidationElements* represent only validating and checking behaviours. They own an attribute *type* which can be used to set how important is the result of a particular validation. Thus, a validation that fails when the user expects a successful result can be reported just as a warning if type is *weak*, or as an error or an exception if type has been set to *strong*. These kind of attributes needs also to be considered during the codification of each transformation engine to provide the right translation. *ActionElements* represent non-validating behaviours, such as filling or clicking. These behavioural elements contain attributes to customize the behaviour (i.e. a text that needs to be checked or references to data from pools).

Action and validation elements can be grouped in a specific sequence in order to create a single reusable unit, a *BehaviouralGroup*. Thus, a group can be cre-

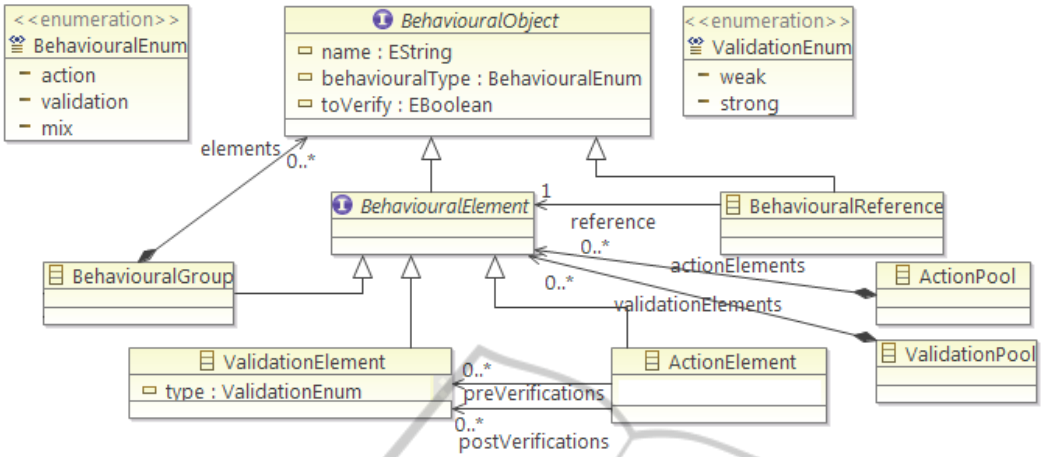

Figure 3: Behaviour Package.

ated once, and used several times. The reuse of *BehaviouralElements* is performed due to the existence of *BehaviouralReferences*, a special behavioural object which allows to point to any *BehaviouralElement* and customize their attributes to adapt them to each particular situation. A *BehaviouralGroup* can also include references among its collection of elements.

BehaviouralElements can be considered as predefined behaviours, generated by expert tests developers and accessible from pools, which act as repositories for being reused among different users and test plans.

Apart from the predefined elements, it is also possible for a user to create their own behavioural elements. The user would indicate the behaviour of the element and set the collection of attributes which need to be considered for its right execution. These new behaviours will not have a direct translation in transformation engines unless they are specifically included if its considered necessary. The advantage of creating behavioural elements is that the update of transformation engines is based on users' needs.

The described *BehaviouralObjects* are used in very few parts of the whole developed package. Regarding the elements in the *Structure Package*, only *TestCaseBase* owns a list of *BehaviouralObjects*. *TestScenarios*, due to its nature only owns a list of *ValidationElements* or references to those kind of elements. Apart from these structural elements, no other elements contain behaviour except for *Context*, an element contained in the package context, explained next.

4.4 Context Testing Language

This last package is focused on the concept of context, which can be understood as a collection of information which: (1) helps the system to reach a specific execution point before going on with the test; It moves the system from a state A to another B, which is considered the right starting point for the test to execute; and (2) ensures that the system has reached that execution point.

The concepts contained in this package, extended with some auxiliary terms from other packages for better understanding, can be seen in Figure 4.

A context includes behavioural elements - *BehaviouralObjects* of any kind-, and parameters - *ConfigurationParameter*-. The behavioural elements are the responsible for moving the system from one state to another, in a similar way to concepts such as 'set-up' and 'tear-down' in JUnit. The parameters are pairs attribute-value in order to set information related with the testing environment to use at runtime. Examples of parameters for a web application such as *PiggyBank* can be the list of browsers where an entire test plan or a particular test need to be successfully executed or some constant value to be used as a configuration option. These parameters can be as complex as needed since they can be nested for a better description of the execution environment. An example of this nested description, and continuing with the example of browsers for *PiggyBank*, we could imagine a test for which a particular navigator should be checked for a set of specific versions, while for others this version specification would not be necessary. The parameter browser could be described then as:

Browser={IE={version=21, version=22}, Firefox }.

Contexts can also include some other contexts already defined. This is interesting for avoiding repetitions when several structural elements have the same behaviour in their contexts or it is necessary a slightly extension of a context. Finally, a context can also point to a specific *StructuralTestElement* o *TestSce-*

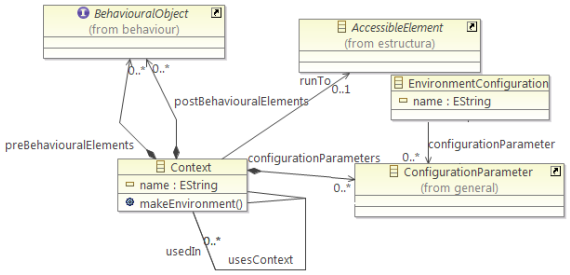

Figure 4: Context Testing Language.

nario (*AccesibleElement*). At runtime, this last feature would execute all the necessary behavioural elements to reach that point and use it as starting point for the rest of the test. To clarify this concept, imagine the *TestSuite* Display Balance. To perform any of the tests contained in the *TestSuite*, it is necessary to execute a login process. A common solution would be repeating the login process again in the context of the *TestSuite* Display Balance. However, this can be done just pointing to the *TestCaseBase* which describes the login test. Once the login process has been included, the user can continue designing the test suite related to checking the balance.

Only the structural elements which are a kind of *StructuralTestElement* own a context attribute. The aim is the design of test plans minimizing the number of elements to manage, since the elements defined in a context can be shared by a large number of structural elements. Focusing on *ConfigurationParameters*, a parameter in a context set at a certain level in the structural hierarchy will be visible by any of its descendant structural elements. As an example, a parameter defined at *TestProject* level will be seen by anyone down in the project, while a parameter defined at *TestCaseBase* will only have influence in that element. Something similar happens with behavioural elements defined in the context of a structural element. Those ones will be visible and performed by any descendant of that element in the hierarchy.

ConfigurationParameters defined as shown allow to reduce the complexity at design time since the transformation engines will be the responsible for exploding all the information contained in these parameters, giving rise to a large number of individual tests in the testing platform that otherwise would be difficult to manage by users. Using *PiggyBank* as example, and given that it is a web application, a context element could be set at *TestProject* level containing all the information that will be common to all the test plan. This information can include *ActionElements*, such as *OpenBrowser($browser)* and *GoToURL($url)*, which will be necessary to perform by any test, but also parameters such as the list of browsers to use at runtime. At a lower level, *TestSuite* Login for instance, the context will only contain actions related with the filling of the access form, since the information to reach that point comes from its parent, the *TestProject* element.

5 FROM TESTING LANGUAGES TO TESTS EXECUTIONS

Section 3 and Figure 1 introduced the general structure for integrating testing platforms and testing modelling language. This approach is based on transformations that generate automatically the inputs of testing platforms based on the testing models.

The testing platform considered in this work uses a database technology for the representation of testing objects and their properties. Examples that use this storage approach are: HP QTP (QuickTest Professional) (Rao, 2011), IBM RFT (Davis et al., 2009) and GP in Santander Group. In this work, GP is the platform chosen as example of transformation. It is currently used for testing different kinds of software, mainly, model-based banking systems, but also their own modelling tools. GP reuses some other testing technologies such as Selenium. GP, as the other platforms based on databases, can be addressed through model-to-model transformations since the store that support the platform can be formalized with a schema, which we can reuse for the construction of an intermediate target modelling language. This language will represent the database of the target testing platform, using Eclipse projects such as Teneo and CDO to support the persistence into the database of models that conform the intermediate language.

Figure 5 shows the general structure of transformation from agnostic testing models to executable GP test projects. The structure of the transformation is decomposed into two QVT transformations; the first one uses the testing model to create the GP structure model, while the second one updates that model to include specific behaviour and validation elements. The

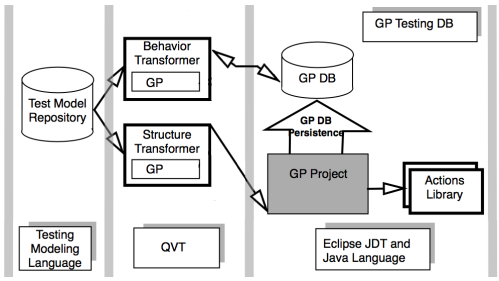

Figure 5: Generation of Test Projects for a database-kind testing platform.

QVT transformer maps agnostic-platform behaviour from the testing model into specific elements of the target platform. This second transformation, also retrieve data and context information, generating all the possible combinations for testing. Each combination will result in a single executable test in GP. When the transformation is completed, it can be persisted into the database. Finally, once tests are executed in GP, using its API, results are integrated in the source testing model.

6 CONCLUSIONS

An architecture has been proposed for testing following a model-based approach. The developed modelling language allows to design test plans which are agnostic about the final target platform. It represents architectural and behavioural aspects using an integrated notation, and provides notation for test data and context information. Models designed according to this language become a generic repository to be used in any testing environment. Transformation engines with the appropriate transformation rules are responsible for generating the right test cases for each target platform. The language includes the reuse of concepts, reducing design time. Transformation engines are also aware of these reuse possibilities, since they would be written by experts in each target testing platform and the modelling language. This approach allows a quick starting of the testing phase, right when the development process begins with the requirements specification. The test plan can be enhanced in an iterative way as more information is obtained from the development process until it captures all the needs established. The graphical editor and transformation engines hides the complexity of testing languages and platforms from non-expert testers, being possible that more people include tests in their working routine.

In the future, behavioural options should be enriched, since only sequential behaviour is considered right now. Results from test executions will be retrieved and included in the models at the repository. In the long term, some support to deal with changes in the SUT could be also considered. Increasing the available transformation engines should be also considered.

ACKNOWLEDGEMENTS

The work for this paper was partially supported by funding from ISBAN and PRODUBAN, under the Center for Open Middleware initiative.

REFERENCES

Baker, P., Dai, Z. R., Grabowski, J., Haugen, O., Samuelsson, E., Schieferdecker, I., and Williams, C. E. (2004). The UML 2.0 Testing Profile.

Baker, P., Dai, Z. R., Grabowski, J., Haugen, O., Schieferdecker, I., and Williams, C. (2007). *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Briand, L. C. and Labiche, Y. (2001). A UML-Based Approach to System Testing. In *Proc. of the 4th Int. Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools*, pages 194–208, London, UK, UK. Springer-Verlag.

Cristiá, M. and Monetti, P. R. (2009). Implementing and applying the stocks-carrington framework for model-based testing. In *ICFEM*, pages 167–185.

Davis, C. et al. (2009). *Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource*. IBM Press, 1st edition.

Eclipse(a) (2011). Eclipse Foundation: Test & Performance Tools Platform (TPTP). http://www.eclipse.org/tptp.

Eclipse(b) (2010). Eclipse Graphical Modeling Framework (GMF). www.eclipse.org/modeling/gmf/.

Eclipse(c) (2013). Eclipse Modeling Framework (EMF). www.eclipse.org/modeling/emf/.

Fowler, M. (2003). *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition.

Fowler, M. (2010). *Domain Specific Languages*. Addison-Wesley Professional, 1st edition.

Holmes, A. and Kellogg, M. (2006). Automating functional tests using Selenium. In *Agile Conf.*, pages 270–275.

Javed, A. Z., Strooper, P., and Watson, G. (2007). Automated generation of test cases using model-driven architecture. In *Automation of Software Test , 2007. AST '07. 2nd Int. Workshop on*, pages 3–3.

Karl, K. (2013). GraphWalker. www.graphwalker.org.

Lamancha, B. P. et al. (2009). Automated Model-based Testing Using the UML Testing Profile and QVT. In *Proc. of the 6th Int. Workshop on Model-Driven Engineering, Verification and Validation*, MoDeVVa '09, pages 1–10, New York, NY, USA. ACM.

Meszaros, G. (2006). *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall, Upper Saddle River, NJ, USA.

OMG (2011). Meta Object Facility (MOF) 2.0 Query/View/Transformation Specification, v1.1.

Pedrosa, C., Lelis, L., and Vieira Moura, A. (2013). Incremental testing of finite state machines. *Software Testing, Verification and Reliability*, 23(8):585–612.

Rao, A. (2011). *HP QuickTest Professional WorkShop Series: Level 1 HP Quicktest*. Outskirts Press.

Tahchiev, P., Leme, F., et al. (2010). *JUnit in Action*. Manning Publications Co., Greenwich, CT, USA.

Utting, M. and Legeard, B. (2007). *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

Wendland, M.-F. et al. (2013). Fokus!MBT: A Multi-paradigmatic Test Modeling Environment. In *Proc. of the Workshop on ACadeMics Tooling with Eclipse*, ACME '13, pages 1–10, New York, NY, USA. ACM.