Executable 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]
[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
Email: {concepcion.sanz, alejandro.salas, miguelangel.demiguel, alejandro.alonso}@centeropenmiddleware.com,
jpuente@dit.upm.es

***Abstract*—Among the different approaches to deal with testing, model-based techniques come up as promising solutions due to their independence from testing platforms, their quick adaptation to changes during the modeling process, and the use of abstract concepts, which make testers focus only on the business. However, most of the solutions are proprietary or based on the complexity of the Unified Modeling Language (UML) models. We propose a testing framework based on a non-UML test-centred modeling language which is agnostic about any execution platform. The framework also allows to increase the expressivity of the test models by designing execution flows, which connect to elements existing in test models. Apart from design tests, the proposed framework deals with execution, including the necessary mechanisms to transform models, on demand and in an automatic way, into executable tests for a variety of testing platforms. In order to show the flexibility of the proposed approach, we introduce as case study two testing platforms which manage tests in different ways.**

***Index Terms*—Model-based testing; Reuse.**

I. INTRODUCTION

Research on testing activities has provided along time a wide variety of approaches to deal with more and more demanding needs in terms of quality assurance, time-to-market, devices, test automation, productivity and maintenance, among others. Among the existing approaches, it can be found mechanisms which were initially considered for purposes different from testing, as it happens with the approximations based on models. Models were only considered for documentation at first, later were included in software development mechanisms and now can be found in testing frameworks as promising alternatives for increasing the automation and the abstraction level of the testing activities.

Most of the non-model-based testing frameworks offer solutions which are highly coupled to the specific platform where tests will be finally executed. This makes necessary having a deep technical knowledge about the target platform, preventing non-expert people to take advantage of the testing frameworks. The absence of models also makes difficult the migration of tests between different testing frameworks. A significant effort in time and work is necessary in order to adapt the entire collection of tests to a new framework. Currently, as software applications increase their complexity and extend the list of devices and platforms where being executed, the variety of frameworks that can be involved during the testing process also increases. Such diversity leads to more complex testing environments and requires specialized testers to manage them. Some examples of testing frameworks are QuickTest Professional(QTP-HP)[1], JUnit [2] and Selenium [3].

In contrast, approaches based on models offer an intermediate layer between the tester and the final testing platform, allowing users to design tests based on abstract concepts, which are independent from any platform. Once tests are described as models, they are translated into executable tests for a specific platform. This is the approach followed by commercial frameworks, such as Conformiq [4], which translates models into JUnit, Selenium or HP Functional Testing [5], among others. The level of abstraction existing in a model-based approach provides a faster mechanism to design tests, since testers only need to focus on the business domain instead of technology. Besides, the conversion from test models to executable tests is usually made in an automatic way, being less error-prone than applying non-modelled approaches directly.

Regarding model-based approaches, most of commercial and open source solutions are usually based on UML, a general-purpose language whose extension and lack of a precise semantic has given rise to the development of more specific languages for testing. Among these, it can be mentioned the specific UML Testing Profile (U2TP) [6], or domain specific languages (DSLs) for easing the use of UML diagrams for testing purposes [7]. These alternatives are still dependent on UML and usually need to be combined with action languages in order to provide behaviour and execution features. Regarding functionality, commercial model-based frameworks offer integrated proprietary solutions where users can design models; execute them in their own platform; or allow the transformation of the models into tests for specific testing languages. However, it is difficult to find all these features in open solutions, so users usually need to integrate and adapt different tools by themselves in order to get similar functionalities.

Consequently, two are the goals to achieve in this work. Firstly, developing an open framework to enable the design of platform-independent test models using abstract concepts ease to manage by non-expert testers and non-UML based. Secondly, enabling the integration of the independent test models with different testing platforms, so users do not require deep knowledge of those platforms to get executable tests.

There has been developed an Eclipse-based open-source testing framework which allows the design of platform-independent test models and their integration with different execution platforms by means of customized transformations.

 ISBN: 978-1-61208-458-9

Thus, one test model can be used in different testing environments just applying, in an automatic way, the right transformation. These transformations enable users to make use of different testing environments simultaneously without being experts on all of them, and ease the migration among platforms. The only expert needed is the one that builds the transformation rules. Among the testing environments involved in the proposed framework, JUnit and Selenium are the ones selected as open-source tools.

The proposed framework is based on a non-UML based modeling language which manages high-level concepts related to testing. These concepts isolate test models from specific testing frameworks and allow the reuse of test elements among models, easing the design of tests to non-experts testers. The framework not only allows to design tests but also establishes platform-independent execution flows using the elements existing in the test models.

The rest of the paper is structured as follows. Sections I and II motivates this work and provides an overview of the state-of-the-art respectively. Section III gives an overview of the testing framework and introduces the way execution flows are set. Section IV explains how the proposed framework connects models to different testing platforms giving rise to test executions. Finally, Section V summarizes the main results of this work and sketches the lines for future work. From the development point of view, a fully functional prototype of the entire proposed framework has been implemented, providing details about it along the different sections of the paper.

II. Related work

The need for producing reliable applications and ensuring the functionality in a scenario of short time-to-market, tight budgets and complex applications, puts pressure on the testing process that the companies carry out. Research on this topic is increasing, giving rise to a large variety of testing approaches, and thus, to a large variety of platforms for testing. This variety requires testers with a large experience and a deep knowledge in each platform in order to take advantage of all the existing features. This expertise prevents other potential users, such as developers, from making or at least sketching their own tests.

Among the most promising approaches applied to testing, we can find solutions based on models. According to this paradigm, test cases can be designed using concepts with low or none technical knowledge. Through these concepts, the functional behaviour of the system under test can be described, giving rise to test models, which can be later transformed into tests which are executable in a specific testing platform. The functional behaviour can be described using concepts based on formal specifications (e.g., B, Z) [8][9], diagrams of any kind - state charts, use case, sequence diagrams [10], (extended) finite state machines [11][12] -, or graphs, among others. Among all, one of the most popular ways to describe tests is based on diagrams, being the UML modeling language [13] the most extended language to design them. UML is a general-purpose modeling language and owns a wide and ambiguous semantic. For this reason, and due to its extended use in testing activities, it was developed a specific UML profile, UML 2.0 Testing Profile (U2TP) [6], which has been used in different research [14][15][16]. However, the ambiguity and complexity still continue in U2TP, motivating the development of alternative DSLs, although still linked to UML [7]. Once models are described, independently of the applied modeling language, its execution is not straightaway due to the absence of an execution engine in the framework used for the design process. Even when an engine is included, models need to be manually adapted or customized in order to be executed.

The proposal described in this work differs from the existing approaches based on models in the fact that the design of test models is completely independent of UML or any other of its associated DSLs. The models proposed are based on a different modeling language, which involves abstract test concepts in order to increase the number of potential test designers. This is possible since models are independent from any testing platform, so no technical knowledge is required from users. The resulting models can be also automatically derived to executable tests for specific testing platforms, once again without requiring technical knowledge from the final user. In summary, the management of abstract concepts, the absence of technical knowledge, and the simplicity and reduced size of the modeling language, compared to UML, can help to reduce the learning curve and increase the potential users of the framework.

Considering model-based testing tools which include the design and execution of tests, these tools are mostly proprietary, being difficult to adapt or extend, and being limited to manage specific testing platforms for execution. On the other hand, open source testing solutions are usually incomplete in terms of functionality. In this proposal, the entire testing process, based on the design, derivation and execution of tests, is integrated in an open source platform. This platform can be easily extended to manage a large variety of testing platforms according to user's needs.

III. Testing framework: Introducing execution flows

The testing framework proposed in this work is outlined in this section, focusing on the way test models can be enhanced by adding execution flows which allow testers to select, from test models, the elements which will be executed on a platform and the flow that these elements will follow during execution.

The aim of the proposed testing framework is to isolate testers from specific platforms as much as possible, allowing them to focus on the design of the tests by means of modeling techniques. The framework will provide the appropriate mechanisms to create test models, and transform them into executable tests with the minimum interaction of the user. Figure 1 summarizes the testing framework developed.

A. Framework overview

The core of the framework relies on the testing modeling language described in [17], which covers abstract concepts related mainly to structural and basic behavioural aspects neces-

 ISBN: 978-1-61208-458-9

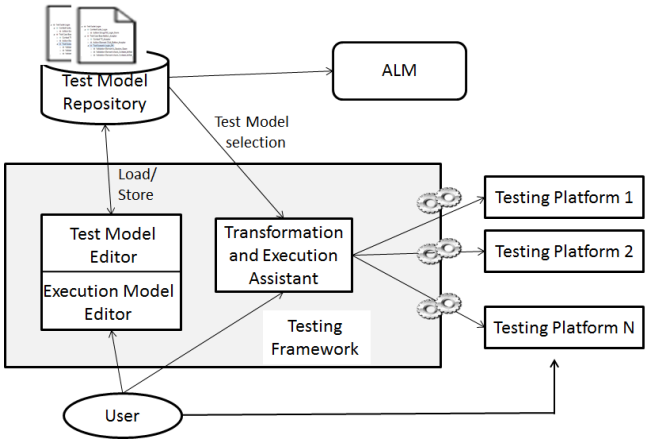

Fig. 1. The proposed framework allows users to focus the effort on the design of the test, being agnostic about any testing platform.

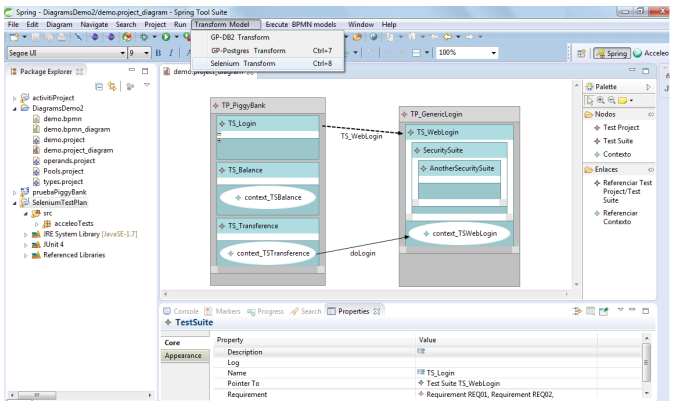

Fig. 2. Example of the developed Test Model Editor building a test model.

sary to design tests in a platform-independent way. An important part of the language is related to the reuse of test elements, structural and behavioural, in order to reduce design time avoiding repetitions and minimizing mistakes or omissions due to those repetitions. This reuse is carried out by pointing directly to the wanted elements (structural or behavioural). A specific GMF (Graphical Modeling Framework)[18] editor, Test Model Editor, encapsulates the language and allows to build the test models graphically. The way the test elements are graphically represented is related to their definition in the defined modeling language. Thus, structural elements are components of different complexity that can be nested in order to provide the test plan with a hierarchy of test elements. An example of structural elements being graphically represented in the editor can be seen in Figure 2. Behavioural elements are inside structural elements and represent basic execution units which will have a direct translation in each target testing platform. Sequences of these elements provide the behaviour of the tests.

Once models are created and the user wants to get executable tests, it is time for transforming the model. The user selects the target platform for executing the tests and the assistant for that platform starts working. This assistant applies a collection of transformation rules, which automatically translate the initial agnostic model into tests which can be executed in the chosen platform. The independent nature of the test models and the existence of specific transformation assistants let the same model be executed in several testing platform, saving time at test design.

Finally, the test models designed can be stored in a regular repository in order to be accessible from the testing framework and from outside. Inside the framework, the models can be accessed from the editor and the transformation assistants for modifications or transformations respectively. From outside, the repository could be accessed by Application Lifecycle Management (ALM) tools for supervision activities. These ALM tools, such as HP-ALM[19] or IBM Rational Team Concert[20], using suitable automatic interpreters, could get the required information directly from test models to fill in their own structures. In this way, migrations from one management tool to another would require less effort since test models are independent from those tools, remaining unaltered during the entire migration process.

This initial framework has been extended with execution models in order to provide more expressivity to the test models. These models are also platform-independent and allow to select which elements from a specific test model will be executed and establish their execution flow. The built of such models is enabled by an Execution Model Editor, while the transformation of these models into executable flows is performed by specific assistants, similarly to the way tests models are managed.

In both cases, the assistants bridge the gap between the agnostic models and the chosen target testing platform, and hide the complexity of the platform. They are also responsible for retrieving relevant information during the execution of the test elements and provide appropriate reports to users.

B. Execution modeling language

The test modeling language used in this framework allows to design hierarchies of structural test elements as a way to organize the test model. However, the language does not provide a way to select which of the test elements will be finally executed in the target platform and the order of that execution. Thus, the user will have to rely on the features included in the target testing platform related to the specification of execution order, if any. This is an important feature since all the executable tests do not need to be executed at once and it is usually interesting to have different execution flows during the testing process. Besides, all the testing platforms do not provide the option to create these flows.

The functionality of the proposed framework has been extended to include the design of execution flows. The aim is to provide more expressivity of test models without increasing the complexity of the test modeling language. For this reason, a new language has been introduced in the framework, the Business Process Model and Notation (BPMN) [21], giving rise to execution models which are still independent from any target testing platform.

 ISBN: 978-1-61208-458-9

BPMN is a standard defined by Object Management Group (OMG) and originally developed to provide a modeling notation comprehensible for all business users, from purely business people to analysts and technical developers. Nowadays, BPMN is widely used in academia and industry due to the large set of concepts managed, independent from any specific domain, and its flexibility to be adapted to other scenarios different from business. This flexibility is due to the fact that BPMN enables the extension of its specification in order to include custom concepts, which represent characteristics of particular domains. The standard is also interesting in terms of execution, since there is a variety of runtime execution engines, which allow the execution of BPMN models almost effortless.

Although the number of concepts managed in the standard is huge (activity, flow, event, gateway, etc.), in this work, we only make use of a subset of them. The chosen elements are available through the developed GMF Execution Model Editor, represented in Figure 1. This editor acts as a filter for the allowed elements, since the standard has not been cut back. The number of BPMN elements managed in the initial subset can be easily increased by simply updating the editor.

Making use of the extensibility feature existing in BPMN to adapt the standard to particular domains, it has been possible to link specific elements in BPMN models to elements defined in agnostic test models. The BPMN element chosen to establish this connection has been the *ServiceTask* element. It represents atomic activities in a process flow, in particular, activities which can be seen as an individual service and can be automated. This description fits well with the purpose of the proposed methodology.

An example of the way these *ServiceTasks* are linked to agnostic test models can be seen in Figure 3, where each of the *ServiceTasks* included in the execution model is pointing to structural elements of different complexity (*TestSuite*, *TestProject*, *TestCase*, etc.) in the test model. Thus, in the figure, the execution model establishes that among all the structural elements existing in the test model for PiggyBank (a simplified online banking application used as case study in this research), only the *TestProject Login* and the *TestSuite Balance* will be executed, showing also the execution order of those elements. The rest of the elements existing in the test model are not required for execution and then, they are not included in the execution model.

An actual example of the Execution Model Editor related to PiggyBank is shown in Figure 4, pointing out the way test elements are linked to BPMN elements.

The execution model shown in Figures 3 and 4 represents a very simplified flow using BPMN. However, the number of concepts and features defined in BPMN, combined with the different concepts defined in the test language allow to introduce different levels of expressivity to the proposed framework.

A first type of expressivity included in the framework, shown in the figures, allows to link *ServiceTasks* only to structural test elements - TestProjects, TestSuites, etc. -. In

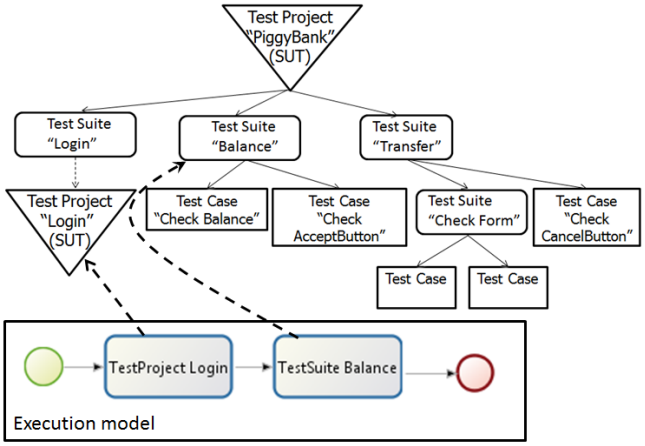

Fig. 3. Extended BPMN specification in order to link to test elements.

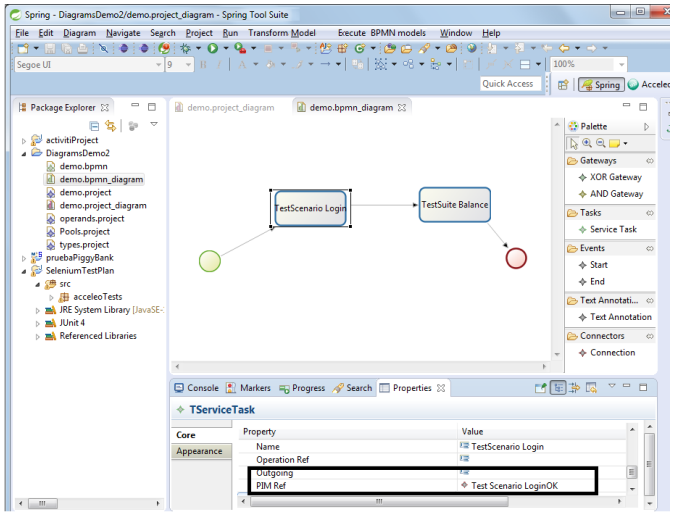

Fig. 4. Example of the developed Execution Model Editor building an execution model related to Piggybank.

these BPMN models, the test elements linked are independent among them in the flow, i.e., the failure of a *ServiceTask* does not imply the suspension of the remaining elements in the model, unless there is a branch in the flow considering that case. In these models, conditional flows can depend on the result of the execution of *ServiceTasks*, seeing this execution as a black box which generates a single result for each *ServiceTask*, independently of the structural test elements involved.

IV. From Modeling Languages to Testing Platforms: A Practical Case

This section shows the way all the elements existing in the proposed framework work together in order to evolve from agnostic test and execution models to executable tests in different platforms.

As shown in Figure 1, users start designing test models for specific applications, ignoring the final platform where tests will be finally executed. These platform-agnostic test models can be enriched with BPMN models, which help users to

 ISBN: 978-1-61208-458-9

define the execution flow of test elements defined in test models. Given the agnostic approach considered in the modeling languages and editors involved in the proposed framework, users can completely ignore any execution platform during the design of test models and execution flows. Only when users want to turn a model into executable tests, test execution platforms are required. Even then, users do not need any technical knowledge about the target platform or the way final tests are built. When a target platform is selected from the proposed framework, the transformation and execution assistants are the only elements responsible for automatically generating tests which are executable in the chosen platform. Both, test models and execution flows, can be transformed to be executed in any available platform.

The mechanisms used to perform the transformation into executable tests are explained next. Before that, the testing platforms managed in the proposed framework are explained.

A. Selenium and GP: testing platforms as case study

Two testing platforms with different characteristics have been considered for test execution. Both suitable for testing web applications, since this is the kind of applications managed in this work. The considered platforms are: Selenium[3], as open source solution; and GP, a proprietary tool suite developed by Santander Group[22].

- Selenium is the selected platform for testing web applications using an open source solution. It lacks of a way to organize tests, being the tester entirely responsible for providing a structural organization. Tests for execution can be written entirely in Java using Selenium WebDriver [23] or as a combination of JUnit tests and Selenium Server [3]. In this proposal, we use the second option in order to simplify the generation of the tests files.
- GP is a proprietary testing tool, which integrates different open source technologies in a common working environment in order to homogenize their usability. It allows the design and automatic execution of tests for web and Eclipse-based applications. Selenium and SWTBot are some of the technologies included.
  From the structural point of view, GP manages elements which provide hierarchical information to organize the tests, allowing individual tests and collections of tests. According to the functionalities included in the platform, it provides a repository of basic behaviours, which can be grouped appropriately to design the each individual test. The technology involved in these behaviours is completely transparent to testers.
  The internal structure of GP tool suite is based on a database technology as storage mechanism for all the elements managed in the platform. This approach can be also found in other testing platforms, such as HP QuickTest Professional (QTP) [1] or IBM Rational Functional Tester (RFT) [24].

Since both platforms build and organize tests in a different ways, the approach to build the transformation engines needed in each case will be also distinct.

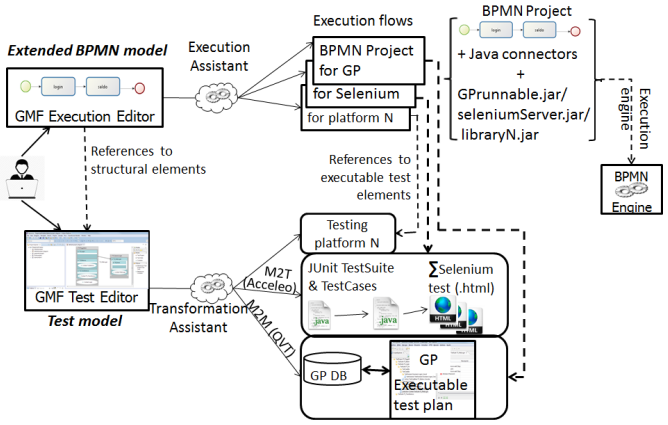

Fig. 5. Transformation and execution assistants link models to testing platforms.

B. Getting executable tests

Following the proposed workflow, and assuming that test models are ready to be transformed into executable tests, users have two ways to continue working. Both are based on transformation and execution assistants, whose role in the testing framework is summarized in Figure 5. This figure also sums up the different technologies and languages used in the implementation of the developed prototype.

1) Transform an entire test model: The user can opt for transforming an entire agnostic test model into a collection of equivalent tests to be executed in a specific testing platform. In this use case, generically represented in the lower part of Figure 5, the user selects the agnostic test model that needs to be transformed and decides the specific testing platform among the options where tests will be executed. Having this information, the assistant delegates the task to the specific transformer for the chosen platform. This assistant provides flexibility to the proposed framework, isolating the editors and users from any testing platform.

Each transformer knows the internal structures of its target platform and the agnostic testing language, establishing the correspondence between the elements in the target platform and the agnostic elements. The existence of this correspondence makes possible that, in an automatic way, the transformer applies to the test model the appropriate rules to generate an equivalent collection of tests in the target testing platform. In this process, the elements existing in the test model (structures, behaviours, data, etc.) are mapped into their corresponding elements, creating all the necessary structures to reproduce the test model in the executable target platform. The transformer not only translates test models, but also interprets them. This is specially relevant referring to data, which potential complexity (unique/multiple values, complex structures, etc.) can generate a single test or a collection of tests as a result of all the possible combinations.

The implementation of each transformer will depend on how the target platform can be represented. If it can be represented as an Eclipse Modeling Framework (EMF) model, model-to-

 ISBN: 978-1-61208-458-9

model transformations can be performed. Otherwise, model-to-text transformations or other alternatives will be necessary. Considering GP tool, based on a database, the tool can be represented by its own modeling language inferred through the database schema. According to this, the content of the database in a particular moment is the state of the system, a state that can be represented as a model. This model enables the use of model-to-model transformations to implement the transformer to GP, using the operational Query/View/Transform (QVT) language. Instead, transforming to Selenium requires model-to-text transformations, using Acceleo, due to the lack of a modeling language to represent Selenium and JUnit.

2) Execute a BPMN flow: Users that want to focus just on the execution of a portion of an agnostic test model can design execution flows (upper part of Figure 5). Before executing the flow, the testing platform where tests will be executed has to be selected. Similarly to test models, BPMN models are not executable straightaway, being also necessary an assistant. When the assistant knows the target platform, it delegates the task to the appropriate interpreter, which will check whether the agnostic model associated with the BPMN model has already been transformed and exists in the chosen platform. Otherwise, the corresponding transformation assistant will be performed first. After that, the interpreter will face two tasks before triggering the execution of the BPMN model automatically. First, connecting the BPMN model to a BPMN engine, which reads, interprets and executes the model. This connection requires a suitable infrastructure (called BPMN project in Figure 5), which varies depending on the chosen BPMN engine. Second, since *ServiceTasks* in the proposed extended BPMN models point to agnostic test elements, it is mandatory their translation to references to executable tests which exist in the target testing platform. For this second task, it is worth to mention that during the transformation process of a test model, some extra data can also be generated if needed. The need for this auxiliary information depends on how costly is to associate in a BPMN model an agnostic test element to its executable tests once the test model has been transformed. This association can be based on one-to-one relationships, but it is mostly based on one-to-many relationships because of two reasons. First, since the structural element referenced in a *ServiceTask* can be a container of other tests elements, the BPMN engine will need to call for execution to each of the individual tests that the transformation process generates from the container element. For instance, *ServiceTask* in Figure 3 is referencing the complex element *TestProject Login*, represented in Figure 6.a. All the tests contained need to be known. Second, the agnostic models manage collections of data for specific inputs of the tests. These collections give rise to a set of individual tests where each input only has one value associated with it. This is the case represented in Figure 6.b, where one single test element, *TestCase Button_Accept*, is transformed into a variable number of executable tests depending on the data defined in it.

According to this, executing BPMN models in the GP platform, requires an intermediate model as extra information in order to associate the structural tests elements in the agnostic model to the identifiers of their corresponding executable tests generated in the target platform. However, using Selenium as target platform does not require extra information, since the association between test elements and executable tests is easier to obtain.

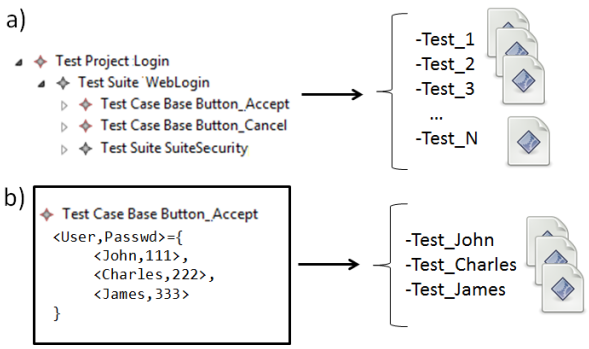

Fig. 6. Model-to-model transformations usually turn into one-to-many relationships that are managed through the intermediate model.

Once the BPMN model points to executable tests, it is time to establish the infrastructure to make the flow executable. This infrastructure depends exclusively on the BPMN engine chosen for execution. In this work, it has been used the BPMN engine provided by Activiti [25], which requires an Eclipse project to execute the BPMN models. Apart from the designed BPMN model, the project must contain information to deploy it for execution and set up the BPMN engine. It also includes the way to implement the execution of the *ServiceTasks* existing in the BPMN model. This implementation codifies the way all the executable tasks associated to a particular *ServiceTask* are identified and called for their automatic execution one by one. In this process, it is used any extra information that might have been generated during the transformation of its corresponding test model. As can be seen in Figure 5, the execution of the tests is provided by platform-dependent libraries (GP runnable, selenium-server), which allow an automatic interaction with each platform. The Activiti-based project is the output of the interpreter when a user wants to execute a BPMN model, and the rules to generate the involved Java classes are based on model-to-text transformations, using as model the test model itself (for Selenium platform) or the intermediate model (for GP platform). Once the building of the Eclipse project is complete, the interpreter initiates its execution, triggering the BPMN engine which reads the BPMN model, interprets each one of the existing elements, and performs the actions associated with each one. When the BPMN engine finds a *ServiceTask*, all its associated tests are launched for execution before continuing with the flow.

V. Conclusion and Future Work

The proposed model-based testing framework, the prototype of which has been entirely implemented, provides a infrastructure to design graphically test and execution models

 ISBN: 978-1-61208-458-9

independent from any testing platform. The execution models represented as BPMN flows, increase the expressivity of the test models by extending BPMN elements to include custom concepts related to the testing language.

The proposed framework allows a quick starting of the testing phase, being able to work in parallel to the development process, and a quick adaptation to changes in the requirements. It also enables the reuse of test elements among models. The models are only attached to an application, but can be applied to very different platforms for test execution due to the assistants provided in the framework, which automatically transform models to executable tests in each particular platform. The use of graphical editors and assistants allow to hide the complexity of testing execution platforms, increasing the number of potential users of the framework.

As future work, the expressivity of the tests will be increased by including more BPMN concepts to the ones already managed by the editor, and the implementation of complex behavioural elements based on BPMN flows. The platforms managed currently in this framework are Selenium as open source solution, and GP as proprietary tool. Other platforms can also be integrated building their corresponding assistants. Thus, the integration of tools, such as HP Quality Center (HP QC), also database-based as GP platform, could be affordable in the near future.

ACKNOWLEDGEMENT

The work for this paper was partially supported by funding from ISBAN and PRODUBAN, under the Center for Open Middleware initiative [26].

REFERENCES

[1] A. Rao, HP QuickTest Professional WorkShop Series: Level 1 HP Quicktest. Outskirts Press, 2011.
[2] P. Tahchiev, F. Leme, V. Massol, and G. Gregory, JUnit in Action. Greenwich, CT, USA: Manning Publications Co., 2010.
[3] A. J. Richardson, Selenium Simplified: A Tutorial Guide to Selenium RC with Java and JUnit. Compendium Developments, 2012.
[4] A. Huima, "Implementing Conformiq Qtronic," in TestCom/FATES, ser. Lecture Notes in Computer Science, A. Petrenko, M. Veanes, J. Tretmans, and W. Grieskamp, Eds., vol. 4581. Springer, 2007, pp. 1–12.
[5] Hewlet-Packard Company, "HP Functional Testing," 2012.
[6] P. Baker et al., "The UML 2.0 Testing Profile," Sep. 2004.
[7] J. Iber, N. Kajtazovic, A. Holler, T. Rauter, and C. Kreiner, "Ubtl - UML Testing Profile based Testing Language," in Model-Driven Engineering and Software Development (MODELSWARD), 2015 3nd International Conference on. SciTePress, February 2015, pp. 99–110.
[8] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2007.
[9] M. Cristiá and P. R. Monetti, "Implementing and Applying the Stocks-Carrington Framework for Model-Based Testing," in ICFEM, 2009, pp. 167–185.
[10] L. C. Briand and Y. Labiche, "A UML-Based Approach to System Testing," in Proc. of the 4th Int. Conference on The Unified Modeling Language, Modeling Languages, Concepts, and Tools. London, UK, UK: Springer-Verlag, 2001, pp. 194–208. [Online]. Available: http://dl.acm.org/citation.cfm?id=647245.719446
[11] C. Pedrosa, L. Lelis, and A. Vieira Moura, "Incremental testing of finite state machines," Software Testing, Verification and Reliability, vol. 23, no. 8, 2013, pp. 585–612. [Online]. Available: http://dx.doi.org/10.1002/stvr.1474
[12] K. Karl, "GraphWalker," URL: www.graphwalker.org [accessed: 2015-12-18].
[13] M. Fowler, UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003.
[14] B. P. Lamancha, P. R. Mateo, I. R. de Guzmán, M. P. Usaola, and M. P. Velthius, "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, ser. MoDeVVa '09. New York, NY, USA: ACM, 2009, pp. 1–10. [Online]. Available: http://doi.acm.org/10.1145/1656485.1656491
[15] M.-F. Wendland, A. Hoffmann, and I. Schieferdecker, "Fokus!MBT: A Multi-paradigmatic Test Modeling Environment," in Proc. of the Workshop on ACadeMics Tooling with Eclipse, ser. ACME '13. New York, NY, USA: ACM, 2013, pp. 1–10. [Online]. Available: http://0-doi.acm.org.cisne.sim.ucm.es/10.1145/2491279.2491282
[16] P. Baker et al., Model-Driven Testing: Using the UML Testing Profile. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2007.
[17] C. Sanz et al., "Automated model-based testing based on an agnostic-platform modeling language," in Proceedings of the 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), 2015, pp. 239–246.
[18] Eclipse(b), "Eclipse Graphical Modeling Framework (GMF) Tooling," URL: http://www.eclipse.org/gmf-tooling/ [accessed: 2015-12-18].
[19] Hewlet-Packard Company, "HP Application Lifecycle Management," 2011.
[20] International Business Machines Corp., "IBM Rational Team Concert," 2008.
[21] OMG, Business Process Model and Notation (BPMN), Version 2.0, Object Management Group Std., Rev. 2.0, January 2011, URL: http://www.omg.org/spec/BPMN/2.0 [accessed: 2015-12-18].
[22] URL: www.santander.com/ [accessed: 2015-12-18].
[23] S. Avasarala, Selenium WebDriver Practical Guide. Packt Publishing, 2014.
[24] C. Davis et al., Software Test Engineering with IBM Rational Functional Tester: The Definitive Resource, 1st ed. IBM Press, 2009.
[25] T. Rademakers, Activiti in Action: Executable business processes in BPMN 2.0, 1st ed. Shelter Island, NY: Manning Publications, 2012.
[26] URL: http://www.centeropenmiddleware.com/ [accessed: 2015-12-18].

 ISBN: 978-1-61208-458-9