Using Model-Based Testing for Testing Application Models in the Context of Domain-Specific Modelling

Janne Merilinna
VTT Technical Research Centre of Finland
P.O. Box 1000,
02044 Espoo, Finland
+358 442 788 501
janne.merilinna@vtt.fi

Olli-Pekka Puolitaival
VTT Technical Research Centre of Finland
P.O. Box 1100,
90571 Oulu, Finland
+358 400 606 293
olli-pekka.puolitaival@vtt.fi

ABSTRACT

Domain-Specific Modelling (DSM) has evidently increased productivity and quality in software development. Although productivity and quality gains are remarkable, the modelled applications still need to be tested prior to release. Although traditional testing approaches can be applied also in the context of DSM for testing generated applications, maintaining a comprehensive test suite for all developed applications is tedious. In this paper, the feasibility of utilizing Model-Based Testing (MBT) to generate a test suite for application models is studied. The MBT is seen as a prominent approach for automatically generating comprehensive test cases from models describing externally visible behaviour of a system under testing (SUT). We study the feasibility by developing a domain-specific modelling language and a code generator for a coffee machine laboratorial case and apply MBT to generate a test suite for the application models. The gathered experiences indicate that there are no technical obstacles but the feasibility of the testing approach in large-scale models and languages is still questionable.

Categories and Subject Descriptors

D.2.5 [**Testing and Debugging**]: Testing tools

General Terms

Experimentation and Verification

Keywords

Model-driven development; Verification; Test generation

1. INTRODUCTION

Quite often Domain-Specific Languages (DSL) and Domain-Specific Modelling Languages (DSML) are mentioned to attain 5-10 fold productivity gains compared to traditional software development practices [1]. The productivity increase is primarily caused by the Domain-Specific Modelling (DSM) basic architecture, i.e. DSML, a code generator and a domain-specific software framework. It is also often argued that utilization of DSM increases software quality by decreasing programs errors among other things [1].

While well-defined DSML promotes modelling of correctly defined applications and in this way decreases program errors, the ultimate reason for productivity gains and the decrease of program errors is achieved via automation. Code generators are responsible for systematically transforming application models to source code on the target platform. While code generators systematically transform application models to source code, they also systematically produce program errors. The difference between code generation and manually transforming software specification to implementation is when program errors produced in code generation look the same and there are many of the same kinds of errors, manually transforming software specifications to implementation results in various kinds of errors.

From a testing point of view, the difference is that it is always easier to pinpoint an error which emerges frequently and systematically compared to errors emerging in various parts and in various shapes in the source code. This has a direct impact on source code quality. While applications produced without code generation need to be corrected one at a time, all errors found and corrected in code generators contributes to the overall quality of the whole product family.

Although it should be easier to find errors produced by code generators, locating all errors in code generators is not a trivial process. It is highly unlikely that all paths in code generators are traversed every time the source code is generated therefore errors do not reveal themselves easily. Similar to traditional application testing, improving the level of quality of code generators requires extensive test suite. In the case of code generators the test suite is a set of application models similar to compilers in traditional software development where source code is an input to the compiler. Thus, to improve the quality of code generators requires an extensive set of application models. In [1], iterative and incremental DSM, the development approach is argued to produce DSMLs with code generators of good enough quality. In our earlier work [2] it was argued that a more systematic approach is required as the iterative and incremental development approach may not produce an extensive enough test suite for code generators. Therefore an approach to produce an extensive set of application models as a test suite is required.

In [2], we presented a concept for testing the whole DSM basic architecture. The approach consists of two phases: 1) generating application models from a metamodel with an approach of Model-Based Testing (MBT) [3], and 2) generating a test suite for generated application models with MBT. In this paper, we further elaborate the second phase and demonstrate the approach in a laboratorial case study. We gather the experiences in testing application models with MBT in a laboratory case for a coffee machine for which a DSML and Python source code generator were developed.

This paper is structured as follows. First, the principles of DSM and MBT are discussed to set a background and baseline for our work. Second, utilizing MBT as a means for testing application models is presented. Third, the application testing approach is demonstrated in a laboratory case involving a coffee machine. Discussion and conclusions close the paper.

2. BACKGROUND

To get an understanding of the DSM testing approach under scrutiny, the background of DSM basic architecture and MBT needs to be known. Next, in this section, the background of DSM and MBT are discussed.

2.1 Domain-Specific Modelling

Increasing productivity in software development is largely dependent on software reuse and automation. Often the work required to increase productivity follows the same pattern as Roberts et al. present in [4] when reusability is considered. First, a couple of example applications are developed according to traditional means. The applications of same product family share a set of components that can be reused in the product family. When the amount of reusable components increases, white-box and black-box frameworks begin to emerge. While the frameworks mature, the application development increasingly shifts from low-level programming to utilization of the developed framework. Ultimately, the development of applications may be about choosing different alternative features from a pre-defined feature tree. In the case where there is a considerable number of variation and neither feature-trees nor wizards can be utilized, domain-specific languages (DSL) and DSMLs start to emerge.

A DSM solution consists of three main parts, often described as DSM basic architecture [1]:

- A metamodel defines the syntax of a modelling language. In the case of DSMLs, a metamodel mirrors the problem-space by providing modelling elements found directly from the problem domain. In practice, the metamodel also includes elements and restrictions of the target platform.
- Code generators define the transformation rules on how to transform application models that are based on a metamodel to a source code representation.
- The software framework abstracts low-level details of the target platform and functions as a platform on which a code generator generates source code. Sometimes no framework is required and the generated code directly accesses the services and functions of the target platform.

Actual applications are modelled based on the model elements and constraints of the developed metamodel. The models can be transformed into source code or any given representation with generators.

2.2 Model-Based Testing

The MBT is a black-box software testing method in which test cases are automatically generated from a model describing the behaviour of a system under testing (SUT) [3]. The MBT consists of three phases, i.e. modelling, test generation and test execution.

In the modelling phase, behaviour of the SUT is modelled according to specifications of the SUT where functional requirements are the primary source for developing the MBT models [3]. The MBT being a black-box testing method, the MBT models are required to embody the externally visible behaviour of the SUT, i.e. input and output data of the SUT. The input data is used for executing the tests and output data for verifying the tests. The notation of the models can be graphical, textual or mixed where the notation varies from general purpose to domain-specific [5].

Test generation is based on model traversal where several test design algorithms are utilized for generating test cases from the model. For offline testing [6], i.e. generating a test suite first and then executing it, there are two categories of test design algorithms i.e. requirement-based criteria and coverage criteria [7]. Requirement-based criteria test design algorithms are based on model traversal algorithms that traverse the MBT models until all required parts of the model are visited. Coverage criteria test design algorithms aim to traverse the MBT models until a required coverage criteria is fulfilled. For online testing [6] walking test design algorithms are utilized [7]. In the walking test design algorithm approach, each subsequent test step is decided after executing a preceding test step. It must be noticed, that the coverage of the test suite can only be as extensive as the model describes, i.e. parts of the program behaviour not described in models are not tested.

In the test execution phase, the generated test suite is executed against the SUT. As the software implementation is developed from the same specification as the MBT models, two opinions regarding the behaviour exist. The difference between these opinions is seen as errors during test execution.

The main benefits of the MBT are the facilitation of test suite maintenance and the coverage of the test suites. The facilitation of the test suite is based on the supposition that only MBT models are required to be kept up-to-date when the SUT evolves and the test suite can always be updated via test generation. The increased coverage is based on sophisticated test design algorithms that are the result of long time research. [3]

3. TERMS OF UTILIZING MBT FOR TESTING APPLICATIONS

Testing of applications in the context of DSM can mean the following aspects when the utilized metamodel restricts modelling of incorrect application models:

- Does the modelled application satisfy its functional and quality requirements, i.e. is the application modelled correctly and according to specifications?
- Does the correctly modelled application model transform to a source code representation correctly?
- Does the correctly-modelled generated application function as modelled when executed?

In this paper, we concentrate on the two latter aspects. Thus we assume that the application models are always correctly defined and the reason for failure is always caused by either:

1. failure in code generation alone,
2. platform failure alone, or
3. a combination of the preceding.

Considering 1), we do not strive for white-box testing and we do not consider source code inspection but rather strive for black-box testing. Thus in this paper we solely concentrate on testing how the generated code integrated with the software platform function as a combination, i.e. black-box testing.

Such as presented in Section 2.2 the failure of a test is caused by an incorrectly implemented application or MBT model. Considering DSM, the application model is always correct (according to our terms) therefore the failure can be caused by F1)

failure in code generation, F2) platform, F3) a combination of the preceding or F4) incorrectly defined MBT models.

If we apply MBT out of its initial purpose and generate the test suite directly from a formal specification (see illustration in Figure 1, see also [2]), i.e. an application model from which source code is also generated, we no longer compare during test execution whether the modelled application is performing according to the specifications as we take the application model as a fact. This in no way contradicts our initial terms.

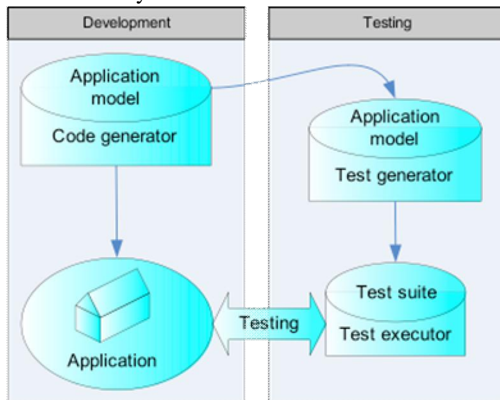

Figure 1. Using MBT for testing application models.

Now, as the same model is used as an input for code generation and test generation, we can rule out failure in F4 thus only F1-3 remain. F2 is also ruled out from discussion in this paper as it can be done using traditional testing approaches. Therefore a failure in code generation is the only thing remaining when the following terms are true:

- The application model is defined according to software specifications,
- The application model is correctly defined considering the utilized metamodel, and
- The test suite generated from application models is always correct.

4. A COFFEE MACHINE AS A LABORATORY CASE

To gather experiences and the technical limitations of utilizing the MBT in testing code generators in the context of DSM, a laboratory case example involving a coffee machine is utilized. The purpose of such machines is to take coffee orders as an input and deliver coffee as an output. There are also different kinds of machines where some are equipped with displays of various types and some machines require a different amount of money as an input whereas some make coffee for free. Nowadays there are also various blends of coffees available in addition to the basic combination of coffee and cream. Some of the most special coffees even have a very delicate preparation procedure thus producing a cup of coffee might be more than just the simplest procedure.

4.1 Tools for the Laboratory Case

As a language workbench for developing DSMLs, code generators and application models, MetaCase MetaEdit+[1] was chosen. MetaEdit+ includes tools to define DSMLs with GOPPRR (Graph-Object-Property-Port-Role-Relationship) metamodelling language and generators with MetaEdit+ Reporting Language (MERL) in addition to providing basic modelling facilities.

For the MBT of application models, there are two different approaches:

- develop test design algorithms within MetaEdit+ environment and generate a test suite by applying the developed generator, or
- take advantage of existing MBT tools.

The first approach requires implementing test design algorithms with MERL. The second approach requires exporting application models developed with MetaEdit+ to an external MBT tool. Exporting an application model requires implementing a model transformation specific to a metamodel with MERL. We chose the latter approach as developing a set of test design algorithms was anticipated as non-trivial and troublesome and we have had good experiences with MBT tools such as Conformiq Qtronic[2] (CQ), which was also chosen based on our evaluation presented in [7].

CQ expects the Qtronic Modelling Language (QML) as an input. QML is a variant of the Unified Modeling Language (UML) State Machine Diagram where as an action language a variant of Java is utilized. The action language is utilized for describing expected input and output values. From QML, CQ is able to generate test cases by applying a few coverage algorithms. CQ provides two pre-developed test scripters, which are used for generating Testing and Test Control Notation version 3 (TTCN-3) and Hypertext Markup Language (HTML). The TTCN-3 scripter produces a test suite described in TTCN-3 which can be used in test execution platforms. HTML scripter produces a UML Sequence Diagram as an illustration of the test cases. In addition, CQ provides a plug-in interface for the development of custom scripters.

4.2 Coffee Machine Modelling Language

For the coffee machine in question, Coffee Machine Modelling Language (CMML) was implemented with MetaEdit+. The CMML consists of two sub-languages where the first (see left hand side of Figure 2) is a User Interface Modelling Language (UIML) and the second language is a Coffee Making Process Modelling Language (CMPML) (see right hand side of Figure 2). The UIML enables testers to model the users interface (UI) of the coffee machine where the following aspects can be modelled: available sorts of coffee, the cost of a cup of coffee of a chosen sort, and textual information which is displayed to the user. The CMPML includes concepts for modelling e.g., heating a certain amount of water that is poured through a certain amount and blend of coffee to a cup of various sizes. Milk, cream, sugar etc. can be added when desired and foaming can be applied when required.

[1] www.metacase.com

[2] www.conformiq.com

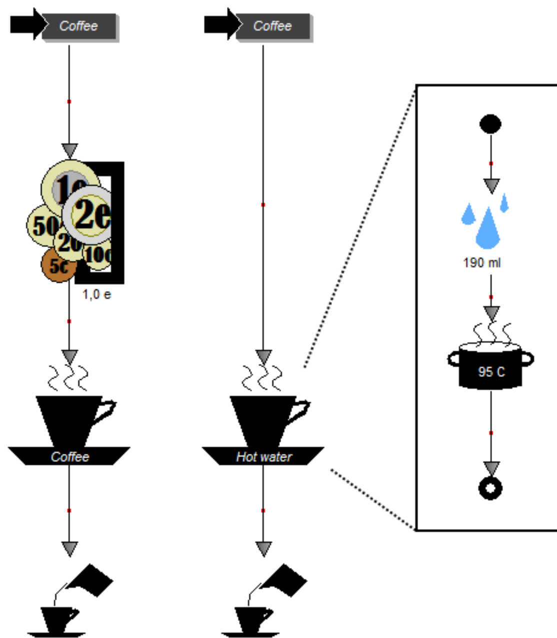

Figure 2. Simple coffee machine modelled with Coffee Machine Modelling Language.

For CMML, a Python source code generator was developed which produces complete code from models in the sense that there is no need to modify the generated code. The generated code enables simulation of the coffee machine behaviour in a desktop computer environment. The code generator is also able to produce a debug version of the modelled application in addition a stand-alone version. The debug version utilizes Simple Object Access Protocol (SOAP) Application Programming Interface (API) of MetaEdit+ to report a change of a state during the application execution back to the model where the application was generated. The change of a state is shown as a red rectangle highlighting the currently active state in the application models. This feature enables graphical debugging of the application.

4.3 Testing the Coffee Machine by Using MBT

The generated code follows an architectural division of UI, control logic (CL) and cooking machine interface (CMI). The UI is responsible for taking coffee orders and money as an input, and displaying information to the customer. The UI provides orders for CL to start making the ordered coffee. The CL is responsible for controlling the preparation process and it closely collaborates with CMI which simulates the physical hardware responsible for performing various preparation- related tasks. All components are implemented as Python threads to simulate concurrent processing and asynchronous events that are common to embedded devices. The preceding components have clearly defined interfaces to promote testability. No separate domain-specific framework exists because of the simplicity of the application domain.

In this laboratory case study, testing the behaviour of the CL is demonstrated. As discussed above, the CL has an interface to the UI and CMI. The provided interface of CL towards the UI consists of two kinds of signals i.e. pressing the coffee ordering button with the value of an ordered blend of drink, and the value of coin. The required interface towards the UI consists of text to the display signal. The required interface of CL towards CMI consists of the order signal with a value either to add water, add coffee, add milk, add cocoa, add cream, add sugar, warm water, and serve the coffee signal. The provided interface consists of a response signal with an ok/fail value as a parameter.
From an implementation perspective, i.e. after code generation, the signal exchange between the components in this example application is as follows when a customer orders a hot water product:

- CL receives a selected blend of a drink from UI.
- CL sends an add water order to CMI which immediately starts pouring the water to a heater.
- After CMI has finished pouring the water, it notifies the CL about the finished task.
- After receiving the pouring complete message from CMI, the CL notifies CMI to heat the water to 95 degrees.
- After CMI finishes heating the water, it notifies the CL.
- After receiving the heating complete message from CMI, the CL notifies CMI to serve the coffee.

4.3.1 Model Transformation

CQ expects QML as an input. As the metamodel of the CMML and the QML are different, a model transformation from CMML to QML is required. The model transformation can be divided into two main steps, i.e. transformation of the CMML objects and relationships into the QML state machine, and transformations of the information contained by CMML objects into QML input and output signals.

In the case of transforming the coffee machine application model to QML where CL is the SUT, the transformation is as follows. First, the QML state machine is initiated by generating QML Start, End and Idle states. The Start state has a transition to the Idle state. The Coffee Pressing Button objects (see the topmost entities in Figure 2) transform into QML transitions between the Idle and the Coffee decomposition states (see below). Serving the Coffee objects (see the lowest entity in Figure 2) transform into QML states and transitions to the End state. The Coin Input object transforms into a looping state which loops until the correct amount of coins is received. Transformation of Coffee objects depends on their decomposition. Transition to Coffee objects transform to a transition to the first object in a decomposition graph, and the transition from the Coffee object transforms to a transition leaving from the last object in a decomposition graph.

Objects of CMPML transform to QML states. Transitions entering to objects in CMPML transform to QML action transitions i.e. transitions that trigger an action represented by the connected object. Transitions leaving the CMPML objects transform to QML triggering transitions. Input and output values for QML action and triggering transitions are generated by considering values and types of CMML objects.

After the model transformation, the model is in the required format. The state machine part of QML is described in XML Metadata Interchange (XMI) format and action language for input and output transitions in QML. The result of the transformation does not include graphical presentation, however, for illustration purposes the transformed model can be manually visualized as shown in Figure 3.

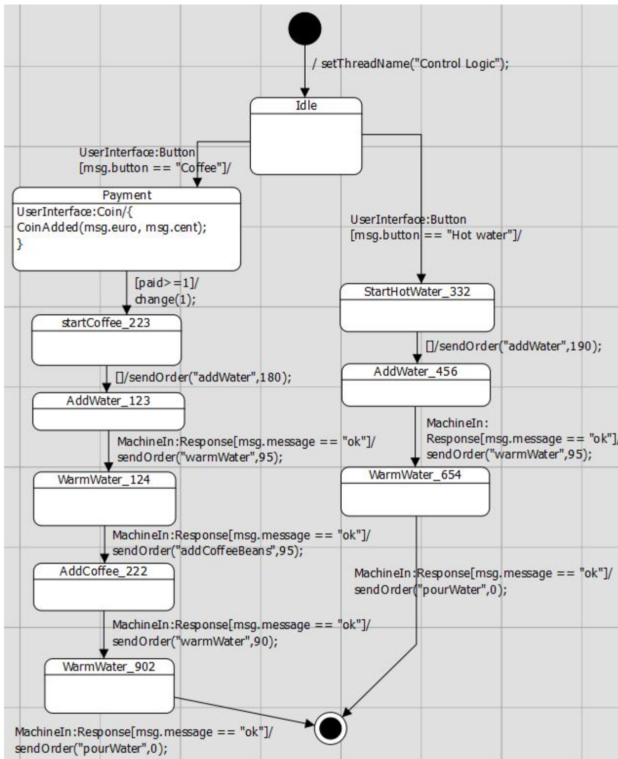

Figure 3. Qtronic model.

In Figure 3, the state machine part of the externally visible behaviour of the CL is presented. Input and output QML code is omitted in the figure for the sake of clarity. The left side of the figure represents ordering the coffee product and the right side represents ordering the hot water product.

4.3.2 Test Suite Generation

After the model transformation, a test engineer can choose which test design algorithms are to be utilized from the algorithms provided by the CQ. In this laboratorial example, transition and state coverage test design algorithms were chosen. The transition coverage test design algorithm generates a test suite in which each state of the model is visited at least once. The state coverage test design algorithm is similar to the transition coverage test design algorithm but visits all states. As an output format, HTML was chosen which is one the pre-made scripters provided by CQ. Now, CQ generates three test cases where one of the test cases is depicted in HTML format as in Figure 4.

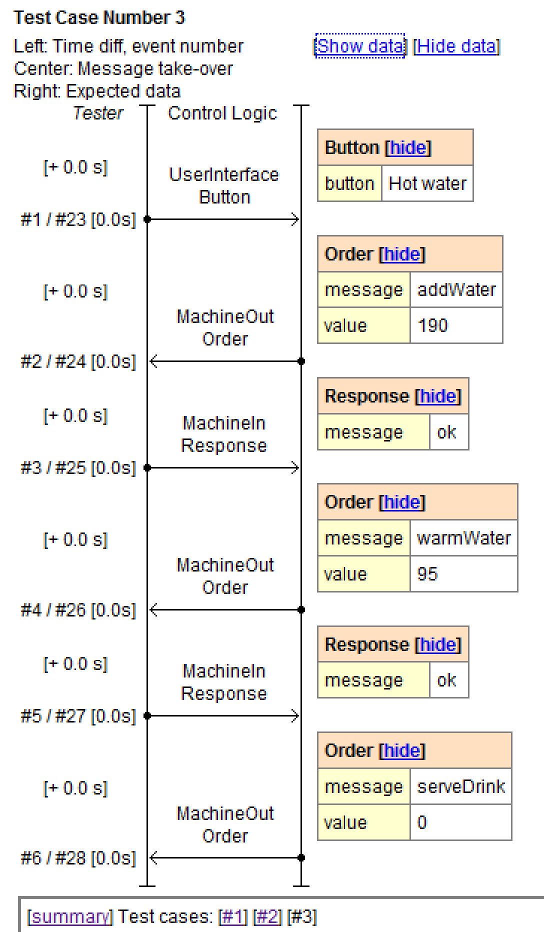

Figure 4. An illustration of a generated test case.

In Figure 4 a message exchange between the CL and the Tester is illustrated. As shown, CL receives *Button pressing event* with "*hot water*" parameter as an input from Tester, to which the CL reacts by sending *Order* with parameters "*addWater*" and "*190*" to Tester. The Tester reacts by sending *Response* with "*ok*". After that, the CL sends *Order* with parameters "*warmWater*" and "*95*" to Tester which again replies with "*ok*". After the CL receives the "*ok*" signal, it sends "*serveDrink*" with value "*0*" *Order* to the Tester which closes the test case. Now, if an error occurs when this test is executed, it is shown as a discrepancy of the expected output values.

5. DISCUSSION

For the feasibility study in utilizing MBT for testing applications developed with DSML for a coffee machine, MetaCase MetaEdit+ was chosen as a DSM environment whereas Conformiq Qtronic was chosen as an MBT tool. To enable the test generation, application models export from MetaEdit+ to QC was required. As the metamodels between MetaEdit+ and QC are different, a model transformation was required to transform CMML to QML, which is the format required by QC.

In the coffee machine laboratory case, the model transformation was trivial to implement as the mapping between CMML and the QML is straightforward. However, this might not be the case in real and perhaps more complex languages than the

CMML. As the MBT is completely dependent on the accuracy of the source model, even the slightest variation between the modelled behaviour of the application models and the MBT models ruins the test accuracy. In the case of CMML, model transformation was able to be validated by visualizing the MBT model but this again might not be possible when more complex languages are considered. It might be so that the chosen approach to utilize an external MBT tool might not be the perfect choice as the verification might just shift from testing the generated applications to testing the model transformations from the DSM to MBT environment. However, it must be noticed that such transformation has to be developed only once per language.

Another approach to utilize MBT is to replace an external MBT tool with test design algorithms developed directly in the DSM environment. This removes the need for model transformation but requires implementing the test design algorithms. Whereas by utilizing existing MBT tools that provide extensively-verified test design algorithms, now the test design algorithms have to be implemented for every language and a question about the quality of custom algorithm emerges. Currently, a trade-off when to utilize an external MBT tool compared to developing the test design algorithms by itself is a matter of debate and should be scrutinized before attempting to use MBT for testing a complete DSML, which is our ultimate target.

6. CONCLUSION

Industry cases constantly attain 5-10 fold productivity gains compared to traditional software development practices when DSMLs with full code generation is applied in software development. Not only are productivity gains witnessed but also quality is increased in the sense of decreased program errors. The increase of quality is partially explainable by well-developed modelling languages which prohibit the design of incorrect models but also because code generation has a remarkable impact on quality.

Although the quality increase is evident, software products cannot be released without proper testing without knowing that the modelling infrastructure, i.e. metamodels and code generators, is flawless. Iterative and incremental development of the modelling infrastructure is a state of the practice approach means to produce quality languages but still there is uncertainty about the quality without systematic and extensive testing of the whole infrastructure. Without such systematic and extensive testing methodology the resulting applications still need to be tested.

The contribution of this paper is a feasibility study of applying MBT to test the generated applications. While traditionally MBT models are developed from software specification parallel to implementation, we strive for applying the MBT to generate a test suite directly from domain-specific application models. In this way, the test suites are always up-to-date with the application models. In this paper, we demonstrated the utilization of MBT to generate test suites from application models in a laboratorial case study of a coffee machine for which DSML and Python code generator were developed. As a conclusion, the test generation seems to be technically feasible but it is still unknown if the approach is also feasible with more complex modelling languages.

7. REFERENCES

[1] Kelly, S. and Tolvanen, J-P. 2008. Domain-Specific Modeling: Enabling full code generation, John Wiley & Sons, ISBN 978-0-0470-03666, 427p.

[2] Merilinna, J., Puolitaival, O.-P. and Pärssinen, J. 2008. Towards Model-Based Testing of Domain-Specific Modelling Languages, The 8th OOPSLA Workshop on Domain-Specific Modeling, Nashville, TN, USA.

[3] Utting, M. and Legeard, B. 2006. Practical Model Based Testing: A Tools Approach, Morgan Kaufmann 1st ed., ISBN: 978-0123725011, 456p.

[4] Roberts, D., Johnson, R. 1996. Evolving Frameworks: A Pattern Language for Developing Object-Oriented Frameworks, Proceedings of Pattern Languages of Programs Vol. 3 (1996).

[5] Hartman, A., Katara, M. and Olvovsky, S. 2006. Choosing a test modeling language: A survey, Haifa Verification Conference, pp. 204-218.

[6] Utting, M., Pretschner, A. and Legeard, B. 2006. A taxonomy of model-based testing, Working papers series. University of Waikato, Department of Computer Science, Hamilton, New Zealand, University of Waitako.

[7] Puolitaival, O.-P., Luo, M. and Kanstren, T. 2008. On the Properties and Selection of Model-Based Testing tool and Technique, 1st Workshop on Model-based Testing in Practice (MoTiP 2008), Berlin, Germany.