# A Process to Increase the Model Quality in the Context of Model-Based Testing

Vladimir Entin, Mathias Winder, Bo Zhang, Andreas Claus
Software Quality Assurance
Omicron electronics GmbH
Oberes Ried 1, A-6833 Klaus, Austria
{name.surname}@omicron.at

***Abstract*—In the past years model-based testing (MBT) has become a widely-used approach to the test automation in the industrial context. Until now the application of MBT has been limited to the software quality engineers with very good modelling skills. In order to guarantee the completeness of a model and to increase its precision there is a need to allow the usage of the approach by other project stakeholders such as requirements engineers as well as software quality engineers with a limited modelling experience. In this contribution we share the challenges discovered during the several years of the application of a certain MBT technique in a SCRUM project with particular regard to the definition of precise and complete models. A process which involves the entire software project team into the model definition starting at the very early stages of product development is presented along with its concrete implementation. First experiences with the application of the process in a particular project are presented.[1]**

***Keywords—model-based testing; domain-specific languages; requirements engineering; SCRUM; maintainability***

## I. INTRODUCTION

OMICRON is an international company serving the electrical power industry with innovative testing and diagnostic solutions. The application of OMICRON products allows to assess the condition of the primary and secondary equipment with complete confidence. Services in the area of consulting, commissioning, diagnosis and training make the product range complete. The software written for our measurement devices supports the user during diagnostic testing for analyzing condition of various assets.

In the past four years we made research in the area of model-based testing, conceived, implemented and leveraged an approach [1,2] that allows us to easily maintain the automatically generated test cases in such agile software development process as SCRUM [3]. The test cases generated are both graphical user interface (GUI) and application programming interface (API) tests. One of the main advantages of this approach is that test cases are derived from usage models so that they can be completely executed by a test runner without any further adjustment by a test engineer.

Since one of the primary tasks of quality assurance is to guarantee the proper functionality of the product from the customer's perspective, integration along with system tests by means of black-box techniques are indispensable. Usage models allow the definition of possible navigation paths through the software under test from the customer's perspective.

In this contribution we present the practical challenges encountered while deriving the usage models from the textual requirements specifications. Moreover, we introduce a process which increases the quality of the derived models since it makes their definition more precise and complete. A model is considered to be complete when it covers all specified requirements and its degree of formality allows the automated derivation of test cases. The process also allows to speed up the usage model definition by involving not only software test engineers working in the software quality assurance (SWQA) with a broad experience in modelling but also other project stake holders from the very early stages of the product development long before any test case specifications have been defined. The article may be of interest both to the academic researchers interested in industrial application of MBT approaches and industrial MBT practitioners who want to increase the usage model quality and to speed up its derivation from the textual requirements. In the following we list the topics covered by the paper.

In a SCRUM environment the usage model definition is a manual task which involves interpretation of the requirements by SWQA. Since in the current practice software test engineers are the only project stakeholders to define a usage model, the peril of misinterpreting or omitting the requirements is high. In this contribution we present a concrete process of usage model definition which increases the collaboration of all project stakeholders thus minimizing the mentioned risk and increasing the usage model completeness.

To speed up the derivation and maintainability of the usage models there is a necessity to structure the textual functional requirements specifications (FRS) in such a way that at least a semi-automated generation of the usage models from these is possible. We motivate the usage of formal notations e.g.

[1] 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)
10th Testing: Academic and Industrial Conference - Practice and Research Techniques (TAIC PART)
**978-1-4799-1885-0/15/$31.00 ©2015 IEEE**

domain specific languages such as Gherkin [21] in the area of requirements specifications and explore practical possibilities of their employment in an industrial SCRUM environment.

Finally, we present a concrete implementation of the aforementioned process selecting a suitable formal method of FRS definition, a tool chain which allows the semi-automated generation of usage models from structured textual requirements specifications along with a first case-study in the domain of PC desktop applications.

The remainder of the paper is organized as follows. Section two describes the current process of usage-model definition along with the challenges which its application posits. The section also motivates the usage of formal requirements notations with regard to the semi-automated derivation of usage models from these. Finally the new process is introduced. Section three explores various possibilities of a formal requirements definition both in academic and industrial literature. A prototypical implementation of the process presented in section two along with a brief description of a case study is also outlined in this section. Section four concludes the paper and gives an overview of possible process limitations. Section five presents the future work.

## II. Challenges and Solutions

### A. Current Process and Challenges

In an industrial setting a SCRUM team usually works on a number of user stories in parallel during a sprint. In the first phase the product owner derives user stories from the epics (Figure 1). User stories are defined in English either in the Microsoft Team Foundation Server [4] or in Microsoft Word by means of a template. In case GUI features are concerned a product owner (PO) may consult a usability expert in order to take usability requirements into consideration. PO is a domain expert responsible for the requirements specification.

As soon as the user story fulfils the “definition of ready” criteria (DoR), the SWQA starts with the derivation of the usage model. In Phase 2 first a paper draft of the model is created by the SWQA. This first paper draft allows the quality engineer to better understand what parts of the user story are to be modelled and on which abstraction level. In Phase 3 a software quality engineer creates the complete usage model by means of a modelling tool. He or she pays a particular attention to the semantic correctness of the usage model, this might be in the case of e.g. UML state charts [5]: meaningful guards, triggers as well as actions. As soon as the usage model has been created, a peer review by the product owner and developers takes place during which such irregularities as unreachable states or semantic errors are discovered and discussed. Also the completeness of the model is being checked. Finally the model improvements are made so that the result of this iteration is a semantically correct model which represents a valid input for a test case generator. Between the Phase 3 and Phase 4 there can be a number of iterations.

During the derivation of models from the textual requirements by the software quality engineer (Phase 2 and 3 in Figure 1), certain system aspects might not be covered or omitted. The reason why this might happen is either a misinterpretation of the requirements or a possible lack of the domain knowledge by SWQA. The information gets lost in translation. In the review phase, the danger that PO and SWQA focus too much on the present model and leave out important software product aspects to be tested is high.

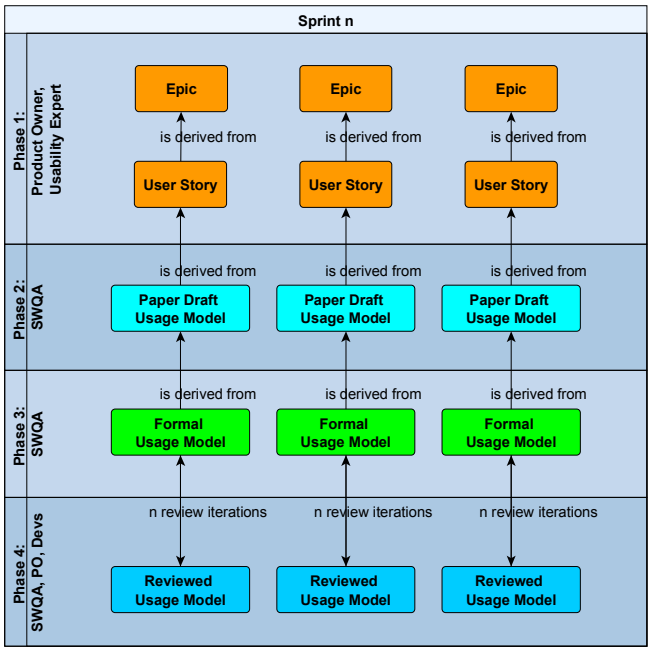


Fig. 1. Current process of usage-model definition

In order to avoid the knowledge loss which might occur during the model derivation by SWQA, the PO and other stake holders need to be able to define parts of the model on their own. The challenge that arises is how to do this since product owners are domain experts and often lack deep knowledge in modelling. "Thus they need guidance while reading model-based test specifications which are usage models. Other problem is that every usage model is written on a certain level of abstraction so that domain experts have difficulties imagining the exact functionality of a certain feature to be tested, especially if this feature has not been implemented yet and thus its functionality cannot be demonstrated directly to them" [2].

Another challenge is how to reduce the maintainability effort of usage models. It is a well-known fact that in an agile environment requirements can change often. Thus in case of a requirements change there is an effort to be made not only to adjust the textual FRS, but also a usage model needs to be updated. In Entin [1] we defined and successfully implemented a process which decreases the maintainability effort of test automation code in that the so-called test steps have been introduced. However the usage models itself still need to be changed completely manually. The question that arises is how to structure the requirements in such way that

after a possible change of these, the adjusted usage-models could be derived in at least a semi-automated way. This would not only reduce the maintainability effort but also generally speed up the entire process of automated test case generation. Clearly, in case of a requirements change, the manual effort needed to adjust a model will hardly ever be zero because even if parts of the model would be generated automatically, the SWQA still enhances the latter manually in many cases.

Both our experience and that of other industrial MBT practitioners [6] has shown that model creation is a complex task which requires not only basic knowledge of modelling methods such as e.g. state machines or UML state charts but the expertise in structuring and tailoring the model along with finding the right level of abstraction. Especially the latter point is oftentimes very problematic for the MBT beginners who think on the abstraction level of a single test case which does not represent the customer's perspective of system under test (SUT) usage in its entire complexity. Thus the question here is how to ease the software quality engineers, who are not well-versed in modelling, the application of model-based testing approaches in their projects by generating well-structured parts of the model automatically from FRS?

### *B. Motivation of Using Formal Notations*

In our everyday project life we once applied the following approach [7]. During a user story discussion at the beginning of a sprint we have asked our product owner to add to the requirements specification the so-called "happy test case" description. The "happy test case" is a textual, unstructured English language, step-by-step description of a standard feature usage enhanced by the definition of the expected results. We observed that the introduction of this approach has brought several benefits to the project stakeholders. The SWQA had a clear example, a path through the application, which could be used as it is in the usage model definition. The software quality engineer did not have to elicit this test case from the user story description in the first place risking to omit or not to cover some important requirements. The software developers had a better understanding of the feature to be implemented. Moreover they were able to take over the execution of the happy test case before handing over the implemented user story to the SWQA.

The application of the aforementioned approach brought us to the following idea: why not let product owners describe an exemplary set of happy paths by means of a textual but more formal notation? A happy path is a standard, every-day usage of a feature to be tested described in a structured, step-by-step way. From one happy path one or more happy test cases can be derived. Enhanced by a set of alternative paths through the feature under test in the same formal language, it will be possible to derive usage models from these in an automated or at least semi-automated way.

### *C. New Process*

We suggest the following process aimed to increase collaboration between the project stakeholders while defining a usage model and thus ultimately increasing the completeness and precision of the latter (Figure 2).

In the first phase, product owner eventually in collaboration with a usability expert (UE) derives a user story from an epic. The result of this step is a requirements document in plain text possibly enhanced by pictures. Based on the document the product owner defines a set of happy paths by means of a formal notation. During the definition he is able to generate the so-called partial usage model in order to visualize the requirements expressed by the mentioned set of happy paths.

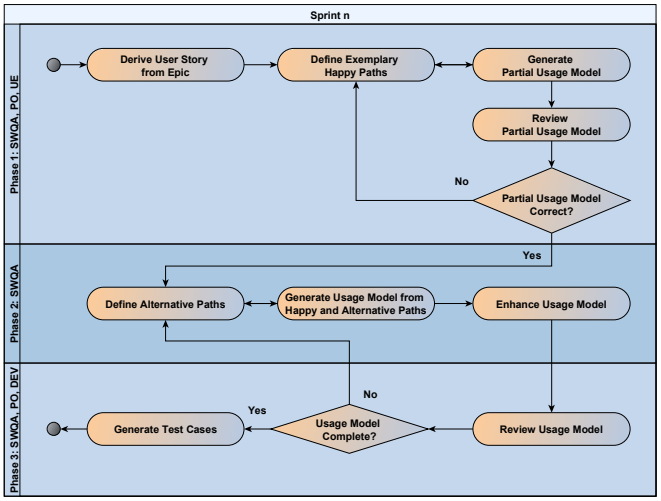


Fig. 2. New process of usage-model definition

In case he detects any missing requirements he may want to enhance the already created formal description. As soon as the product owner is ready with the definition, he or she reviews the partial usage model together with the SWQA. The software quality engineer is able to assess the semantic correctness of the model and in case the model is incomplete or incorrect, SWQA is able to adjust the latter. Obviously, a discussion of the happy paths between SWQA and PO takes place allowing to possibly discover missing requirements already at the very early stage. The result of the Phase 1 is the set of complete and semantically correct formally defined happy paths.

In Phase 2 the SWQA enhances the set of happy paths by a number of alternative paths. An alternative path is a step-by-step definition of a less frequent feature usage from the user perspective accompanied by the definition of expected results per each step. After this step the software quality engineer might want to visualize the resulting usage model, thus he or she may want to generate the latter from the current set of happy and alternative paths. This step can occur various times during the definition phase since the SWQA may discover either incorrectness of the resulting model or some missing information. In such case the software quality engineer adjusts or enhances the paths definition after which he or she visualizes the resulting usage model again. It is important to notice that experienced software test engineers might prefer to define the set of alternative paths not by means of the formal

textual notation but directly in the usage model generated automatically from the set of happy paths. As for the less-experienced SWQA, they are free to choose between using the formal textual notation or, if they feel confident, to make the enhancements directly in the generated model.

As soon as the SWQA considers the set of paths to be complete, the quality engineer might want to enhance the resulting usage model by some formal information needed to be able to automatically generate test cases from it. In case of UML state charts, for instance, he or she might want to add some guards or actions. The result of Phase 2 is a complete and semantically correct usage model. In Phase 3 the final review of the usage model by product owner, developers and quality engineers takes place. In case the other project stake holders do not have anything to add to the discussed model, SWQA generates test cases automatically from it. Otherwise SWQA has to make necessary adjustments.

## III. Process Implementation

### *A. Selection of the Formal Notation*

We have identified the following criteria to be taken into consideration while selecting a formal notation. Firstly, the notation must have a degree of formality necessary to derive usage models. Thus the selection of the suitable formalism largely depends on the formal method used to define usage models currently in operation. Since at OMICRON SWQA we work with a subset of UML state-charts [5], it must be possible to derive these from the path specifications described by means of the selected notation.

Another important aspect is the fact that the selected formal notation must be readable and writable by all project stakeholders: Since electrical engineers are most familiar with the requirements specifications defined in plain or structured English language, the notation must contain as much natural language as possible. Obviously each formal language contains a set of keywords. The number of such keywords must be kept low. Ideally four to five key words are the maximal number. The reason for this is the fact that electrical engineers have only basic knowledge in programming, so that requirements definition should not be turned into coding. Finally, the tool support for the formal requirements definition must be guaranteed. Ideally no new off-the-shelf tools are needed for the introduction of the approach into the daily project business. Also the number of proprietary tools needed for the introduction of the process has to be minimized [2]. In our context the following tools are being used for the application of MBT: Microsoft Visual Studio and Test Automation Infrastructure [1].

The research of both industrial and academic literature suggested the creation of a proprietary domain specific language or use of an existing one [8, 9]. Among the mature industrial approaches to the creation of domain-specific languages (DSL) there is XText by Eclipse foundation [9]. It allows the definition of a dedicated grammar and provides the users with corresponding parsers and integration into Eclipse IDE. An advantage of this approach is the possibility to decide upon the number and semantics of the key words. One main disadvantage is: since a new DSL would be created, there is no knowledge about it present in the community. That would mean an additional documentation overhead because otherwise product owners and other project stakeholders would have no possibility to learn the language on their own.

There is a number of academic approaches to the automated generation of usage models. One set of approaches tries to generate usage models from the logged data recorded during the real-users' interactions with the SUT [10,22]. A clear disadvantage of these is the fact that generated models represent the current system behaviour but not the desired one as is the case with manually created usage models. Another alternative that has been in use for at least ten years [21] is Behaviour Driven Development (BDD) with a dedicated domain specific language called Gherkin. “BDD is a set of software engineering practices designed to help teams build and deliver more valuable, higher quality software faster. BDD provides a common language based on simple, structured sentences expressed in English (or in the native language of the stakeholders)" [21]. The first idea to transform Gherkin into state machine notation appeared in industrial literature [12]. In the article the author gives a motivation for translating BDD notations into the FSM (final state machines) in order to answer the question: how many scenarios, or paths of scenarios, are enough to test the system under test thoroughly? Unfortunately no concrete implementation or application in an industrial environment of the approach is given.

In Scerri [13] the authors present an algorithm which transforms Gherkin-based notations to the state-machine representation used by QuickCheck tool [23]. The publication also presents a first case-study result which shows the general feasibility of transformation between this language and state machines. The application of our approach shows a concrete process of how to use it in an industrial environment during the product development starting on the requirement definition level whereas the authors in [13] start at the stage when the product is already partially tested and a number of automated test cases has been already written. From these test cases the set of paths has been derived. This approach, much like [11], might lead to the requirements loss since test cases are usually written by SWQA which might misinterpret or omit important information. Such an application also requires a lot of refactoring of already existing automation code. It is unclear what concrete advantages the usage of structured requirements specification languages such as e.g. Gherkin brings to the project stake-holders since there seem none to be involved except SWQA.

In the context of OMICRON, we have decided to employ Gherkin as the formal notation of choice for the following

reasons. This language has only four key words a user needs to learn: Given, When, Then, And. There is a vast body of knowledge in the community about this formal notation. There is a Microsoft Visual Studio plugin which allows the usage of the Gherkin language – SpecFlow [14]. The only disadvantage of using SpecFlow-based paths descriptions is the necessity to implement a dedicated tool which parses and maps the SpecFlow files to the UML state chart notation. However it turned out that the complexity and coding effort of such tool are rather low: approximately 1000 lines of code.

### *B. Usage of SpecFlow for the Path Definition*

Each scenario is defined by a key-word "Scenario". It consists of three key-words: "Given", "When", "Then". Each scenario must have an ID. ID is defined by square brackets in which the key-word ID="Some Unique ID" is placed. ID must be unique. The Given clause represents a current state of the system under test in SpecFlow. The state is a prerequisite for a certain activity performed by the user on the SUT which is described by When clause. The Then clause describes a target state of the SUT in SpecFlow after the activity stated in When clause has been performed by a user.

A path is a logical succession of scenarios. Each scenario describes an interaction of the user with the SUT on the system level. Each scenario in the path refers in its Given clause to a predecessor scenario. The Then clause of the predecessor scenario represents the current state of the SUT. The reference to a predecessor scenario in the Given clause is denoted by the predecessor Scenario ID and its full name (Figure 3). The When clause maps to the transition between the current SUT state and the target SUT state which is represented by Then clause.

```
#Begin of the Happy Path 1 (defined by PO)
Scenario: [ID=1] Customer creates and then changes an asset
    Given Customer has two clients: Client A and Client B
    When Client A creates an Asset and syncs it to the server
    | AssetKind   | AssetType   | SerialNumber |
    | Transformer | two-winding | SampleNumber |
    Then There is a changed asset on the server
    | AssetKind   | AssetType   | SerialNumber |
    | Transformer | two-winding | SampleNumber |
    And The asset is ready for the download

Scenario: [ID=2] Customer decides to change the asset nameplate
    Given [ID=1] Customer creates and then changes an asset
    When  Client B changes asset nameplate
    | Manufacturer   | Year        | SerialNumber |
    | ABB            | 1985        | 777          |
    Then  Asset nameplate on client B is changed
```

Fig. 3. An exemplary path definition by Gherkin

A concrete tool chain which implements the process is described in the following. The definition of the paths is performed by a dedicated SpecFlow plugin in Microsoft Visual Studio. The manually generated feature file is imported into the Text2Mod tool which has been implemented as a plugin of TAI framework [1]. The output of the transformation step performed by Text2Mod is a usage model in graphml file format [15]. We have chosen this format due to its lesser complexity compared to XMI [16]. The visualisation of the transformation result is possible by yED editor [17]. Finally, the usage model in graphml format is imported into TAI framework which offers a number of test generation algorithms [18]. TAI framework is able to generate test cases in various languages: C++, C# and plain Excel style sheet which allows for an easy integration with most test case management systems such as e.g. Silk [19] test case management system. The test case execution is performed by MSTest [4] in case of C# test cases and by Google Test Framework [20] for C++ test cases.

### *C. Case Study*

The Primary Test Manager is an application developed by OMICRON which supports (together with OMICRON devices) the workflow for analysing the condition of transformers and other assets which are organized by assigning them to different locations. The system size is approximately 400.000 lines of code. During the analysis a so-called job is created. A job consists of a number of tests each of which contains a number measurements to be executed on a certain asset. The asset is situated at the specific location. The system under test is a client server application which allows to synchronise all locally created jobs to a server. The feature to be tested is the synchronisation of a single job to the server. In Figure 4 an exemplary happy path is defined. In total there were 5 paths defined for the feature.

```
#Begin of the Happy Path 2 (defined by PO)
Scenario: [ID=5] Customer changes an existing job asset and job location
Given [ID=1] Customer starts PTM
When Customer changes an existing job asset and job location
| JobName | LocationName |
| Trafo3W | New York     |
Then job asset and job location are changed
| JobName | LocationName |
| Trafo3W | New York     |

Scenario: [ID=6] Customer creates and syncs a Job based on existing Location/Asset
Given [ID=5] Customer changes an existing asset and location
When Customer creates  new job and then he clicks Synchronize in the job context menu
| JobName   |
| NewCTtest |
Then The new job is uploaded
| JobName   |
| NewCTtest |
#End of the Happy Path 2 (defined by PO)
```

Fig. 4. A case-study happy path

Our product owner has been presented the approach during a user story discussion meeting. Firstly, two use cases from the user story have been written in plain English text. In the next step a software quality engineer has explained the SpecFlow scenario structure and the path definition to the product owner. It took about 7 minutes, discussion included. Finally the product owner has created a path definition for the second use case on his own by means of a text editor. This step took about 5 minutes, review of the partial model included (Figure 2, Phase 1). The SWQA needed approximately 90 minutes to define a set of alternative paths (Figure 2, Phase 2). The final usage model review (Figure 2, Phase 3) took approximately 30 minutes. The final model itself consisted of 11 states and 14 transitions.

Generally the PO could imagine using the notation. Mainly the feedback given concerned the usability improvements of the approach. For example, there are path steps which do not have

an immediate customer value such as Scenario [ID=5] in Figure 4. For that purpose the PO wished a sort of a library where predefined scenarios are stored so that he or she can refer to them while defining a path. In such way the amount of copy/paste operations would be reduced. The resulting automatically generated model is shown in Figure 5.

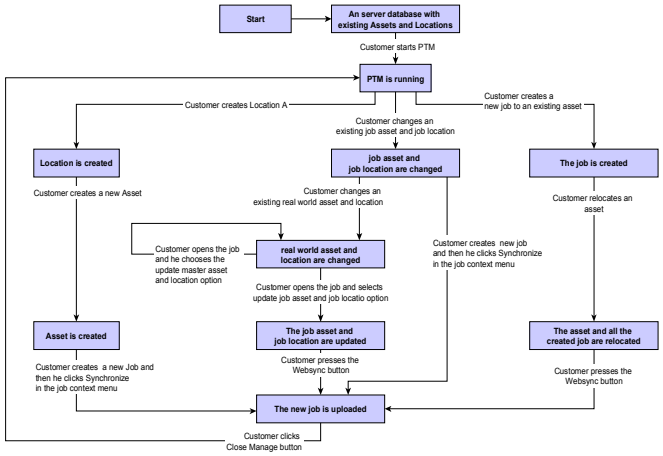


Fig. 5. The automatically generated usage model

## IV. Conclusion and Discussion

In this contribution we presented a process which aims at increasing the usage model precision and completeness in the area of model-based testing. The main advantage of the process is that it allows the involvement of the entire project team into the model definition from the very early stages of the software product development. Another advantage is that by using a very simple formal notation for the requirements definition also less experienced software test engineers are able to generate well-structured parts of the usage model and enhance them subsequently. Lastly, the maintenance effort of the usage models should be reduced by the application of the proposed process.

In the following we would like to discuss possible limitations of the presented process. In order to define models which fully comply with such standards as e.g. UML, the automatically generated model has to be enhanced by the SWQA which implies a certain manual effort. Another possibility would be to consider these enhancements already in the formal textual requirements definition. However that might decrease the readability of these and also require some modelling method knowledge by the PO which might make the application of the presented process difficult. Our experiences with the application of the UML state charts have shown that e.g. such elements as transition actions are not always needed so that the only modelling element the SWQA had to enhance the created usage models by were the guards. Another aspect to consider is that each project stake holder has a different perspective on the requirements. Thus a PO, while defining a set of happy paths, might theoretically omit certain aspects crucial to the SWQA. In this case a review of the happy paths (Figure 2, Phase 1) by the software quality engineer is essential to the successful application of the process presented in this contribution.

## V. Future Work

The PTM is being developed following the Scaled Agile Framework [24] in which approximately thirty project participants divided in four SCRUM teams develop various product features. Until now the presented approach has been evaluated only within one team with the special focus on building a client-server application for the management of measurements. In the future we plan to evaluate the process in the SCRUM teams which focus on the implementation of such product features as measurement configuration for various assets and their execution.

Further extension of the approach within the PTM project would require the possibility to reuse certain scenarios across different teams. One of the prerequisites of increasing the reusability is the possibility to parameterize scenarios. Gherkin language allows the definition of parameters within the scenario clauses. The effort has to be put into the enhancement of the transformation tool which must be able to recognize such parameters. Arguably one of the future challenges will be how to increase the usability of the textual scenario definition. For that purposes we plan to implement a Visual Studio plugin which would suggest a number of possible pre-defined scenarios to the user thus speeding up the textual path definition.

## Acknowledgment

Vladimir Entin thanks Izrail and Liubov Entin for moral support. The authors thank Bernd Paulmichl, Alexander Aberer and Rainer Kanzi for fruitful discussions and inspiration.

## References

[1] V. Entin, M. Winder, and B. Zhang, “Combining Model-Based and Capture-Replay Testing Techniques of Graphical User Interfaces” Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on.

[2] V. Entin, M. Winder, B. Zhang and S. Christmann „Introducing Model-Based Testing in an Industrial Scrum Project” Automation of Software Test (AST), 2012 7th International Workshop on

[3] R. Pichler, “Scrum”, book, dpunkt.verlag, Heidelberg, Germany, 2008.

[4] Microsoft Team Foundation Server. http://msdn.microsoft.com/en-us/vstudio/ff637362.aspx Accessed January 2015.

[5] M. Hitz et al., “UML@Work”, book, dpunkt.verlag, Heidelberg, Germany, 2005.

[6] H. Fredriksson “Using Model Based Testing for Robustness Tests” User Conference on Advanced Automated Testing, 2014 2nd International Conference

[7] J. Koch, S. Middeke, “Testfallbasierte Anforderungsdokumentation” ObjektSpektrum, February 2014.

[8] Arne-Michael Törsel, "Automated Test Case Generation for Web Applications from a Domain Specific Model", *COMPSACW*, 2011, 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops, 2012 IEEE 36th Annual Computer Software and Applications Conference Workshops 2011

[9] Eclipse XText. https://eclipse.org/Xtext/ Accessed January 2015

[10] P. Brooks, and A. Memon, “Automated GUI Testing Guided By Usage Profiles,” ACM ASE'07 pp. 333–342, November 5-9, 2007, Atlanta Georgia, USA.

[11] Antti Jaaskelainen, Antti Kervinen, Mika Katara, Antti Valmari & Heikki Virtanen (2009): Synthesizing Test Models from Test Cases. In: Hardware and Software: Verification and Testing, LNCS 5394, Springer

[12] The Truth About BDD, Clean Coder. https://sites.google.com/site/unclebobconsultingllc/the-truth-about-bdd Accessed January 2015

[13] C. Colombo, M. Micallef, M. Scerri, "Verifying Web Applications: From Business Level Specifications to Automated Model-Based Testing" EPTCS 141 9th Workshop on Model-Based Testing, 2014, Grenoble, France.

[14] SpecFlow. http://www.specflow.org/ Accessed January 2015

[15] The GraphML File Format. http://graphml.graphdrawing.org/ Accessed January 2015.

[16] XML Metadata Interchange. http://www.omg.org/spec/XMI/ Accessed January 2015

[17] yED graph Editor. http://www.yworks.com/en/products/yfiles/yed/ Accessed January 2015

[18] Winder, M. Automation of regression testing based on usage models. Master's thesis, Technical University of Graz, 2012

[19] Silk Central http://www.borland.com/Products/Software-Testing/Test-Management/Silk-Central Accessed January 2015

[20] Google Test Framework. https://code.google.com/p/googletest/

[21] J.F. Smart. “BDD in Action: Behavior-Driven Development for the Whole Software Lifecycle”, book, Hanning, Shelter Island, USA, 2015

[22] J. Sant, A. Souter and L. Greenwald “An Exploration of Statistical Models for Automated Test Case Generation,” Workshop on Dynamic Analysis (WODA 2005) 17 May 2005, St. Luis, MO, USA.

[23] QuickCheck. http://quviq.com/ Accessed January 2015

[24] Scaled Agile Framework: http://www.scaledagileframework.com/ Accessed March 2015