Lessons Learned from Making the Transition to Model-Based GUI Testing

Rudolf Ramler
Software Competence Center
Hagenberg, Austria
rudolf.ramler@scch.at

Claus Klammer
Software Competence Center
Hagenberg, Austria
claus.klammer@scch.at

Thomas Wetzlmaier
Software Competence Center
Hagenberg, Austria
thomas.wetzlmaier@scch.at

ABSTRACT

Model-based testing (MBT) has been proposed as an effective and versatile approach for testing graphical user interfaces (GUIs) by automatically generating executable test cases from a model of the GUI. Model-based GUI testing has received increasing attention in research, but it is still rarely applied in practice. In this paper, we present our experiences and share the lessons we learned from successfully introducing MBT for GUI testing in three industry projects. We describe the underlying modeling approach, the development of tests models in joint workshops, the implementation of the test model in form of model programs, and the integration of MBT in the test automation architecture. The findings distilled from the three cases are summarized as lessons learned to support the adoption of a model-based approach for GUI testing in practice.

CCS CONCEPTS

• Software and its engineering~Software testing and debugging

KEYWORDS

GUI testing, model-based testing, test model development, test automation architecture

ACM Reference format:

Rudolf Ramler, Claus Klammer, and Thomas Wetzlmaier. 2019. Lessons Learned from Making the Transition to Model-Based GUI Testing. In *Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation (A-TEST '19), August 26–27, 2019, Tallinn, Estonia. ACM, New York, NY, USA,, 6 pages.* https://doi.org/10.1145/3340433.3342823

1 INTRODUCTION

Graphical user interfaces (GUIs) are a common way for users to interact with software systems. The GUI's quality is widely observable and has a direct impact on how the quality of the entire system is perceived. GUI testing is therefore considered highly important in practice, but it is a costly and challenging activity. Model-based testing (MBT) has been proposed as an effective and versatile approach for improving the situation in GUI testing (cf. [1], [2], [3]). Despite the increasing attention of model-based GUI testing in research, it is still rarely applied in practice.

In this paper, we describe the introduction of MBT for automated GUI testing in three industry projects from different companies. Each of the projects already had automated tests for the GUI, but they were considered insufficient to cover the huge number of possible scenarios in which a user can interact with the system under test (SUT). MBT was introduced to complement the existing tests and to increase the coverage with end-to-end testing via the GUI. The results achieved with MBT were positively received by the practitioners. In all three cases, the application of MBT helped to reveal new defects that the existing tests were not able to find. Furthermore, in two companies the involved practitioners also applied MBT to their other projects.

MBT is still far from being a mainstream testing technique for GUIs. There are only a few reports on applications in real-world projects (e.g., [4], [5], [6]). The goal of this paper is to summarize our insights and lessons learned from successfully introducing MBT for GUI testing. We believe that these lessons can be helpful for planning the transition to MBT and, furthermore, also for introducing other approaches for automated test case generation in GUI testing [7] beyond MBT.

2 INDUSTRY CONTEXT

The lessons we present in this paper have been derived from the introduction and application of model-based testing for three software systems from industry. The introduction of MBT has been initiated as part of the knowledge and technology transfer conducted in an industry-academia collaboration project.

The common ground for transitioning to MBT in the three industry projects was that they all had existing automated GUI tests, although using different automation approaches. For each project, we created a roadmap with the current automation approach as starting point and MBT as final goal. The suggested paths were based on our experience with introducing test case generation for GUI testing in industry [8]. Fig. 1 illustrates the (simplified) paths to MBT taken in each project.

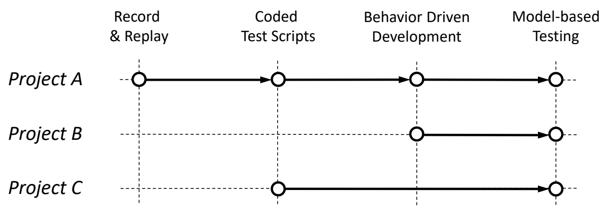

Figure 1: Transition to MBT for GUI testing.

Project A: The involved company develops solutions for large-scale building automation. Recently they developed a new Web client for their central management server. Besides manual testing, automated test scripts were recorded for the Web client using a capture and replay approach. MBT was introduced with the goal to move from static test scripts to a more versatile automation approach that supports end-to-end testing with different user roles. MBT helped to find new defects in the GUI representation and in input data constraints.

Project B: In this case, the SUT was a human machine interface for industrial machinery in a safety-critical environment, requiring high quality standards. Existing automated GUI tests had been implemented with the testing framework SpecFlow for Behavior Driven Development (BDD), page objects as abstraction layer, and the tool Ranorex. However, the automated test cases covered only a small fraction of the huge number of scenarios in which a user could potentially interact with the system. We applied model-based testing to generate tests for a wide variety of interaction scenarios. With MBT we were able to reveal new defects for an already tested version of the SUT.

Project C: MBT was applied for testing a Web application for business analytics providing dashboards and reports for process data. Automated tests had been implemented with Selenium mainly for individual dashboards. We introduced MBT for generating end-to-end test scenarios spanning over several dashboards representing paths a user would take when drilling down and analyzing the data from different perspectives. The MBT approach revealed timing issues when navigating from one dashboard to another.

Lesson #1: *Complement existing tests with model-based testing.*

In all projects, we found that there are already exiting test scripts covering the main usage scenarios. These tests are not only a valuable source of information for the design of the test model, but they also assure that at least the most essential parts of the SUT are working. Instead of replacing the existing tests, the tests generated from models should be used to increase the level of detail in which the functionality is tested and to extend the test scope to other functional areas. Thereby, reusing an already established test infrastructure can give the introduction of model-based testing a head start. Moreover, sharing common layers of the test automation architecture can help to reduce the overall test implementation and maintenance effort.

3 MODELING APPROACH FOR GUI TESTING

Different techniques for modeling GUI-based systems have been proposed in the literature. Banerjee [10] provides a comprehensive overview of common techniques used for GUI testing, including modeling techniques based on state machines (e.g., finite state machines, variable finite state machines, off-nominal finite state machines) and event-based techniques (e.g., event-flow graphs or event-interaction graphs).

We decided to use *extended finite state machines (EFSMs)* for modeling, because this technique has already been successfully applied in software testing practice [11] and it is supported by available tools for mode-based testing [12]. Like conventional finite state machines, EFSM models consist of a set of states in which the system can be and transitions between these states, triggered by events. The extension provided by EFSMs over finite state machines are state variables. These variables are used to describe a set of states, which allows a compact representation of state machines with large numbers of different states. Furthermore, transitions are guarded by trigger conditions that are also controlled by state variables.

Lesson #2: *The modeling concepts provided by EFSMs can be used to distinguish between externally visible states of the GUI and internal states of the SUT reflecting its behavior.*

A state in an EFSM can be described either by depicting it graphically as box or in form of state variables. We used the graphical states (boxes) to represent the screens of the SUT (i.e., dialog windows or Web pages). The transitions between the states represent the navigation paths from one screen to another, triggered by the corresponding user actions. Hence, the graphical representation of the state model shows the GUI in the same way as it is perceived by the user. Screens and states match quite well, as states describe the situation in which the system is waiting for a user to perform an action, which then results in a transition to another state. Fig. 2 shows an example for such a state model for a web-based application.

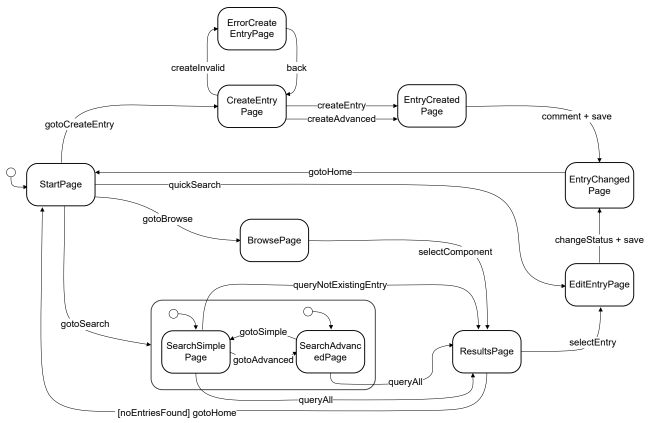

Figure 2: Example GUI model using reduced EFSM.

The internal state of the SUT and its behavior is less visible and, therefore, usually also less obvious for the user. We used state variables to represent the internal states of the SUT. A state variable can represent, for example, the *total number of customer entries*, which have been created. Then, state variables can be used to define guard conditions (e.g., *numberOfEntries > 0* for the action *removeEntry*), and they can also be used to check the number of entries returned by a search. In the graphical model, state variables and guard conditions are shown as annotations to transition labels.

Lesson #3: *Abstraction helps to reduce the complexity of models representing the GUI.*

Abstraction is key concept in modeling. It reduces the level of detail and therefore also the complexity of the model. We used abstractions in two ways. First, screens were represented by states in the graphical model. It is not always obvious what should be considered as an individual screen in the GUI, since the same screen may appear several times in the SUT but with a different layout and with different data. Furthermore, the same screens can provide dynamic GUI elements that also change the visual appearance. The complexity of the graphical model can be reduced by aggregating related screens to one abstract screen and using state variables to describe the individual behavior.

The complexity of state models can be reduced even further by combining sequences of events into abstract, high-level user actions. Instead of including each individual interaction performed by a user on the GUI such as *enter first name*, *enter last name*, *click the ok button*, etc., we define high-level user actions such as *create new customer* that combine all related user interactions. The model becomes more concise and easier to understand, especially when descriptive names are used for the actions. The resulting model is also called *reduced (E)FSM* [10].

4 TEST MODEL DEVELOPMENT

Where do (good) test models come from? In theory, the basis can be existing models from (formal) requirements specifications or system design. In practice, however, formally specified requirements are rare [13]. Usually, projects have no or only few specifications other than natural language documents.

The projects of our industry partners applied agile practices and requirements were mainly specified as user stories, describing the functionality in an informal way from the perspective of the user. No graphical notations or models were available. The user stories were adequate for managing development activities, but the textual descriptions were inherently ambiguous and lacked details relevant for testing. Furthermore, important knowledge about the requirements and related design decisions was spread among various stakeholders. In order to develop test models, the necessary information had to be collected from different sources including the actual implementation.

Lesson #4: *Domain knowledge is required for developing appropriate test models. It has to be collected from different sources and by involving different stakeholders.*

In the industry projects, the main source of information used to design test models was the latest version of the SUT accompanied by existing tests, documentation, etc. However, these sources were not sufficient to develop a complete test model. We analyzed the SUT as a starting point and then conducted a series of workshops with stakeholders to elicit all relevant information from which we then built the test models.

In case of *Project A*, the test manager organized a workshop including the product owner, developers, and testers. Together they created a first version of the test model, which was later refined by the testers when exploring the SUT. In *Project B* and *Project C*, we worked together with the senior testers and conducted two half-day workshop meetings to discuss and continually refine the initial test model that had been based on a released version of the SUT and the accompanying test cases.

The goal of the modeling workshop was to consolidate the knowledge of the participants into a common test model. It turned out that the decision to map states to screens was a huge benefit for modeling. It made the identification and description of states, transitions, events, actions, guard conditions and state variables simple and easy to understand, even for participants not familiar with the notation of state diagrams.

Lesson #5: *The state model is easy to create and understand if it closely matches the system's GUI and navigation paths.*

When analyzing the SUT, we prepared printouts of screenshots taken from every page we visited during manual exploration. In the workshop, we pinned the screenshots on a whiteboard and connected them by drawing lines to visualize the navigations paths users can take (see Fig. 3). We labeled the lines connecting the screens with descriptive names for the events caused by user actions, e.g., creating a new customer entry. They correspond to high-level actions performed by the user on a page and they trigger a transition to another page when completed. Usually, the actions consist of a defined sequence of individual interactions, e.g., entering values in fields and clicking the submit button. Grouping these individual interactions into high-level actions reduced the model complexity and raised the awareness for usage scenarios spanning over several pages. In addition to the labels, we annotated the connecting lines in the model with guard conditions, which refer to variables that also reflect the state of the SUT. In contrast to screens, which we explicitly modeled as states, we used state variables to describe internal properties of the SUT. These properties are usually not directly visible and accessible via the GUI, e.g., the number of entries in the database. However, they are linked with actions performed by the user (e.g., creating a customer entry) and, thus, state variables can be used to keep track of internal states.

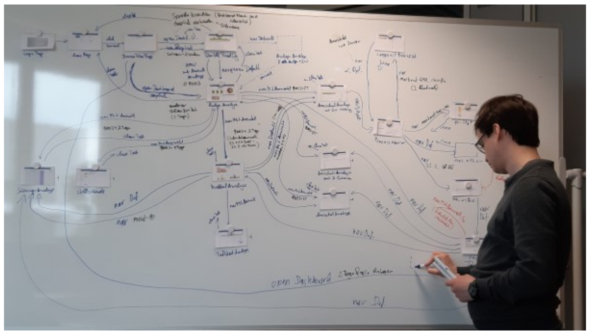
Figure 3: Model development on the whiteboard.

All tested industry systems were large and complex. In order to keep the model concise, we included only those aspects of the system (states, user actions, variables, etc.) that were relevant for the particular testing objective. Unrelated details were deliberately omitted. Nevertheless, a model must contain enough details to eventually derive tests that can be executed automatically and that are able to reveal defects. The largest model was produced in Project C. It had 22 states representing different pages of the SUT. The model was created with the goal of end-to-end testing the entire system in order to detect unwanted interactions between individual use cases, which were otherwise only tested in isolation. In the other two projects, smaller partial models were created. For example, in Project A, separate models were created for different user roles, which were used to generate tests of each specific role first. Later, the partial models were combined to a large model including logon and logoff for switching between roles. The different smaller models focused on input validation and creating complex data structures via the GUI.

Lesson #6: *Focus on the aspects of the SUT that are relevant for the testing objective. Create separate models for different objectives.*

Aligning the model with testing objectives usually led to the question "What parts/aspects of the system should be in the focus of testing?". In the studied projects, initially, the testing objectives were not clearly defined. The ongoing discussion initiated as part of the modeling step helped to elicit and improve testing objectives.

5 TEST MODEL IMPLEMENTATION

Different ways exist to transform a test model into a machine executable or interpretable form, so it can be used to automatically generate test cases or test scripts. Examples are UML models, domain-specific languages, or common programming languages. The choice depends on the used model-based tool or framework and – critical for the acceptance in practice – on the implementation skills and experiences of the testers, who have to work with the models in their daily job.

Lesson #7: *Implementing test models as code using a common high-level programming language increases user acceptance.*

Automated testing usually requires some programming skills and experience. Hence, in the studied projects, the testers were already familiar with writing code, e.g., for developing GUI test scripts. For MBT, we transformed the test model into an executable model program. Expressing the model in form of source code, preferably in the same programming language as the SUT, has several benefits. No new modeling languages or tools have to be learned and testers can start right away. Development expertise and tools are usually available in-house. Best practices and processes from development can be applied. Extra costs due to license fees or support for modeling tools can be avoided. Furthermore, technology boundaries between the test implementation and the implementation of the SUT can be reduced.

We implemented the test models as executable model programs [12]. Thereby, the state machine of the test model is implemented in a common programming language using a model-based testing framework. We used the framework OSMO Tester [14]. It allows writing the test model in Java (Fig. 4) and generating test sequences by running the resulting program.

```java
public class TestModel {

  private Page curPage;
  private int nrOfEntries;

  @BeforeTest public void init() {
    curPage = Page.STARTPAGE;
    nrOfEntries = 0;
  }

  @Guard("gotoCreateEntry")
  public boolean gotoCreateEntryGuard() {
    return curPage.equals(Page.STARTPAGE);
  }

  @TestStep("gotoCreateEntry")
  public void gotoCreateEntry() {
    Adapter.gotoCreateEntry();
    curPage = Page.CREATEENTRYPAGE;
  }

  @Guard("createEntry")
  public boolean createEntryGuard() {
    return curPage.equals(Page.CREATEENTRYPAGE);
  }

  @TestStep("createEntry")
  public void createEntry() {
    Adapter.createEntry(TestDataFactory.getEntryData());
    nrOfEntries++;
    curPage = Page.ENTRYCREATEDPAGE;
    assertEquals("Entry successfully created",
      Adapter.getMessage());
  }

  ...

  @Guard("selectEntry")
  public boolean selectEntryGuard() {
    return curPage.equals(Page.RESULTSPAGE)
      && (nrOfEntries > 0);
  }

  @TestStep("selectEntry")
  public void selectEntry() {
    Adapter.selectFirstEntry();
    curPage = Page.EDITBUGPAGE;
  }
```

Figure 4: Test model program implemented in Java.

6 TEST AUTOMATION ARCHITECTURE

The test model captures all information that is required to simulate a user interacting with the SUT when it is executed. However, MBT itself provides no means to actually perform interactions such as keyboard inputs or mouse clicks on GUI elements. To close this gap, an adapter is used that is based on a robot framework for GUI automation (e.g., Ranorex or Selenium). At this point, thus, MBT has to cope with the same technical challenges as any other automation approach for GUI testing.

Lesson #8: *Do not reinvent the wheel. Integrate MBT in the existing test automation architecture.*

Fig. 5 shows the different layers of the test automation architecture used in Project B [15]. The test architecture provides the environment and the infrastructure necessary for automated execution of test cases on the SUT via the GUI. The left part of the figure shows the automation stack for automating high-level system tests using the BDD framework *SpecFlow* (blue). The right part shows the integration of *MBT* (orange) based on the same stack of test automation frameworks and tools.

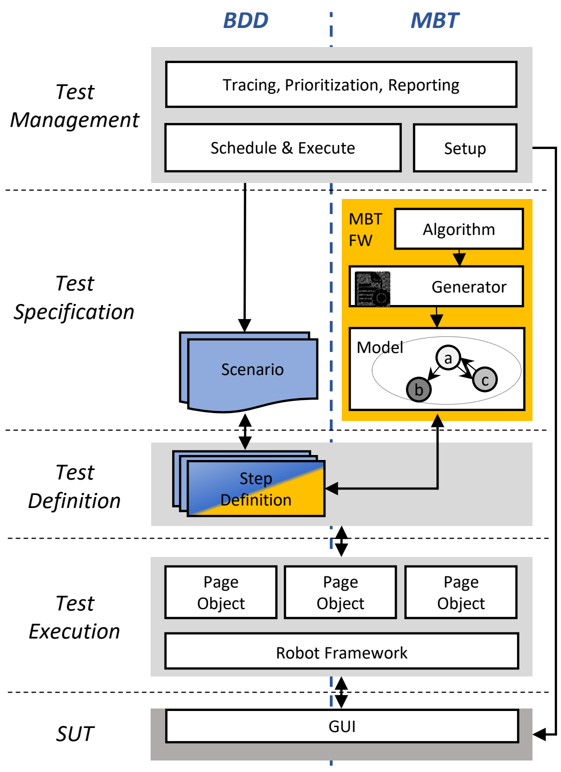

Figure 5: Test automation architecture for BDD and MBT.

Going from top to bottom, the overarching *test management layer* is concerned with prioritization and selection of tests for executions, tracing tests to requirements, and reporting results. It contains the scheduler and runner that triggers test executions. Calls to the different frameworks used for BDD and MBT are wrapped by unit tests to allow executing these tests side-by-side on the same infrastructure.

The *test specification layer* contains on the left hand side the logical descriptions of BDD tests in form of constrained natural language scenario specifications. We locate the test model on the right hand side at the same layer. Yet the individual test scenarios are dynamically generated from the model. Thus, the layer also includes components required for MBT, i.e., the generator engine that runs an algorithm to traverse the test model for deriving test cases.

Test definition layer: Automation of BDD tests is achieved by binding the specified scenarios to executable test step definitions implemented in C# or Java. They hide the technical details of accessing the SUT. We reused the test step definitions for MBT by binding the model actions to the existing step definitions. Additional step definitions were added when necessary.

Hence, the implementation of the *test execution layer* consisting of Page Objects and the use of a robot framework to access the GUI elements is shared between BDD tests and MBT tests, reducing the maintenance effort required to keep up with the evolution of the SUT.

Lesson #9: *Use a test automation architecture to manage the flexible integration of different specialized test tools and techniques.*

We organized the test automation architecture in layers according to the main tasks and activities of test automation. It helped us to manage the different technologies involved in test execution and it fostered reuse of key concepts for testing different systems. The layered architecture is also a cornerstone for an automated approach using MBT, as it allows the seamless integration of MBT into an existing test automation strategy. In Fig. 5, the architecture elements required by MBT are highlighted in orange color.

Lesson #10: *Automated model-based testing requires additional investments in reliability and testability.*

MBT takes automation to the next level. With MBT, it is possible to mass-generate a huge number of test cases that can be automatically executed to test drive the SUT. This highly automated approach implies high standards in terms of stability, robustness, error recovery, and performance of the entire automation infrastructure. The same applies to the SUT. Thus, the stability and testability of the SUT play an important role for the successful introduction of MBT. Flaky tests resulting in false positives and low performance of test execution are typical issues that need to be addressed for a high-volume test automation approach. We frequently experienced that testers insert delays in automated tests as response to timing issues in test execution. However, the root cause of these timing issues is usually a lack of robust synchronization mechanisms in the concurrent execution of the automated tests and the SUT. If a tester adds timeouts of 10 seconds as a workaround to make test execution more robust, this amounts to more than an hour of additional execution time when generating 400 tests.

7 SUMMARY AND CONCLUSIONS

The combination of MBT and GUI testing is a natural choice for dealing with the huge input space and the high number of interaction sequences users can perform on the GUI. However, model-based approaches still struggle to become a mainstream testing technique for GUIs. In this paper, we described our experiences and lessons learned from successfully introducing model-based GUI testing in industry projects, and from developing associated test models.

Table 1 provides a summary of the lessons we described and discussed in the paper, related to the different activities involved in MBT. Most lessons concern the development of appropriate models for MBT, since we found that for many industry projects the open question is "Where do (good) test models come from?".

Table 1: Summary of Lessons Learned.

#	Lesson Learned	Activity
1	*Complement existing tests with model-based testing.*	Testing strategy
2	*The modeling concepts provided by EFSMs can be used to distinguish between externally visible states of the GUI and internal states of the SUT reflecting its behavior.*	Model development
3	*Abstraction helps to reduce the complexity of models representing the GUI.*	Model development
4	*Domain knowledge is required for developing appropriate test models. It has to be collected from different sources and by involving different stakeholders.*	Model development
5	*The state model is easy to create and understand if it closely matches the system's GUI and navigation paths.*	Model development
6	*Focus on the aspects of the SUT that are relevant for the testing objective. Create separate models for different objectives.*	Model development
7	*Implementing test models as code using a common high-level programming language increases user acceptance.*	Model implementation
8	*Do not reinvent the wheel. Integrate MBT in the existing test automation architecture.*	Test automation
9	*Use a test automation architecture to manage the flexible integration of different specialized test tools and techniques.*	Test automation
10	*Automated model-based testing requires additional investments in reliability and testability.*	Test automation

Our goal in the described industry-academia collaborations was to foster the transition to a sustainable and thorough automated testing approach using MBT. Although we formulated our lessons mainly for practitioners, the academic community should also be able to identify open issues and to derive avenues for further research from our work. It is imperative that researchers systematically addresses the pain points faced by practitioners in order to support a widespread adoption of model-based GUI testing in practice.

ACKNOWLEDGMENTS

The research reported in this paper has been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry for Digital and Economic Affairs, and the Province of Upper Austria in the frame of the COMET center SCCH, grant no. FFG-865891.

REFERENCES

[1] M. Vieira, J. Leduc, B. Hasling, R. Subramanyan, and J. Kazmeier. 2006. Automation of GUI Testing Using a Model-driven Approach, In *Proceedings of the 2006 International Workshop on Automation of Software Test*, 2006, pp. 9–14.

[2] A. M. Memon. 2007. An Event-flow Model of GUI-based Applications for Testing: Research Articles, *Softw. Test. Verif. Reliab.,* vol. 17, no. 3, pp. 137–157, Sept. 2007.

[3] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. 2006. Model-Based Testing Through a GUI. In *Proceedings of the 5th International Conference on Formal Approaches to Software Testing*, 2006, pp. 16–31.

[4] V. Entin, M. Winder, B. Zhang, and S. Christmann. 2012. Introducing Model-based Testing in an Industrial Scrum Project. In *Proceedings of the 7th International Workshop on Automation of Software Test*, 2012, pp. 43–49.

[5] V. Chinnapongse, I. Lee, O. Sokolsky, S. Wang, and P.L. Jones. 2009. Model-based testing of GUI-driven applications. In *IFIP International Workshop on Software Technolgies for Embedded and Ubiquitous Systems*, 2009, pp. 203–214.

[6] A. Kervinen, M. Maunumaa, T. Pääkkönen, and M. Katara. 2005. Model-based testing through a GUI. In *International Workshop on Formal Approaches to Software Testing*, Springer, Berlin, Heidelberg, 2005, pp. 16–31.

[7] R. Ramler, G. Buchgeher, and C. Klammer. 2018. Adapting automated test generation to GUI testing of industry applications. *Information and Software Technology*, vo. 93, Jan. 2018, pp.248-263. https://doi.org/10.1016/j.infsof.2017.07.005

[8] C. Klammer and R. Ramler. 2017. A Journey from Manual Testing to Automated Test Generation in an Industry Project. In *IEEE International Conf. on Software Quality, Reliability and Security Companion (QRS-C)*, 2017, pp. 591–592.

[9] R. Ramler, C. Klammer, and G. Buchgeher. 2018. Applying Automated Test Case Generation in Industry: A Retrospective. In *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2018, pp. 364–369.

[10] I Banerjee. 2017. Advances in Model-Based Testing of GUI-Based Software. In *Advances in Computers*, vol. 105, Elsevier, pp. 45–78.

[11] R. Yang, Z. Chen, Z. Zhang, and B. Xu. 2015. EFSM-Based Test Case Generation: Sequence, Data, and Oracle. *International Journal of Software Engineering and Knowledge Engineering*, vol. 25, no. 04, May 2015, pp. 633–667.

[12] M. Utting and B. Legeard. 2007. *Practical model-based testing: a tools approach.* Elsevier/Morgan Kaufmann, 2007.

[13] H. Robinson. 2003. Obstacles and opportunities for model-based testing in an industrial software environment. In *Proceedings of the 1st European Conference on Model-Driven Software Engineering*, pp. 118–127.

[14] T. Kanstrén and O.-P. Puolitaival. 2012. Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation. In *Electronic Proceedings in Theoretical Computer Science*, vol. 80, Feb. 2012, pp. 58–72.

[15] R. Ramler and C. Klammer. 2019. Enhancing Acceptance Test-Driven Development with Model-based Test Generation. In *IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, 2019, pp.503-504.