A Review of Domain-Specific Modelling and Software Testing

Teemu Kanstrén
Security and Testing Technologies
VTT, Oulu, Finland
teemu.kanstren@vtt.fi

***Abstract*—Domain-specific modeling is an approach of using customized, domain-specific languages tailored for the domain as a basis for modeling the target system. The intent is to provide a means for domain experts to work with tools and a language closer to their domain knowledge, while abstracting away excess detail. This should provide more effective communication and ease the work done by providing a higher abstraction level. In the test automation domain, this means providing the domain experts with means to effectively create test cases based on their domain knowledge, and to communicate with the test automation experts. Despite the potential benefits and its applications, this viewpoint domain-specific modeling has received little consideration so far in test automation research. This paper reviews different approaches to applying concepts from domain specific modeling to test automation to provide a basis for further work in the area.**

Keywords-domain-specific modelling; software testing; test automation

I. INTRODUCTION

Testing is generally considered to be one of the biggest cost factors in software development. The testing process requires collaboration between several stakeholders, large investments in test infrastructure and continuous efforts in maintenance and evolution. The test infrastructure needs to be built to be able to address verifying both low-level details and high-level requirements. Domain experts need to be able to effectively communicate with the testers to ensure what needs to be implemented is implemented and is implemented correctly. Optimally, this means test automation needs to be built in layers to enable test engineers and software developers to verify the low level details, while providing domain experts the means to work and understand what is implemented and verify it, while working together to improve the resulting product.

When discussing concepts at a local level, where only a single team at a single organization is involved, having a common understanding becomes quite naturally. The people can sit down at common face-to-face meetings and quickly reach a common understanding. When teams become geographically and organizationally distributed, more difficulties arise. Different backgrounds and limited communications contribute to long delays in reaching a common basis for discussion between the different parties.

This applies to all works, not just software testing. Reaching a common understanding and maintaining that understanding requires that people can communicate using a shared terminology. Agreeing such a domain terminology is an obvious requirement for applying domain specific modeling and creating domain specific languages. A less obvious requirement is the need to first agree on what the different parties mean when they talk about domain specific languages and modeling in general.

A domain-specific model (DSM) is expressed in terms of a domain-specific language (DSL). These languages can be hugely diverse and take completely different representations, typically with "domain specific" referring to the language being specific to a company and its application(s), each language being highly customized to a specific purpose [1]. A domain specific language works best in a domain that has a lot of variation that can be expressed by the language, leading to possibilities for cost-effective application vs. the initial language design costs. In software testing, test cases describe the behavior of a system, through a common base language while each test case can be seen as a variant expressed over that language. This makes testing a great domain for application of DSM.

Domain specific languages in the context of test automation can take different forms. In our experience, some people prefer shell scripts as their domain-specific languages. Others prefer to create their own textual scripting languages, such as keywords over test frameworks. Some prefer to create graphical modeling languages, such as those presented in [2] and [3]. This can be influenced by different factors such as the expert background, test requirements, and the target domain. Often there is no clear understanding of the involvement of a domain-specific language design process, which leads to less optimal results.

In our experience the language choice and design is heavily influenced by the people's background. Someone with a strong background in Unix scripting wants to write everything in shell scripts (e.g., termed previously as "little languages" [4]). Someone with a strong background in graphical modeling languages will only consider those when talking about domain-specific languages. While all these different factors contribute to what is a suitable solution for test modeling in different contexts, a better understanding of domain specific language concepts in the testing domain, and knowledge of different options provide a basis for making more informed decisions. This paper contributes to this basis by reviewing current work and approaches for domain-specific modeling and test automation.

The following section II presents different types of domain specific languages we have observed in our work on building and applying test automation systems. Section III

 ISBN: 978-1-61208-283-7

presents examples of these using an example of a calendar application. Finally, conclusions end the paper.

II. LANGUAGES FOR TEST AUTOMATION

Various tools exist specifically intended for designing domain-specific languages. In this paper the MetaEdit+ [5] is mainly used to illustrate the concepts but various others are also available. These tools can provide good mechanisms for defining language concepts and transforming these into different types of artefacts (e.g., test scripts). The tools intended to create these languages are in general not intended to build languages with features required to express all the low-level details of the systems in the domain-specific models. They work best when the transformations for them can be written to target higher level abstractions.

For these reasons, it is typically a useful approach to build the support for the domain-specific test languages in different layers. **Figure 1** illustrates these different layers. Test frameworks (TF) are in essence programs written using general purpose programming languages (GPPL) such as Java and Python. General purpose programming languages allow for freely expressing different computational concepts, providing good support and existing libraries for writing a test platform that allow one to express the required test concepts at different levels. The test framework takes care of connecting to the actual system under test and executing the concrete test cases.

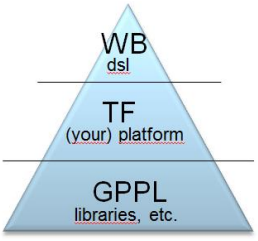

Figure 1. DSL layers.

On top of this test framework, the higher level representations and test languages can be created. In the terminology of domain-specific languages the tools used to create these languages are often called DSL workbenches (WB). Depending on the tools used and the type of test language targeted, this top layer can also be integrated with the test framework layers in a form such as a keyword driven test framework, where the keywords form the test DSL.

It is our experience that a domain specific test solution is often best build from bottom up. That is, having a good test platform available first to create and execute test cases, and using this to create the required support for testing in general. Once this support is in place it makes sense to start designing and providing the higher level DSL support on top of this existing platform. This also lends itself well to support cost-effective decision making when the extension of the support towards the top layers can be made when requirements and needs are identified.

A. Common Language Elements

While there are different approaches to building domain specific test languages, and using those to model test cases, these share a number of common language elements. The target domain needs to be expressed in terms of the domain terminology and as such a set of domain objects needs to be defined for the language. In the test automation domain, we are typically interested in expressing the various ways that the different actors involved in the system behavior can interact, and how the results of these actions should be considered (correct or failed).

Testing can be seen to represent a number of different concepts. However, in general testing is about exercising the different relevant aspects of system behavior in different ways and evaluating the results. A basic element of the test models is then the ability to explore the flow of execution in the system. In some cases, such as textual scripting languages this can take the form of implicit expression through the ordering of the script elements. In graphical notations the tools may allow one to connect the different elements as best seen fit. In test generation modeling languages this may take the form of expressing constraints over the possible combinations of the different test elements. The following subsections will discuss each of these types of test expressions.

B. Scripting Languages

A typical approach to test automation is to have a scripting language that the user can use to write regression test cases. These scripting languages can take different forms and abstraction levels.

An example of a low-level test scripting language is the TTCN-3 (testing and test control notation version 3 [6]), which is a scripting language intended for testing communication systems. While it is a low-level language requiring a lot of effort and expertise in its use to write test cases, it is something designed for testing of systems in a particular (communications) domain. For example, it has specific support for features such as ports and messages. The benefit of this type of a standardized industry domain language is also the ability to exchange information between different partners in an executable format, in a well-defined and formalized terminology.

Examples of higher level scripting languages are those defining a set of keywords for writing test cases. This is supported by keyword-driven test frameworks such as Robot Framework [7]. These keyword-based languages provide textual domain-specific testing language. Examples of their application include telecommunication systems [8] and enterprise systems [9]. These are very domain-specific and different in each case.

C. Graphical Languages

As noted, in the domain-specific modeling community, various approaches exist for creating the domain-specific languages. Besides using general tools to build textual scripting languages, there are also tools specifically intended for creating graphical modeling languages. These are typically used to build a graphical notation on top of a framework that is implemented using a general purpose programming language, or a lower level scripting language. For example, in the test automation domain, we can build a

 ISBN: 978-1-61208-283-7

graphical modeling language on top of a specific test scripting or keyword-based language.

For example, a graphical test modeling language built on top of TTCN-3 was presented in [2]. Highlighting the typical benefits to application of domain-specific modeling, significant benefits were reported in allowing a domain expert to create relevant test cases at a high level, communicate results and test intents with different stakeholders such as management, and in providing cost savings in focusing the variation modeling at a high-level on the most important aspects. At the same time, lower-level details could still be expressed in the underlying scripting language where required.

Other application examples of graphical domain-specific test languages include digital libraries and information systems [10]. Sometimes the distinction is also not so clear, for examples, with a focus on textual elements and formalisms with some graphical elements (an example of safety-critical systems in the railway domain) [11].

D. Model Based Testing

A related concept to domain-specific modeling in the software testing domain is model-based testing (MBT). In our experience, model-based testing for different people can be defined in many ways, such as using a mental model as a basis to manually write test cases, or using test stubs to model the system environment. However, a commonly used definition that we use here is from [12] as "generation of test cases with oracles from a behavioral model".

The models in model-based testing are typically different forms of state-machines, defining the potential test steps of interest and their possible combinations. The models in MBT are traditionally not considered from a viewpoint of a domain-specific language as they are hand-crafted model-programs (term used, e.g. in [13]) used to generate test cases, not to manually model test cases. However, as shown in [14], the act of modeling the potential test steps and their possible orderings also provides a basis for the definition of domain-specific test modeling language. That is, the potential test steps in the model program define the model elements (along with the state variables of the model), and the guard statements defining the possible ordering of the test steps for the generator provide a definition of the possible execution flows that can be created from these elements.

Model-based testing approaches by their nature lead to creating test models for specific domains. Examples of these application domains include smartphones user interface testing [15], automotive systems [16], and healthcare systems [17]. These approaches typically report good results when applied in a suitable context (i.e., choice of right abstraction level, addressing of high variability, and so on). However, only a few works discuss these in relation to domain-specific modeling concepts (for smartphones in [15] and generally in [14]). As widespread adoption of model-based testing has long been an elusive goal in practice, providing more synergies in this area to make it more approachable to domain experts while also making it more cost-effective to tie into other different testing techniques holds a lot of potential.

III. CALENDAR EXAMPLE

This section illustrates the principles discussed above with an example of applying them on a calendar example. This example is available online on the OSMO Tester MBT tool website [18]. The calendar is an example application where the user can create meetings and invite other people to those meetings. The user can also create tasks that are only visible to himself. Several users each have their own calendar instance.

The following subsections show an example of defining a domain-specific modeling language for this application. It starts with examples presented using a graphical notation built with the MetaEdit+ domain-specific modeling workbench [5], and proceeds to show different ways to create the underlying implementations of the test frameworks including the use of a keyword-driven framework and a model-based test tool. Possibilities for combining these different options are also discussed.

A. Terminology

Any process of applying DSM needs to begin with a definition of a common terminology. Sometimes this can be the biggest step in getting started and producing a useful and accepted solution. One might expect this to be simple for a calendar application, which is a widely used tool and concept. Yet domain-specific models are commonly defined for internal use at a company, where over time custom terms will have been adopted for effective communication between workers.

Here we use the calendar as an example for the readers of this paper, who can only be assumed to have a varying background. In a global context, many people will not be native English speakers, while the language commonly used to communicate in this context is English. Thus different mappings from the organization and personal language have an effect on how to approach building the basic blocks of a DSL. This makes it much more difficult to stay in line with the target audience and intended use of the language, as the terminology should be generally understandable and not just for the (paper) author(s).

A calendar is a very general entity and the base functionality of a calendar application as discussed in this paper is to add and remove events. As this example was originally devised it was influenced by the background of the author(s), which led to simply using the names "event" and "task" for the calendar entries involving several or just one person respectively. Yet, an event is an overloaded term in the English language, and using it in this way easily leads to confusion on what type of an event it is. Thus "event" was later renamed "meeting", while the underlying platform(s) use the terms varyingly. This simple example shows how the language design needs to consider many factors.

Besides calendar entries, we also need to fix the terminology related to the actors using the calendars. The users are people, who have certain roles in the system (organizers, participants), and perform certain actions on the different elements (create, remove, invite).

 ISBN: 978-1-61208-283-7

B. *Defining the Language Elements*

From the terminology, we can already pick a set of language elements as a starting point. We need to be able to model the properties of calendar users (people), meetings and tasks. We also need to be able to model actions of creating and removing meetings and tasks, as well as inviting people to the meetings. We start with these elements and the basic sequential test flow. **Figure 2** shows an example of a test case where a user named "bob" creates and removes a task for the date first of January, 2012.

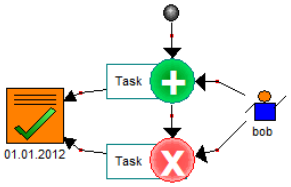

Figure 2. Two-step test flow.

Figure 3 shows the same scenario but with error notification where a second user ("john") has been added and a test case is created where he tries to delete a task created by "bob". However, as he is not in the "owner" role for the referenced task, he is not allowed to delete it. This also illustrates a design choice that needs to be made in considering a test language in general. In this case the choice has been made to allow using the language to model only the "correct" behavior of the system, and notifying when errors in the test models are observed by the modeling tool.

In other cases, it can be meaningful to allow for also the creation of test cases that exercise invalid flows of operation on the target system to test error handling behavior. This, however, requires creating different language elements as a different type of a test oracle needs to be bound to the test flow in this case (to define the correct expected response from the system). In our experience, the use of test cases and the creation of the language also serves a very useful purpose in facilitating communication between the different parties working on the system, where defining what is allowed explicitly also helps communicate the different expectations over the system behavior.

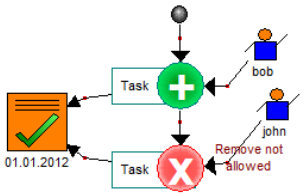

Figure 3. Embedded error checking.

Figure 4 shows examples of some of the other elements in our test language. The meetings are represented by the people shapes in three different colors (red, green, blue). Actions for working with the meetings are shown similar to how the user would interact with the task objects. The action of inviting people to meetings is not explicitly visible here, but is shown by the different types of lines used to connect the people to the meeting object(s). The organizer is connected by a solid line (the main actor on the top of the sequence flow line), whereas the participants are connected with a dashed line along the sequence path. Several concurrent flows of users working on their calendars are shown as parallel flows.

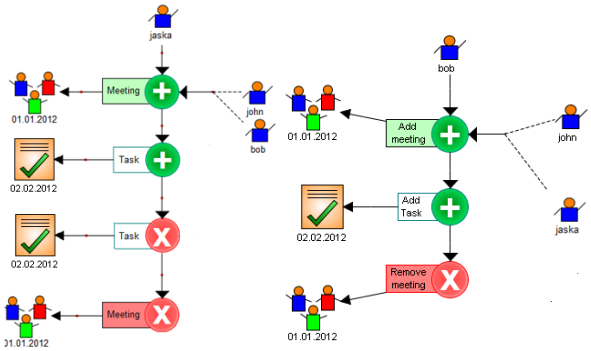

Figure 4. Parallel test flows.

Figure 5 shows an example of defining a set of model building blocks and using those as templates for building test models for the calendar application. In this snippet, we have the actions for creating and removing tasks and meetings shown. Although not shown here, similar approaches can be used to model the calendar users as well, by creating templates for the person objects and allowing copying and modifying these as required.

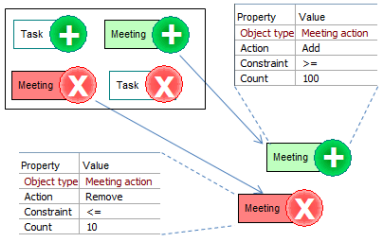

Figure 5. Model building blocks.

Finally, Figure 6 shows an example of creating a test generator configuration for a MBT tool using these same model elements. In this case, we have configured the test generator to produce test cases where four different people are involved in different roles for performing actions on creating and removing meetings.

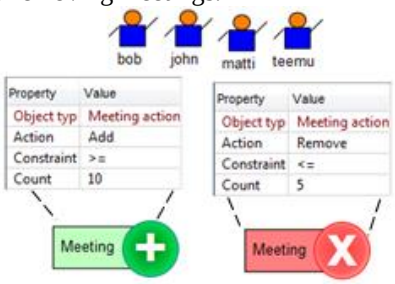

Figure 6. Generator configuration.

C. *Scoping the Language*

To scope the calendar test language, we need to consider who it is intended for and what the users (domain experts) are intended to use it for. In this case, the main purpose of the language is to allow the users (who are expected to be familiar with calendar concepts) to use it to model their basic interactions and functionality of the calendar.

 ISBN: 978-1-61208-283-7

They do not need to be able to model boundary conditions over the possible characters and strings used to express names, dates and other variables for the model elements. These types of low-level details are best handled by test experts who have more direct access to the low-level test platform functionality. The domain experts in this case just need an easy and effective way to express the different objects, fill in valid values and compose them to express their ideas of how the calendar should work, and how the different elements can relate to each other. The test cases generated from their test models and executed against the target system can then be used to validate how well these assumptions hold.

Thus the scope of the language in this case is defined to be exactly what is shown in the example figures in the previous subsection. There is no need to create any more complex properties or model hierarchies for the elements to achieve the set goals.

D. Test Platforms

The test platform for the calendar as described here and provided at [18] is based on different layers as discussed before. The bottom layer is based on a general purpose programming language. It is used to create a basis for the keyword driven layer, which is based on the Robot Framework (RF) test platform. This layer already allows writing test cases using a keyword based language for the calendar application, as shown in **Figure 7**. It allows one to manually compose test cases using such terms as "Add Event" and "Remove Event". However, composing all these elements together manually is still error prone and not an intuitive approach that a domain-expert with no programming background is typically interested to use.

Test Case	Action	Argument	Argument	Argument	Argument	Argument	Argument
Test1	${Event1}=	Add Event	${bob}	18.08.2007 at 21:03:10 EDT	19.08.2007 at 00:57:28 EDT	event1	location1
	Remove Event	${bob}	${Event1}				
	${Task1}=	Add Task	${alice}	27.04.2002 at 02:48:35 EDT	task1		
	Remove Task	${alice}	${Task1}				
	${Task2}=	Add Task	${alice}	26.06.2005 at 04:16:20 EDT	task2		

Figure 7. Calendar script in RF.

A snippet of a test model for the calendar as implemented on top of the OSMO Tester MBT tool is shown in Figure 8.

```
@Guard("RemoveTask")
 public boolean allowRemove() {
  return tasks.size()  > 0;
 }
 @TestStep("RemoveTask")
 public void doRemove() {
  ModelTask task =  tasks.next();
  state.remove(task);
  scripter.removeTask(task);
 }
```

Figure 8. OSMO MBT model snippet.

This shows the part for generating a "Remove Task" test step, basically stating that this step is allowed when some tasks exist to be removed, and when it is taken, an existing task is chosen and removed both from the model state and the system under test state (through the scripter). As described in [14], the names given to these test steps in this type of a test model typically represent domain concepts in domain terminology, and as such provide a basis for a domain-specific test language. In this case, the scripter also generates scripts for the robot framework similar to those shown in **Figure 7**.

The final layer to provide on top of this is the graphical language described in the previous subsections. It can make use of both the model-based testing tool and directly the keyword based language, according to what is available and what is preferred. The main point to take away is that these different layers can be created as required and as seen cost-effective and useful. Different experts with different backgrounds can then use the different layers as they best see fit for their purposes.

IV. DISCUSSION

In general, it is our experience that it typically makes sense to create test languages at different abstraction levels using the different techniques described here when best seen useful. In some cases, it may be enough to just stay at the lowest level and only write unit tests using a general purpose programming language, augmented with some manual testing at the highest level. For example, this is the approach applied with the OSMO Tester MBT tool also mentioned before. It is mainly tested with extensive unit tests and by building a set of test models and test cases manually on top of it. This is possible when domain experts are also technical experts and comfortable with programming tools and techniques. However, this is different in large organizations and with a large number of different stakeholders (managers, customers, domain experts,..) who have a close interest to see and understand and work with the test artifacts.

In some cases, it can be enough to create a simple keyword driven test language to test the applications when it allows expressing all the test cases needed in sufficient detail. Finally, a model-based testing layer can be used on top of the keyword driven layer to provide variation in the generated test cases. However, creating a model-based testing layer or a domain-specific language layer rarely makes sense directly on top of low-level test support such as provided directly by general purpose programming languages (or even general purpose testing languages such as TTCN-3). These modeling tools generate test cases using specific transformations from a source model to a target model. These transformations are made much simpler when built on top of a higher abstraction such as that provided by a keyword driven test framework. By having a layer in between that allows generating test cases as a form or configuration for this (i.e., keyword combinations) makes it much simpler to generate these from the models, simplifies the required transformations, and makes for much better maintenance of the modeling infrastructure. Having a working middle layer (such as keywords based) also allows for writing manual test cases directly on this layer where desired.

 ISBN: 978-1-61208-283-7

For the final part of the domain-specific test languages, it can be built either on top of the model-based testing tools and their test models, or on its own. The model-based testing tools typically do not offer much added abstraction for the domain-specific tools but rather provide a lot of added power in expressing domain variance and generating higher test coverage automatically. When model-based testing tools are used with domain-specific models, it is useful to include them in the loop when possible in order to reduce the costs of maintaining and evolving several different models. That is, using the domain-specific workbench to generate a configuration for the model-based testing tool, which can then generate the actual test scripts from this configuration. This of course depends on having the required support available in the different tools for this type of functionality. Better understanding their relations in a domain-specific context as presented in this paper helps achieve these goals.

V. CONCLUSIONS

This paper presented an overview of domain-specific modeling in the context of software testing. While various approaches for applying test automation in different domains exist, few explicitly consider these two concepts together and aim for most benefit. The overview presented here provides a basis for building better support for making these concepts benefit from each other.

The graphical modeling language definition for the calendar example is available as a MetaEdit+ project on the OSMO Tester MBT tool website [18]. The test platform code can also be accessed as part of the OSMO Tester MBT tool examples code repository at [18].

Overall, although different definitions and approaches are presented here for both domain-specific modeling and test automation, as noted, many different interpretations for these terms exist, and any definition should suffice, as long as it helps the people involved perform the task at hand and the stakeholders are able share the common definition(s). In our experience, being more explicit in the domain terminology and building the test frameworks based on this helps achieve this.

In the future we look forward to more extensive case studies on applying different DSM approach to testing, and how it affects and benefits the different stakeholders.

VI. REFERENCES

[1] S. Kelly, and J.-P. Tolvanen, "Domain-Specific Modeling: Enabling Full Code Generation", Wiley-Blackwell, 2008.

[2] O.-P. Puolitaival, T. Kanstrén, V.-M. Rytky, and A. Saarela, "Utilizing Domain-Specific Modelling for Software Testing," in 3rd International Conference on Advances in System Testing and Validation Lifecycle (VALID 2011), 2011, pp. 115-120.

[3] T. Kanstrén, O.-P. Puolitaival, V.-M. Rytky, A. Saarela, and J. Keränen, "Experiences in Setting up Domain-Specific Model-Based Testing," in IEEE International Conference on Industrial Technology (ICIT 2012), 2012, pp. 319-324.

[4] J. Bentley, "Programming Pearls: Little Languages," Communications of the ACM, vol. 29, no. 8, 1986, pp. 711-721.

[5] MetaCase, "MetaEdit+ Domain-Specific Modeling (DSM) environment," MetaCase, [Online]. Available: http://www.metacase.com/products.html. [Accessed January 2013].

[6] ETSI, "Testing and Test Control Notation version 3," European Telecommunication Institute (ETSI), [Online]. Available: http://www.ttcn-3.org/. [Accessed January 2013].

[7] NSN, "Robot Framework - A generic Test Automation Framework," [Online]. Available: http://code.google.com/p/robotframework/. [Accessed January 2013].

[8] S. Stresnjak, and Z. Hocenski, "Usage of Robot Framework in Automation of Functional Test Regression," in The 6th International Conference on Software Engineering Advances (ICSEA 2011), 2011.

[9] S. Wieczorek, and A. Stefanescu, "Improving Testing of Enterprise Systems by Model-Based Testing on Graphical User Interfaces," in 17tht IEEE International Conference and Workshops on Engineering of Computer Based Systems (ECBS 2010), 2010, pp. 352-357.

[10] S. Bärisch, "Domain-Specific Model-Driven Testing," Vieweg+Teubner Verlag, 2009.

[11] J. Kloos, and R. Eschbach, "A Systematic Approach to Construct Compositional Behaviour Models for Network-structured Safety-critical Systems," Electronic Notes in Theoretical Computer Science, vol. 263, 2010, pp. 145-160.

[12] M. Utting, and B. Legeard, "Practical Model-Based Testing: A Tools Approach", Morgan Kaufman, 2006.

[13] W. Grieskamp, N. Kicillof, K. Stobie, and V. Braberman, "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," Journal of Software Testing, Verification and Reliability, vol. 21, no. 1, 2011, pp. 55-71.

[14] T. Kanstrén, and O.-P. Puolitaival, "Using Built-In Domain-Specific Modeling Support to Guide Model-Based Test Generation," in 7th Workshop on Model-Based Testing (MBT 2012), 2012, pp. 58-72.

[15] A. Jääskeläinen, M. Katara, A. Kervinen, M. Maunumaa, T. Pääkkönen, T. Takala, and H. Virtanen, "Automatic GUI test generation for smart phone applications - an evaluation," in Proceedings of the Software Engineering in Practice track of the 31st International Conference on Software Engineering (ICSE 2009), 2009, pp. 112-122.

[16] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner, "One Evaluation of Model-Based Testing and its Automation," in Proceedings of the 27th International Conference on Software Engineering (ICSE 2005), St. Louis, Missouri, USA, 200, pp. 392-401.

[17] M. Vieira, X. Song, G. Matos, S. Storck, R. Tanikella, and B. Hasling, "Applying Model-Based Testing to Healtcare Products: Preliminary Experiences," in Proceedings of the 30th International Conference on Software Engineering (ICSE 2008), Leipzig, Germany, 2008, pp. 669-672.

[18] T. Kanstrén, "OSMO Tester Home Page," May 2012. [Online]. Available: http://code.google.com/p/osmo. [Accessed May 2012].

 ISBN: 978-1-61208-283-7