Utilizing Domain-Specific Modelling for Software Testing

Olli-Pekka Puolitaival, Teemu Kanstrén
VTT Technical Research Centre of Finland
Oulu, Finland
{olli-pekka.puolitaival, teemu.kanstren}@vtt.fi

Veli-Matti Rytky, Asmo Saarela
Elektrobit Wireless
Oulu, Finland
{veli-matti.rytky, asmo.saarela}@elektrobit.com

Abstract—**Automated execution of manually defined regression tests is a very widely used and well-known area. While test execution can be more easily automated, test case creation and maintenance are still mainly manual efforts and practically the biggest cost factors in software testing. We view writing test cases as basically a programming activity and believe it can thus benefit from extended application of generic programming tools and techniques. In this paper, we describe our work in applying domain-specific modelling (DSM) to the domain of test case creation. DSM is a variability handling method typically applied in software development. It is widely used and powerful method best applied when there are several kinds of variations. DSM is typically tailored to make own optimized modelling solution inside a company, after which it can be applied effectively and without requiring specific programming skills. In this paper we describe how we have applied DSM to describe variability in software behaviour in terms of test cases, and its application in a case study. The results show a reduction in the cost of over test automation.**

Keywords-domain-specific modellin; test automation

I. INTRODUCTION

Test automation these days is a popular concept with an extensive body of knowledge and a large set of mature tools available. The most popular approach in this domain is the automation of test case execution [1]. Automated execution of test cases gives a lot of benefits such as faster execution times, automatic smoke tests after each commit and nightly regression tests. Test cases are typically written in a form of programming language, describing input- and output sequences, data values, and their expected interactions. These test cases are then executed using a test automation framework. This activity of test execution has a long history and a large set of mature testing tools and techniques available. While test execution can be viewed as highly advanced, the largest effort in the software testing process is in manual test case creation and maintenance. As we view test case creation as closely related to general programming activities, we believe it is possible to use more advanced techniques and tools from the domain of software engineering to also enhance the test creation activity.

One way to provide more effective support for test creation and maintenance is to use a higher abstraction level for describing the test cases. In the software engineering domain this is commonly addressed through different modelling techniques. A specific approach for this in the software engineering domain is domain-specific modelling (DSM). In DSM a specific optimized modelling solution is first tailored for a chosen domain by the domain expert and then applied continuously by the domain users [2].

Traditionally, DSM is used to handle variability of product lines. In this case, the different products in a product line are modelled using a common notation and focusing on the differentiating aspects with the optimized, domain-specific modelling language. In our view, this translates very effectively to the domain of test automation and specifically that of test creation.

In software testing, we typically need to exercise the different aspects of system behaviour with different variations. For example, when we have one boundary value that needs to be covered, we need at least three test cases to cover that (below, equal, and above boundary values). Therefore, we view software testing as a domain with high variability and large potential to benefit from the different aspects of DSM. Another related aspect related to this is Model-Based Testing (MBT) [3] that aims to improve test coverage by automatically generating test sets from a system behavioural model utilizing several algorithms. In our previous work, we have described how DSM can be combined to provide added benefits for MBT [4]. However, although advanced test generation techniques such as MBT can be powerful, our experience is that there is still need for manually defined test cases. In this paper, we present our work on using DSM as an aid for more effective creation of test cases. We demonstrate this with the aid of a case study, including the observed cost-benefits achieved.

The rest of the paper is structured as follows. In Section II, we describe the background concepts relevant to the work presented in this paper. In Section III, we describe the case study system used to illustrate the discussed concepts through the rest of this paper. In Section IV, we describe our approach to using DSM to help in test creation. In Section V, we describe the results of the case study and discuss the DSM test modelling approach a wider context. Finally, conclusions summarize the paper.

II. BACKGROUND CONCEPTS

In this section, we describe the background information required to understand the concepts described in this paper.

A. *Test case automatic execution*

With test case automatic execution we refer to a tool chain in which test cases are described in a form of programming language, such as a scripting language, and tests can be run automatically once they have been specified (as defined, e.g., in [1]). This tool chain needs to address the different needs for test components in test automation. This includes providing test input as stimuli for the system under test (SUT), test output as a reference of the expected response, and a test harness to link the test cases themselves to the SUT. A test oracle is a component needed for determining the correctness of the behaviour of software during (test) execution [5]. Together these components form what we term as the test execution environment.

Test automation in this type of an environment takes the following process. First the test scripts are written (typically manually) by a test expert. This includes defining both the test input and expected output in the given notation for the test execution environment. These tests are then executed using the test execution environment and the results are presented to the user.

B. *Modelling for testing purposes*

Modelling for testing purposes can be divided in two categories: modelling test cases, and modelling for test generation. Modelling for test generation refers to creating models that are used as a basis for test generation with techniques such as model-based testing. We have discussed this aspect before in [4], and in this paper we focus on the test case modelling aspects.

Test case modelling refers to modelling specific test cases separately in terms of chosen abstraction level. This can be either textual, graphical or some hybrid notation. In this paper we discuss this in terms of DSM concepts, which are in our case graphical or hybrid notations. Some of the most popular existing graphical notations for test modelling are UML testing profile [6] and TTCN-3 graphical presentation format [7]. These and other generic languages can be widely applied but are not as powerful for the chosen domain as the DSM based notations. In this paper, we describe a case study in using test modelling using DSM tools and concepts.

C. *Domain-Specific Modelling*

DSM is about creating a new modelling language based on domain concepts and using a (typically) self-made code generator to transform this to a different form such as textual source code. These modelling languages are typically easier and quicker to understand for most of the people because they describe the intended domain using higher level domain concepts. These languages can be visual, textual or combinations of both. The modelling language is typically constrained to reduce the modelling options to only those relevant to the expected variance in the domain, which also serves to simplify the modelling process and reduce the number of errors in generated output. Based on our experiences, the DSM are most useful when there is some kind of variability in the product.

The idea in addressing variability in the DSM language is that static part of output (the common part of the target domain) is in generator or in the used platform. Thus the modelling can focus only on the varying aspects of the domain, while the static parts are provided off-the-shelf. Because of this, the support for the dynamic parts can be highly optimized, simplified and made easier to use. A common application domain is with product lines due to variability between products [8] [9].

Domain-Specific Modelling work flow is following:

1. Create a modelling language based on your domain concepts
2. Write a generator which generates your code
3. Create a model using your language
4. Generate the code or document or what you want

Because the code is generated from a model, the debugging can be also made in to the model. Normally the system sends information from its state and the workbench highlighted it to the model. If an error exist it is easy to see in which part of model it happens. There can also be a need to display additional debug data into the model, e.g., performance metrics.

D. *Generating test cases utilizing DSM*

As test cases are typically expressed as scripts using a textual notation, this makes their generation from specific domain test models a viable approach. We can summarize that the main benefits of DSM in the context of test automation are:

- Model is easier to understand because it is expressed using our domain concepts.
- Modelling is faster because it is optimized for the domain and constrained to avoid obvious mistakes.
- Models can be expressed visually, providing for easier to understand test expressions.
- Non-programmer can create test cases.

In this case, DSM can be understood also as a more illustrative user interface for test scripts or a test script visualization method. The structure of DSM for test case generation is illustrated in Figure **1**.

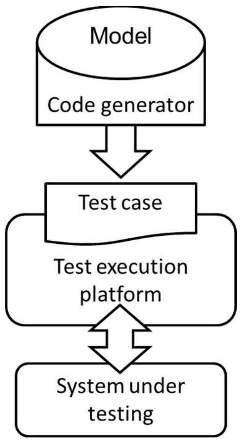

Figure 1. Domain-specific modelling with automated test execution

III. MILITARY PHONE AS TESTING TARGET

The case study system described in this paper is using Elektrobit Tough Voip (ETV) [10] as the SUT. ETV is a military Voice Over Internet Protocol (VOIP) communication device. The device has to be very easy to use and resistive because it is made for military purposes. As failures and bugs in a military system can obviously lead to big problems, the quality and reliability of these devices are a major concern. ETV is presented in Figure **2**.

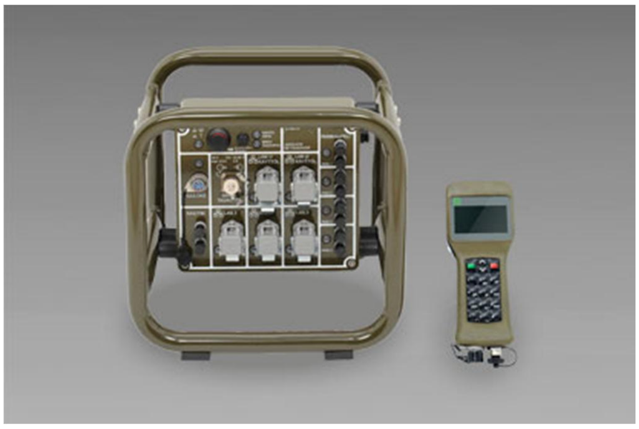
Figure 2. EB Tough VOIP

The ETV has point to "point call" and "all call" features. In a practical deployment setting, all devices have to be connected on the same network and any terminal can connect to another one just by dialling the number of that terminal. This is the "point call" feature. The "all call" feature connects a single device to the all other devices in the network using a specific dialling pattern.

Our viewpoint in testing these devices is that of a high-level abstract black-box viewpoint. From this perspective, the complexity is not in the functional on one particular device, as a single device does not contain highly complex external behaviour. The main complexity is in having many devices connected to the network, calling each other within very tight time limits and in varying sequences. While completely manual test execution is possible, without automation it is hard to address fast time limits or to create test cases for time border values. Therefore, the system also has a TTCN-3 based testing environment for executing automated test cases against the devices.

Besides fully manual test execution, also test creation and maintenance even with an automated test execution environment has its own issues. As noted before, there is a requirement to test various connection- and call-sequences as well as various time limits within these sequences. With the traditional TTCN-3 test scripting and environment, the maintaining of the test suite was found to require a large effort and to require a large amount of TTCN-3 expertise which was found expensive and limited in availability. To address these issues, we created a DSM based solution where the test cases can be expressed in domain concepts without having to work with detailed internals of the TTCN-3 notation.

IV. A MODELLING LANGUAGE FOR TEST CASE GENERATION

In creating a domain-specific modelling language for ETV testing, we used the main domain concepts as a basis. These are the devices themselves, the call types, their ordering and time constraints. The created language is composed of the basic test automation elements of test suite, test case, and test setup. The test suite model is a collection of test cases, the test case model describes test case structure for a single test case, and test setup describes the setup of the device under testing. The overall test model architecture is illustrated in Figure **3**.

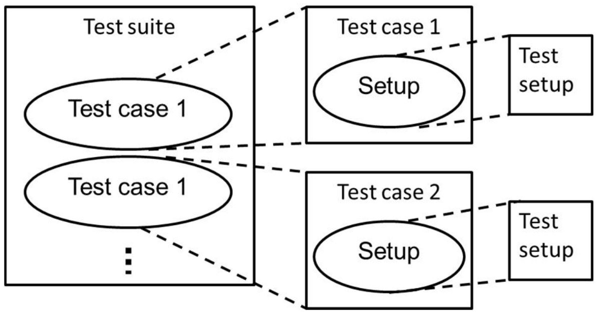

Figure 3. Test model architecture

We used MetaEdit+ [11] as a language workbench for creating our DSM language, and TTCN-3 as the test scripting language. The effort in creating the overall test automation environment was shared with two engineers. The first engineer developed the modelling language (modelling language developer) and the associated TTCN-3 script generator, and the second one manually developed TTCN-3 based test scripts and the overall test case execution process (test case developer).

At the beginning, the test case developer gave a test script sample and initial requirements of modelling language to the modelling language developer. The modelling language developer used these as a basis for creating the first DSM language version. This and the associated test script generator were then evaluated by the test case developer, who made change requests based on the results. Based on this feedback, the modelling language developer made fixes to the language and generator. After one week of iterations, the language and test execution environment was ready for testing in a real test environment.

In the following subsections we describe each of the model elements and an additional test run visualizer component used to show the actual executing test cases.

A. *Test suite model*

The test suite model is a collection of test cases. Test cases are represented as objects in the model and the colour of these objects tells their enabled status. A green object is enabled and red ones are disabled. Figure **4** shows an example of a test suite model, where two of the test cases are enabled and one is disabled.

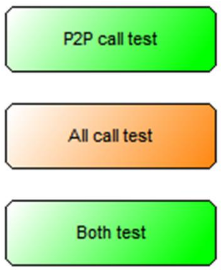

Figure 4. Test suite graph

The test case objects in this model contain sub-models that can be opened to further explore the test case represented by that model. The test case name is presented as the sub model name in the visual presentation shown in Figure **4**. This example has three test cases, one for “point call”, one for “all call”, and one for testing both. The both test graph is presented in the following section.

B. *Test case model*

A test case model is used for representing individual test cases for the EVP system. This model consists of six different types of objects and two types of connection lines. These components are the following:

- *Start object* represents the test case starting point.
- *End object* represents the end of test case.
- *Call object* represents a test step where a device calls to another device or devices including oracle disabling option if time limits are too tight for oracle between the call and next call.
- *End calls object* represent a test step where a single device ends the call.
- *Device object* represents a device including device name and phone number.
- *Test setup object* is a link to the test setup graph.
- *Connection lines* are arrows in to the model representing test case order, calling device and destination device.

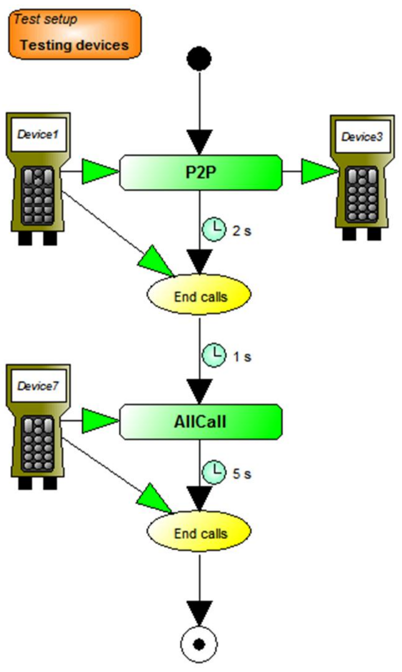

Figure 5. Test case graph

Figure 5 shows an example test case using this notation. It describes the following scenario:

1. Device1 makes a call to Device3
2. After 2 seconds Device 1 ends the call
3. After 1 second Device7 calls to all devices
4. After 5 seconds Device7 ends the call

C. *Test setup model*

A test setup model represents devices in the current test network. In practice, the test setup changes a lot and we do not wish to change all the test cases in case the setup needs to be changed. Instead, we wanted an option that allows us to change the test setup and immediately re-run the existing test cases with a different set of devices and their configurations. In our modeling language, the test setup model is used to represent this type of information and to allow the modification of the different test setups independently of the individual test cases.

In the tests, the test setup model is mainly used for automatic device initialization. The test oracles in the *AllCall* test steps also use the test setup model because they needs to verify the status of all devices that are connected to the network. Figure **6** shows an example of a test setup model in our case study.

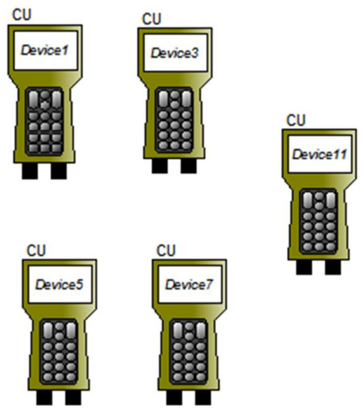

Figure 6. Test setup graph

D. *Test execution visualization*

Showing the execution of test cases is always important for understanding the execution and debugging the results. Initially, our test execution process illustration was just a lot of text running quickly in the command line shell on the test execution platform, which in our case was called Elektrobit Test Tool Platform. This was not a very human friendly form of feedback.

To provide a better support for visualization of test execution, we created visualizations of the DSM test models, showing at all times the current object of the testing as highlighted. This visualization was made using application programming interface (API) of the MetaEdit+ DSM tool, which allows connecting external elements to the model code using the commonly supported SOAP [12] protocol. Practically, the test environment would send SOAP messages to the MetaEdit++ tool, where the specific test DSM language and model received these messages and as a result highlighted the matching elements in the model. For us, this illustrated the possibility to easily and effectively use the test models as tools for following test execution, reporting the test results and debugging possible errors in a human friendly way.

E. Test generation

Test generation is made by MetaEdit+ providing MERL language. MERL is a small programming language and it is decided for code generation providing effective way to go through the model and print the output.

Before starting any generation, the generator checks the most typical errors and reports if some exist. The checks are self-made and based on our experiences. First, the generator makes initializations based on test setup model. Then the generator starts from start state object and goes through the model following the arrows and printing object specific code and finally ends to the end state object.

F. Test model execution

In addition to support the test creation (modelling) process, we also aimed to automate the test case execution itself. Initially, test cases were created and executed manually. In the first phases of our DSM application, we proceeded to generate the test cases for the execution environment from the models which only required manual linking of the generated test cases to the test environment. From this, we further linked the whole test environment into the modelling environment, allowing one to run all tests directly from a single interface and not requiring any direct low-level interaction with the TTCN-3 notation or test environment itself.

Finally, only pressing a single button in the test case or test suite model view is needed. After the test execution, the results are presented. The test execution consists of following steps:

1. User presses test-button in the model.
2. MetaEdit+ generates test cases using the domain-specific test code generator.
3. OpenTTCN tool compiles the test cases.
4. The test platform executes the test cases.
5. During the test case execution, the test platform sends execution information to the MetaEdit+ tool that visualizes the execution in the model.
6. After a test ends, the OpenTTCN provides a test report with highlighted issues that were observed while test execution.
7. Tests engineer check the report, fix from test model and generator or report system bugs to the developers and starts over the testing process.

V. Results and Discussion

Our initial goal in creation of our DSM based test case modelling approach was to enable test case creation for users without requiring specific detailed knowledge of the test environment or the used TTCN-3 notation. However, in practice as experienced in our case study, the test expert still defined most of the test models. This is a person who had also previously been involved in test script creation and manual test case creation for the SUT using the TTCN-3 notation. Thus the test expert already was familiar with the underlying notations and had expertise on the expected SUT behaviour.

However, despite this background we found our results very encouraging. Our experiences are based on about twenty models in real environment and those models are more complex and having some more features than case study models. The test engineer estimated the modelling using our new DSM approach as being at least ten times faster than manual test script writing. The set-up time for creating the modelling notation and the test script generator for this notation was about one week. We used one more week for adding more advanced features and for making our system mature. It took about one week to make changes for the underlying TTCN-3 code to make it easier to generate and to fix bugs we found in it. However, this work was found to improve the overall test system and was not just useful only for our DSM approach. Since our DSM approach needed to evolve to adapt to the actual needs, we added several features to the language. The average time for integrating a new feature was about 30 minutes. In addition, we extended the language to cover other variations of EB Tough VoIP, which took about 3 days. These results were available as we recorded the time we took in the different steps and the company in the case study had also made effort to record this information for their previous approaches.

In our case study, we found that the test case development speed is not the only benefit. The developers and other interest groups assessed the DSM based test case creation and visualization approach as easier to understand. As the test case is graphical, it is easy to see what it does. During the test execution, the progresses are visualized. This saves time in analysing test cases.

In our experience, the development of DSM based test script generation is like normal system development. The DSM method is different but it is just one technology to learn. The DSM modelling workbenches (such as MetaEdit+) are just one form of a programming environment. As these are created specifically to support building this type of modelling notations and environments, the work amount for use has been in the amount of weeks instead of months or years.

The main challenges for applying DSM for test creation are in creating a good modelling language. This requires good understanding of both the application domain and the test automation domain. In this case the modelling language creator and maintainer needs to have quite a wide understanding of the system to create a suitable and powerful test modelling environment. However, this is offset by the reduced need in the test modelling phase where less detailed knowledge of the low-level operations is needed.

Our development style for the case study described in this paper was close to consulting. An external research entity was helping an industrial partner build a DSM test modelling approach and apply it. Thus the people creating the test modelling language and writing the test scripting environment were different. This approach is perhaps not as effective as when entity person is creating both the modelling language and the test scripting environment. However, our experience is that the result is better because of discussion with different entities. In case of a single company this can also be different people from different entities inside the company. During the development people from both entities were describing the solutions to each other and getting feedback on their part. This helped avoid overly complex structures and modelling approaches, as the other entity had to use the results of the other entity in both cases and immediately objected if they found the result too complex for easy adoption.

While results might be different in a different type of an environment and when applied by people with different backgrounds, we believe our result can encourage other people to try our DSM based test modelling and creation approach.

VI. CONCLUSION

In this paper, we described how DSM languages can be used to support the manual test case modelling and creation process, which we observed as one of the most expensive parts in software testing. Using a practical case study, we illustrated the approach in practice. In our experience, the results have been encouraging. The setup effort to create the required modelling languages and environment using existing DSM tools was a couple of weeks and provided more than ten times faster test case creation speed in comparison to previous experience in the case study environment. Thus we can conclude that our modelling approach takes more investment in the beginning but quickly becomes more effective as more test cases need to added and existing ones need to be maintained.

In the future research we need to try this approach in several cases in different domains and get more experiences in its application. We are also using model based testing with DSM test models for reaching even more enhanced test automation.

VII. REFERENCES

[1] Dustin Elfriede, Rashka Jeff, and Paul John, “Automated Software Testing,” Massachusetts: Addison Wesley Longman, 1999.

[2] Kelly Steven and Tolvanen Juha-Pekka, “Domain-Specific Modeling,” New Jersey: John Wiley & Sons, 2008.

[3] M. Utting and B. Legeard, “Practical Model-Based Testing: A Tools Approach,” Morgan Kaufmann, 2007.

[4] O-P. Puolitaival and T. Kanstrén, "Towards Flexible and Efficient Model-Based Testing, Utilizing Domain-Specific Modelling," in 10th Workshop on Domain Specific Modelling, 2010.

[5] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley, "Specification-Based Test Oracles for Reactive Systems," in Proc. of the 14th Internation Conference on Software Engineering, Melbourne, Australia, 1992, pp. 105-118.

[6] OMG. UML Testing Profile. [Online]. http://utp.omg.org/ 15.7.2011

[7] ETSI. TTCN-3 Graphical presentation Format. [Online]. http://www.ttcn-3.org/StandardSuite.htm 15.7.2011

[8] Mika Karaila, Domain-specific Template-based Visual Language and Tools for Automation Industry. Tampere: Tampere University of Technology, 2010.

[9] Kärnä Juha, Tolvanen Juha-Pekka, and Kelly Steven, "Evaluating the Use of Domain-Specific Modeling in Practice," in 9th OOPSLA Workshop on Domain-Specific Modeling (DSM 2009), Orlando, 2009.

[10] Elektrobit wireless. EB Tough Voip. [Online]. http://www.elektrobit.com/what_we_deliver/wireless_solutions/device/products/eb_tough_voip 15.7.2011

[11] Metacase. MetaEdit+ Modeler- Support Your Modeling Language. [Online]. http://www.metacase.com/mep/ 15.7.2011

[12] W3C. (2007, April) SOAP Version 1.2. [Online]. http://www.w3.org/TR/soap12-part1/ 15.7.2011