2014 9th International Conference on the Quality of Information and Communications Technology

Test Process Improvement with Documentation Driven Integration Testing

Florian Häser, Michael Felderer, Ruth Breu
Institute of Computer Science
Quality Engineering
University of Innsbruck
{florian.haeser, michael.felderer, ruth.breu}@uibk.ac.at

***Abstract*—Improving the maturity of the test process in an organization, especially but not limited to integration testing, involves obstacles and risks, such as the additional work overhead of the new process. In addition, integration testing descriptions are often too technical not addressing the language needs of the domain. In research cooperations with companies from the insurance and banking domain it turned out that test descriptions and reports are one of the most useful testing artifacts, while doing ad-hoc testing.**
This paper presents a bottom up testing approach, which first helps the integration tester in producing a semi-formal test description and report, up to be an enabler for automatic model-based testing in the very end. The presented approach is based on a textual domain specific language that is able to evolve over time. This is done by analyzing the test descriptions and reports automatically with machine learning techniques as well as manually by integration testers. Often recurring test steps or used components are integrated into the test language, making it specially tailored for a specific organization. For each test step implementations can be attached, preparing it for the next iteration. In this paper the methodology and architecture of our integration testing approach are presented together with the underlying language concepts.

***Keywords*—Model-Based Integration Testing, Test Process Improvement, Regression Testing**

I. Introduction

Research cooperations with companies from the banking and insurance domain have shown that for integration testing, which is usually done only sparse and in an ad-hoc manner, little effort is required to improve the test process. Documenting the actual test execution, with its parameters and its test results, in a semi-formal way, i.e. ***test configuration, descriptions and reports*** helps to improve the quality of test artifacts. Those artifacts can be used to raise the maturity of the test process.

The companies mentioned before are dealing with fast-paced IT markets and the resulting shorter release cycles involves many risks concerning the quality of software products. Testing is a popular means for mitigating those risks. In order to achieve results of high quality, testing all layers of the software is a crucial aspect. Model-based testing (MBT) enables early validation and definition together with test automation, making it a good choice in overcoming urgent challenges of software testing and making it a good candidate for improving the test process effectively. It is an active area of research [1] and especially suitable to test embedded systems, where safety, security and reliability play an important role. But also when safety is less critical as it is the case with classical business applications, using model-based testing approaches also seems very promising for delivering a qualitative product in the end. A classic business application is implemented through different types of applications in a huge system landscape, usually using different technologies making integration and end-to-end testing difficult.

Introducing however a new strategy like MBT for quality assurance and bringing it to the maturity of the test process in an organization involves some major risks. Starting from managerial problems in finding the right people and project to get started as stated by Katara et al. [2] and Robinson et al. [3], up to showing the return on investment to the executives. Another risk that is introduced by MBT is the added complexity, throughout the whole development methodology. Formal testing methods have often a low learning curve, and require quite a lot of effort and a high discipline when applying them. Often they bring additional work overhead not bearable by the tester as they are already working to capacity. In addition, after the introduction of new tools it is quite hard to reorganize work as many tasks change from e.g. a manual to an automated execution and require new skills.

The arguments mentioned before are valid for all levels of testing. Acceptance and system testing levels, are already well researched and supported in industry, e.g. by behavioral driven testing methods. For integration testing, however, different not that well researched obstacles exist:

- Often **not complete** scenario implemented
- **Mix** between **business** and **technical** roles
- **Few** specialized **tools**
- Tests are often done **manually** based on the testers experience
- Tests are performed under **time pressure**
- Test **reports & documentation**, if any, are **outdated**

978-1-4799-6133-7/14 $31.00 © 2014 IEEE
DOI 10.1109/QUATIC.2014.29

In addition to those major problems integration testing languages need to be able, to express tests in the context of the domain and to cover technical aspects.

This paper presents an approach to improve integration testing, which is able to tackle most of the problems mentioned above. In a nutshell, the approach is a textual test language, which evolves over time. Starting by helping the integration tester in documenting the test runs up to be an enabler for automated model based testing in its final evolution phase.

The remainder of the paper is organized as follows. Section II presents related approaches and discusses their problems. Section III presents the solution proposal. Finally, Section IV concludes the paper and presents future steps.

II. Problems with Related Approaches

In this sections problems that arise with approaches and tools used in industry are highlighted.
Netos investigation on model-based testing approaches [1] has shown that for integration testing only few dedicated approaches exist. Merely automotive and telecommunication domains i.e. embedded software are the primary target for those approaches. Tools and approaches are already very well evolved and used in practice. The major players in this field are the Testing and Test Control Notation version 3 (TTCN-3) [4] in the telecommunication domain, Simulink [5] in the automotive domain and the UML2 Testing Profile (U2TP) [6] for various other domains. Those approaches are also used to test business applications, with minor success however. With the raise of domain specific languages and behavioral driven testing lots of custom tailored test solutions have been developed, for example RSpec [7], which is also the basis for some other languages.

The difficulty of choosing the right test modeling language, and the associated drawbacks, have already been discussed in detail by Hartman et al. [8]. In the following advantages and disadvantages of the major test languages will be presented in the context of the problems as in Section I.

TTCN-3 [4] has been developed, and is used and maintained by the European Telecommunications Standards Institute (ETSI). Despite the fact that it is the official test standard in the telecommunication and automotive domain it is well supported by tools and has already shown that it works also well for large and complex tests. It allows automated test script generation and has been chosen by AUTOSAR (Automotive Open System Architecture) to be the official test language. Even though there exist quite a lot of good tutorials [1] the learning curve is quite flat and one has to invest a lot of effort, with specifying the tests, to get preliminary results. Textual DSLs are well-liked by tester having a strong technical background but also not that great for tester with strong business background. As TTCN-3 is platform and language independent basically it could be used for classical business applications as understood in this paper. Limited time in the testing cycles, however, almost always do not allow the introduction of a heavy weight test methodology as TTCN-3 is. Testing is tight to low level specifications such as protocols and workflows, which is good for integration and system testing, but in order to get the smallest working example a lot of specification has to be done. Moreover, once decided to use TTCN-3 causes a test specification language lock-in, although there exist approaches that allow to transform it to U2TP for example [9]. Tool support is quite good, but open source solutions are merely missing [2].

The UML2 Testing Profile (U2TP) [6] is an extension to the Unified Modeling Language (UML) used for graphically modeling tests. It has become an official standard of the Object Management Group (OMG) covering test architectures, test data and test behavior. The language can be used for visualizing, documenting, specifying, analyzing and constructing test specifications, configurations and reports. Graphical modeling can be error prone and is often seen, as discovered in research cooperations, as extremely time consuming. From the first model to executable tests might take a long time. Good models and the speed of modeling highly depends on the available tools and on the expertise of the tester. The generation of test reports is not explicitly supported [10], bringing no immediate advantage when improving the test process step-wise.

Another important player when it comes to model-based testing especially in the automotive [11] and embedded domain is Simulink. It is basically a tool for modeling, simulating, analyzing and verifying dynamic systems with automatic code generation, and continuous test [5]. It is very well suited for simulating specifications before actually implementing in code to avoid expensive fixes. Besides model in the loop verification also software in the loop and hardware in the loop testing is done with Simulink. The learning curve is however very flat and tester require a lot of expertise. Such a formal tool is very heavy-weighted and not suitable for business applications as of the discussed context. Test reports are however well supported through the Simulink report generator. Simulink is suitable for model-driven-testing but not for improving a deeply deployed test process within a classical development cycle.

The disadvantages that domain specific modeling languages (DSML) bring as discovered by Hartman et al. [8] are also valid for model-based integration testing. Behavioral driven testing tries to mitigate some of the risks and are very suitable for functional system testing. It allows very fast results and helps to improve the test process without introducing a heavy-weight new process. But when it comes to integration

[1] http://www.ttcn-3.org/index.php/learn

[2] http://www.ttcn-3.org/index.php/tools/tools-noncom

testing, the meeting point of the technical and the business world, behavioral driven testing is not the best solution. By definition, it can express only tests against high level requirements not considering interfaces between components and possible failure points. Market research within research cooperations always led to the case that DSML are the first choice of companies when evaluating test process improvement possibilities. This because they allow full control over the language to shape it to the needs of the domain and they support language evolution.

The major disadvantages of the currently used languages, identified from practical and research experience can be summarized as follows:

- long time until first results
- ROI not clearly defined
- either strong technical or business testing languages
- choosing between graphical vs. textual languages is hard
- test data management is handled minimalist

III. Solution Proposal

In this section the approach is presented in detail by first explaining the core idea followed by the language concepts. In the end an attempt how to perform the language evolution, which is a core aspect of the approach, is presented.
In order to overcome the obstacles mentioned above we propose an approach consisting of a textual test language, which is able to evolve over time, as depicted in Figure 1. Starting by helping the integration tester in documenting the test runs, up to be an enabler for automated model based testing in its final evolutionary phase.

Domain specific languages are already used successfully in the system and acceptance testing levels, making it a good candidate for using them also on the integration testing level. A textual modeling language has been chosen, because it enables to mix formal and in-formal elements in various degrees. Moreover, it allows to embed the language of the real domain. In addition, it is highly readable, easily storeable and versionable. It is notable, that in the end test documentation and test specification are expressed using the same language. Furthermore with such a language arbitrary test-case generation can enabled.

A. Documentation Driven Integration Testing Methodology

The documentation driven test methodology as shown in 1 is presented in detail in this section.
The major steps of the evolution are:

S.1	tc_0	Reporting of test activities using a DSL
S.2	$tc_{1...(n-1)}$	Implementation of repeated activities
S.3	tc_n	Automation of the test execution process

In S.1 a test description and report is being written by the integration tester using a provided tool. In the initial phase the supporting DSL is quite informal allowing a lot of natural

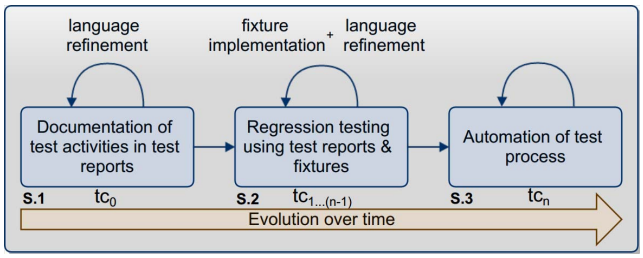

Figure 1. Evolution of Documentation Driven Testing

language as one can see in the example in Section III-C. Its big benefit is to have a storeable test report, that can be used to prove something at a later time or for communication with other stakeholders, such as developers, which are fixing a fault based on the report. In addition it can be of use, when the system changes after a while, when searching for suitable test scenarios. With more test cycles an increasing number of test reports is being collected in the model repository, the evolution of the test language can be performed using machine learning techniques and the aid of the test specialist.
Sub processes that are often performed during test activities are also located during the language evolution. In S.2 the test specialist decides whether a fixture i.e. a piece of executable code, which performs those subprocesses, will be implemented in order to speed up, with a bit of automation, further integration tests performed in the future and in order to have regression tests. Fixtures will be linked to the actual language. After several language improvements and fixture implementation phases a (semi)automation of the test process (S.3) can be performed. The test specialist will be able to model the test cases with a test language, that speaks the language of the domain. Due to the implemented fixtures, the model is executable and automatic test execution can be performed.

B. Documentation Driven Integration Testing Architecture

As depicted in Figure 2 the architecture consists of various components. The core of the approach is built up by the Test-DSL, which consists of a language for specifying the configuration and documentation of the test cases. The language can be interlinked with other languages such as a DSL for describing architectures in order to enable traceability above the limits of the testing domain. Test reports are entered by the integration tester through a special editor. It is linked to the actual DSL and stored in a model repository. After a certain amount has been inputted the model evolution takes place. The test specialist gets suggestions for improvements by the model evolutor component and an improved, i.e. more formal test DSL gets created.

C. Language Concepts

In this section the core concepts of the language are described. Two types of test descriptions are possible, namely one for describing the configuration for certain tests and second for writing the test description and report itself.

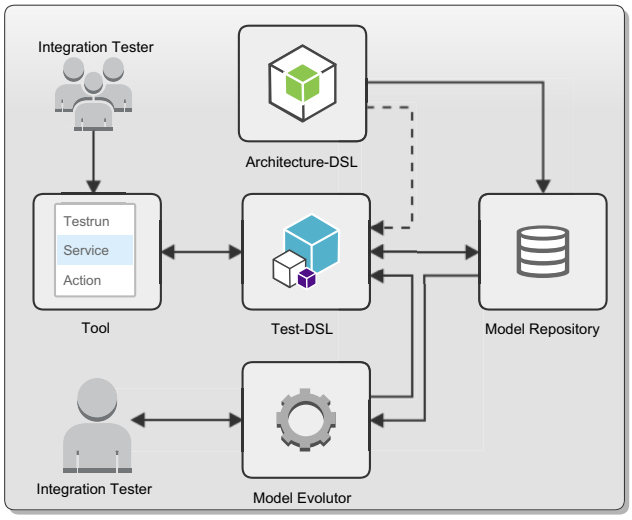

Figure 2. Architecture of Documentation Driven Testing Approach

The language is rooted in three bases. The core of the language is based on previous work of the authors [12], [13] a domain specific test language for service-oriented-systems. Furthermore the success and stability of TTCN-3 [4] and its ability to clearly define components and their interfaces textually was a decisive factor for including those language concepts. Because of the clearly defined and described artifacts of U2TP [6] and its great tool support, also in the open source world, made it an other basis for the test language. The language will have a one-to-one mapping to the U2TP allowing the use of existing tools and tool chains.

Listing 1 presents by an example the relevant concepts for the test configuration. Each test configuration is identified by a *unique name*, such that it can be linked to an actual test documentation and report. Mostly features are developed and therefore tested *release* driven, making it a necessary piece of information in the configuration. This enables traceability, not only for easier search but also for having the concrete information for which release a certain test was performed.
With multi-staged testing environments as it is often the case with larger software systems it may also be of valuable information for which *stage* the configuration is valid, as test outcomes may be different e.g. when measuring execution time in a less powerful stage.
Experience has shown that often, especially in integration testing, it is important to know all *systems* that are *involved* in the test. This is because misbehavior is often propagated and detected in other components then the one that was initially stimulated. Furthermore, in the late evolution stages it is important to know in which interfaces monitoring hooks needs to be placed in order to validate the input and output of certain components in order to detect the exact location of the failure. Integration testing is not forcing to have a complete system in order to start test activities. Additionally, sometimes it is better to have a simpler implementation of a complex component in order to verify whether the rest works as expected. This is usually done using *mocks*, which can be listed within the test configuration.

Listing 1. Test Configuration Language Example

```
configuration<test> "annuityTests" {
  release "2013.2"
  stage "integration"
  involved_systems {
      "ILF"
      "DMS"
      "Batch"
 }
  mocked_systems {
      "Batch"
  }
}
```

Listing 2. Test Documentation & Report Specification Language Example

```
test "prepare␣retirement␣transition" {
  ref "9.2.6" //IT task
  apply configuration "2013.2"
  start 24.03.2014 11:00

  actor "advisor"

precondition {
  if "lv.wish_occupied" and
   <= "vt.begleist␣of␣leading␣insurance"
    -> "lv.wish_occupied"
    else "vt.occupied␣leading␣insurance"
}
postcondition {
 'retirement␣transition␣stored␣for␣batch'
}
process_flow {
      – open dialog 'Scheduled␣Output'
        with workflow default
        R – Prepare Retirement
        Benefit Transition
      – choose contract '1294495'
      – enter startdate 07.09.2013
      – click on save
      – unexpected exception
                -> test termination 11:12
 }
}
```

Listing 2 is an example of a test description and report for the insurance domain. The integration tester writes it while performing the real test on the system. Each test specification is identified by a *unique name* and consists of four major parts.

1) In the header of the, test meta-data is defined.
   Ref can be used to link the test case to a certain task

in the issue management system enabling traceability. To each test a certain test configuration(1) has to be *applied* not only for documentation purposes, but also as an input for the evolutor and later automation.
As in its initial evolution the test documentation & report serves as documentary test report the *start* of test activity has to be logged.
Usually each test case involves one or more *actors* which can be enumerated.

2) The *precondition* can be used to describe in which state the system under test should be in order for the test to be performed.
3) In the *postcondition* the expected outcome or state of the system has to be indicated.
4) After specifying all the boundary conditions the actual test is done on the system under test and logged in the documentation & report, which is done in the *process flow*. Each execution step is indicated with a leading dash followed by a small description on what has been actually tested on the system, including the results.

For the latter three described parts it is the case that in the initial evolution of the test language the content is non-formal natural language. Its primary purpose is to be a digitally storeable test report which is human interpretable. With each iteration those parts become more and more formal, having a machine interpretable artifact in the end, allowing automatic execution of such test documentation and reports.

D. Language Evolution

A core aspect of the approach is the language evolution using the evolutor as depicted in Figure 3. To put it in a nutshell, the evolutor analyzes every test documentation and report that has been created and searches for formalizable expressions, which become part of the language, if allowed by the integration tester who supports the evolution process.

The test specialist triggers the evolutor, whenever an evolution of the language is desired. The evolution algorithm reads all test documentation & reports from the model database, together with the test DSL. A linguistic analysis of the natural language elements of the reports is performed and suggestions for the improvement of the language, for example often used constructs are made to the test specialist. He can decline or accept each of the suggestions. After the selection a new DSL and the associated tool support such as the editor is being generated and the old model instances in the repository are being updated in order to match the meta model.

E. Impact on Test Process

The presented approach can be used, when the test process is still on the maturity levels initial or managed. It helps to document ad-hoc testing preparation and results in a formal, storeable and machine-interpretable way. Once a certain amount of test reports has been collected it can help with test planning, monitoring and controlling. It allows, after some iterations, test generation and execution from a model for defect prevention. The ongoing improvement of the test language helps to optimize the test process and to save precious testing time.

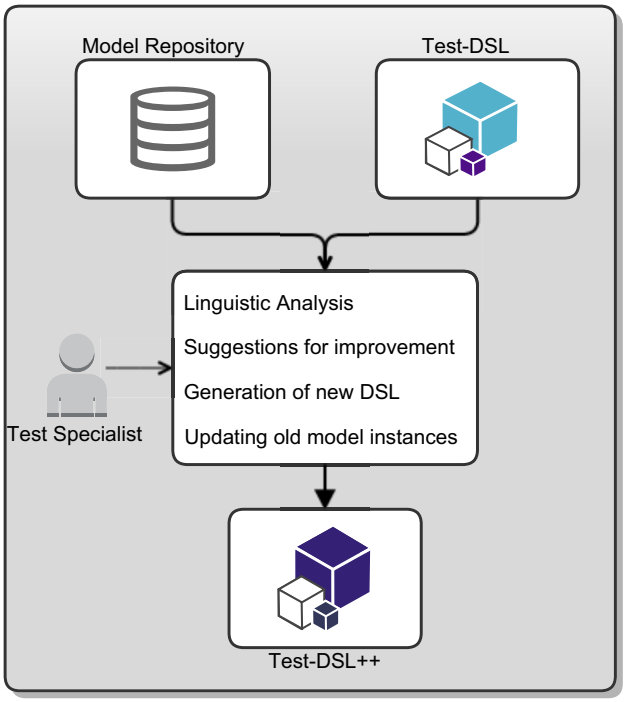

Figure 3. Language Evolution

IV. Conclusion and Future Work

The paper presented a model-based bottom-up testing approach for improving the maturity of the test process, especially when it comes to integration testing but not limited to it. First, it helps the integration tester in producing a semi-formal test description and report, up to be an enabler for automatic model-based testing in its final evolution. The approach counters additional work overhead of the new testing process and it addresses the problem that testing descriptions and reports are often too technical by supporting the language needs of the domain. Research cooperations with companies from the insurance and banking domain have shown that test descriptions and reports are one of the most useful testing artifacts, while having for example a test maturity level 1 or 2 according to Test Maturity Model integration (TMMi) [14], which is the outcome of the approach when in its first evolutionary phase.
As a next step the approach will be implemented prototypically to conduct a case study in the insurance domain in order to get technical and conceptual ideas for future improvements. In addition to the case study, we will include support for testing also non-functional requirements, as this is a very crucial aspect in integration testing.

ACKNOWLEDGMENT

This research was partially funded by the research projects QE LaB - Living Models for Open Systems (FFG 822740) and MOBSTECO (FWF P26194).

REFERENCES

[1] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, "A survey on model-based testing approaches: a systematic review," in *Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007*. ACM, 2007, pp. 31–36.

[2] M. Katara, A. Kervinen, M. Maunumaa, T. Paakkonen, and M. Satama, "Towards deploying model-based testing with a domain-specific modeling approach," in *Testing: Academic and Industrial Conference - Practice And Research Techniques, 2006. TAIC PART 2006. Proceedings*, 2006, pp. 81–89.

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

[4] ETSI. (2014, Jan.) TTCN-3 official homepage. [Online]. Available: http://www.ttcn-3.org/

[5] MathWorks. (2014, Mar.) SIMULINK simulation and model-based design. [Online]. Available: www.mathworks.com/products/simulink/

[6] OMG. (2014, Jan.) UML testing profile. [Online]. Available: http://utp.omg.org

[7] RSpec. (2014, Mar.) RSpec. [Online]. Available: http://rspec.info/

[8] A. Hartman, M. Katara, and S. Olvovsky, "Choosing a test modeling language: A survey," in *Hardware and Software, Verification and Testing*. Springer, 2007, pp. 204–218.

[9] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din, "From u2tp models to executable tests with ttcn-3-an approach to model driven testing," in *Testing of Communicating Systems*. Springer, 2005, pp. 289–303.

[10] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, S. Lucio, E. Samuelsson, I. Schieferdecker, and C. E. Williams, "The uml 2.0 testing profile," in *Proceedings of the'8th Conference on Quality Engineering in Software Technology, Nuremberg (Germany)*, 2004, pp. 181–189.

[11] E. Bringmann and A. Kramer, "Model-based testing of automotive systems," in *Software Testing, Verification, and Validation, 2008 1st International Conference on*. IEEE, 2008, pp. 485–493.

[12] M. Felderer, R. Breu, J. Chimiak-Opoka, M. Breu, and F. Schupp, "Concepts for model-based requirements testing of service oriented systems," in *Proceedings of the IASTED International Conference*, vol. 642, 2009, p. 018.

[13] M. Felderer, J. Chimiak-Opoka, P. Zech, C. Haisjackl, F. Fiedler, and R. Breu, "Model validation in a tool-based methodology for system testing of service-oriented systems," *International Journal on Advances in Software*, vol. 4, no. 1 and 2, pp. 129–143, 2011.

[14] TMMi Foundation. (2014, Apr.) Test Maturity Model Integration (TMMi) Release 1.0. [Online]. Available: ttp://www.tmmi.org/pdf/TMMi.Framework.pdf