A Pattern Language of Test Modelling for Reactive Software Systems

A.-G. Vouffo Feudjio, I. Schieferdecker
Fraunhofer Institute for Open Communication Systems (FOKUS)
Alain.Vouffo, ina.schieferdecker@fokus.fraunhofer.de

Abstract

Patterns have been successfully applied in product software development to improve the development process, by facilitating reuse, communication and documentation of sound solutions. However, the testing domain is yet to benefit from a similar approach. This although, identifying and instrumenting patterns in test modelling to facilitate reuse appears to be a promising approach for reducing the test development cycle. This paper presents a pattern language for test modelling, i.e. a collection of patterns and good practices known for improving the test development process towards more efficiency and shorter development cycles. Further, the paper presents first experiences on applying those patterns in a case study.

1 Introduction

It is now widely acknowledged that testing is no longer just an art, but an engineering discipline in its own, with test development following a similar process as generic software development. Following a process similar to OMG's Model-Driven Architecture (MDA), Model-Driven Testing (MDT) has been advocated as one way of regaining control on complex test system development and is expected to yield comparable benefits for the test development process as those experienced through MDA for product software development. The MDT approach consists in developing abstract platform independent test models which are then transformed in iterative steps into more concrete platform specific test models, until executable test scripts are obtained [1, 23]. Patterns are a method for reusing software methods and artifacts at the design, modelling and implementation level that have been successfully applied in various contexts. Patterns are used to capture experiences, expertise, and facts to improve system quality and shorten the development cycle of software systems. In previous publications, we proposed a similar approach for the design and implementation of tests systems [33]. This is particularly interesting with the growing popularity of MDT, which raises the level of abstraction for test design, to a degree that it allows reuse of concepts for new solutions.

The potential benefits of cataloguing best practices and patterns in test design has been advocated by several authors before. Binder [3] discusses a test pattern template, based on a pattern language of object oriented testing (PLOOT) proposed by Firesmith [14] and introduces a collection of test patterns from the object-oriented software design domain. Meszaros [24] presents a collection of test patterns for unit testing. Howden [19] presents a collection of patterns in selecting tests for maximum error detection. It appears that existing work on test patterns tend to focus on interactions at the object level and are hardly applicable for higher level (i.e. integration, system, and acceptance-level) testing whereby the applied programming paradigm are less relevant. The only collections of test patterns we found that address system-level testing are the one provided by Delano et al [5] and Dustin [9]. However, Delano et al [5] focus more on the organizational aspects of test development as a process, rather than on test design itself. On the other hand, Dustin [9] covers all aspects of test development, with one chapter dedicated

to test design and documentation. in 2005, the European Telecommunications Standards Institute (ETSI) started an initiative on patterns in test development (PTD) in which some of the patterns defined in this work were introduced and discussed. However, to the best of our knowledge, none of the existing work attempts to formalize test patterns, so that they could be instrumented to support the test development process in an automated way. Our work is a continuation of the aforementioned efforts by bringing together test patterns and test modeling concepts.

In this paper, we present a selection of test patterns we have collected so far by performing pattern mining on existing test suites designed using different test modeling and test scripting notations e.g. the Testing and Test Control Notation [16] the UML Test Profile [29] or JUnit [21]. We have compiled those patterns into a pattern language for the modelling of tests for reactive software systems. Each test pattern is defined along a template we introduced in previous work [33], but which was refined to align with generic pattern methodologies. This work is organized as follows: Section 2 presents a selection of the test patterns we have identified, before Section 3 reviews the IMS case study along which the test pattern language has been successfully applied. The paper concludes with an outlook for further research.

2 A Test Pattern Language for Black-box test design

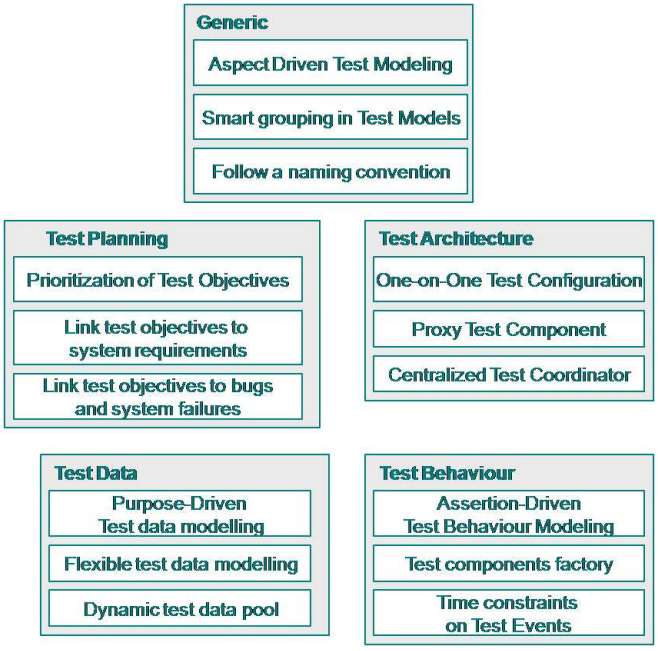

Figure 1. Overview of Identified Test Patterns

We consider test design to be a process that starts from the definition oftest objectives via abstract test models through to executable test scripts. Therefore, we classify test patterns along the various activities of that process into the following categories:

- *Generic test modeling patterns* are those applicable to all activities of test design.
- *Test objectives modeling patterns*: According to IEEE 829 [22], test objectives[1]

  ”identify and briefly describe the special focus or objective for a test case or a series of test cases.”

[1]Test objectives are sometimes also referred to as *test purposes*, *test requirements* or *test directives* in the literature

Therefore, *test objective modeling patterns* are used to model the requirements towards the system under test (SUT) the tests will have to assess. Test objectives can be viewed as the equivalent to system requirements in product development.

- *Test procedure*[2] *modeling patterns* are those that are applicable when designing the test procedures for each of the test objectives.

- *Test architecture modeling patterns* define good practices and established recommendations in selecting and designing appropriate test configurations. The test configurations describe the topology of a test system, i.e. its composition as a collection of (parallel) test components, interconnected among each other and with the SUT and communicating through Points of Control and Observation (PCOs). Depending on the overall goal of a test e.g. conformance, performance, functionality, robustness, etc., different test configurations are suitable.

- *Test data modeling patterns* describe approaches for designing the data to be exchanged between entities of the SUT and the test system in a given test configuration. Test data does not only mean concrete parameters, attributes or messages, but also constraints defining assertions, based on which data received from the SUT can be evaluated to assign a test verdict to a performed test case.

- *Test behaviour modeling patterns* document approaches and principles for designing the behaviour of test systems, i.e. the patterns of interactions between entities in a test configuration.

Pattern mining for these test design patterns can be driven by one or both of the following questions:

- Question 1: What is the best way for designing tests, so that they would help uncovering as many errors of the SUT as possible, before it is delivered to end customers [19]?

- Question 2: What is the best way for designing and modelling tests so that the resulting test specification and/or solution matches best main quality criteria such as reusability, maintainability, understandability etc.?

A pattern template is a list of subjects (sections) that comprise a pattern [3]. The content of the test pattern template depends on which of question 1 or question 2 above is the main driving force for pattern mining. In [3], Binder proposes a test pattern template, which is driven by question 1. Our pattern mining activities are mainly driven by question 2, although question 1 is considered as well. Therefore, we took the test pattern template provided by Binder [3] as the base for our own template, but modified it to reflect the fact that our focus is more on reuse than on effectiveness of the tests. The main difference between our test pattern template and the ones proposed in other work is that we have removed sections that play a less important role, when the focus is on quality of the test design, on the test model and on efficiency in the test development process. For example, the the subjects fault model, entry criteria and exit criteria proposed by [3] play a role for code-oriented, white-box testing, are however not/less relevant for function-oriented, black-box testing, for which test models are being constructed. Instead, we added however the *applicable test scope* to capture the preconditions for applying test model patterns. Our test modelling pattern template consists of the following subjects:

- *Pattern name*: A meaningful name for the test pattern.

- *Context*: To which specific context does it apply? This includes the kind of test pattern (organizational vs. design, generic, architectural, behavioural or test data etc.) as well as the test scope for [3] .

[2] In previous works, the term *test strategy* was used in this context, but we changed it to be in-line with IEEE 829 [22]

[3] The test scope describes the granularity of the item under test [26], which may vary from a low-level entity such as class (for unit testing) to a whole software system (for system testing).

-
- *Problem*: What is the problem, this pattern addresses and which are the forces that come into play for that problem?
- *Solution*: A full description of the test pattern including examples of applications.
- *Known Uses*: Known applications of the test pattern in existing test solutions (e.g. test specifications, test models, test suites, or test systems) or by test modelling approaches.
- *Resulting context*: What impact does this pattern have on test design in general and on other patterns applicable to that same context in particular?
- *Related patterns*(optional): Test design pattern related to this one or system design patterns in which faults addressed by this test pattern might occur. This section is optional and will be omitted, if no related pattern can be named.
- *References*(optional): Bibliographic references to the pattern. This section is also optional and will be omitted, if no reference can be provided.

The test patterns refer to the following test modelling languages:

- The UML Testing Profile (UTP [29]) is defined as a UML profile for designing, visualizing, specifying, analyzing, constructing and documenting the artifacts of test systems. UTP offers test specific stereotypes for test data, test architectures and for test behavior.
- The Testing and Test Control Notation (TTCN-3 [16] standardized by the European Telecommunications Standards Institute (ETSI)) defined as a test specification and test implementation language for the modelling and execution of automated tests. TTCN-3 uses a composition of test components that stimulate and observe the SUT and that locally evaluate the correctness of the SUT reactions. The overall test case verdict is calculated from the test component verdicts.
- The Test Purpose Language (TPLan [32] also standardized by ETSI is a notation for defining test purposes (i.e. test objectives).

The test patterns have been mined from various test suites and test solutions developed by us or being publicly available. The test suites include

- the TTCN-3 Common Library [31]
- the collection of IPv6 test suites [31] used e.g. for the IPv6 logo brand
- the IMS benchmark test suite [7] used for performance testing IMS server equipment
- the CORBA component test suite [2] used for integration testing of CORBA components

To facilitate the application of the identified test patterns, they have been “implemented” into a UML meta-model representing a test-specific notation called Unified Test Modeling Language (UTML [13]). Based on that meta-model, a tool set is generated to guide test modelers in defining test specifications along the test patterns.

Figure 1 displays an overview of those patterns we have identified so far for each of the categories mentioned above. In the following section, we present a selected subset of those patterns.

2.1 Pattern: Aspect Driven Test Modeling

2.1.1 Context

This pattern

- This pattern is a generic test modeling pattern and is applicable at any test scope.
- This pattern is applicable for large size test projects.
- It is assumed that test development is process running in parallel to the development of the SUT, with both of them having the requirements as a common starting point.

2.1.2 Problem

A test model covers both static and dynamic aspects of the system that is to be tested. The size and the complexity of test models as well as the derived test systems can grow considerably. Keeping an overview and a clear understanding of the test model can turn into a difficult task. How can it be ensured that, despite such size and complexity,the test model is kept understandable and maintainable?

Forces:

- Test models are intended to be used as means for communication between several entities in the development process (system implementers, test designers, system designer, test implementers etc.).
- The test modeling activity should not turn into a bottleneck to the test development process. It should be ensured that different teams can work on the same test model to speed up that process, if required.
- Synchronization issues between the people involved in test modeling should be kept at a minimum.

2.1.3 Solution

Spread the test modeling activity over several test designers. Each aspect of test modeling needs to be addressed separately with all test modeling activities possibly being conducted in parallel. This is possible, if the test modeling notation provides the ability for creating and importing modules into each other, according to the Separation of Concerns(SoC) pattern known from Object-Oriented software design. Make separate modules for each of the aspects(e.g. Test data, test architecture) to be addressed and let each of those be a compilation unit, which can be validated independently from others.

2.1.4 Known Uses

Instantiations of this test pattern can be observed in numerous test suites specified using one of the previously mentioned notations. Generally the approach consists in creating separate test models for test data, test architecture and for test behaviour.

- UTML's modularization and import mechanism:
- TTCN-3 test suites defines the *import* keyword, through which existing TTCN-3 compilation entities (modules) can be imported entirely or partly into new ones.The code snippet below from the IPv6 conformance test suite [31] displays an example in TTCN-3 of a test script importing elements of other tests scripts (modules) to build a test suite

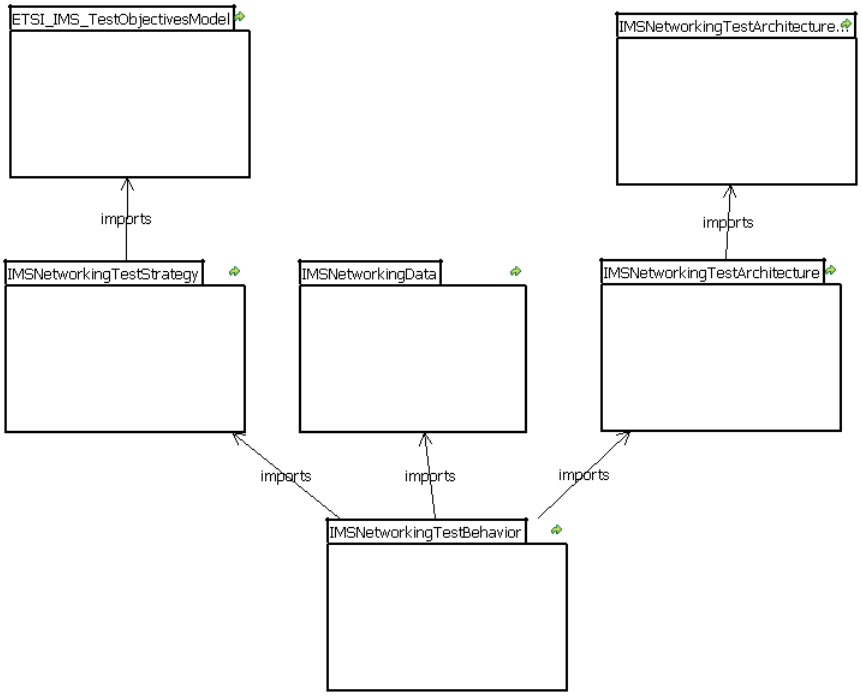

Figure 2. Example of UTML Test Model illustrating the *Aspect Driven Test Modeling* Pattern: The main test model comprises several different test models, each dedicated to a specific aspect of the test process. The arrows indicate that a given test model (starting point) imports elements from another one (end point)

```
1  module AtsIpv6_Common_Functions {
2   // Importing Generic Libraries
3   //LibCommon
4   import from LibCommon_BasicTypesAndValues all;
5   import from LibCommon_DataStrings all;
6     . . .
7   // Importing test data modules
8   //LibIpv6
9   import from LibIpv6_Interface_Templates all;
10  import from LibIpv6_CommonRfcs_TypesAndValues all ;
11    . . .
12  // Importing test architecture modules
13  //AtsIpv6
14  import from AtsIpv6_TestSystem all;
15  import from AtsIpv6_TestConfiguration_TypesAndValues all ;
16    . . .
17 } //end module AtsIpv6_Common_Functions
```

- Most of the existing scripting and programming languages used for testing (Python, Java, Perl etc.) provide a modularization and import mechanism, that can be used to implement this pattern.
- TPLan also defines an import mechanism to include items defined in existing files into new ones.

```
1 #include 'root/MyTPLan/MyDefs.txt'
```

2.1.5 Discussion

A difficulty in applying this pattern consists in ensuring that the number of separate modules remains within sensible limits. Otherwise, the effort of managing all parallel activities can reduce the positive impact of the pattern and even lead to less productivity. This is particularly the case, if different people work on the same files at the same time, leading to issues of version controlling. In such cases the usage of an appropriate version controlling system, along with clearly defined policies is highly recommended. An alternative to this pattern would be to put everything into one module. Which would make the issue of synchronization between parallel developers/designers even more critical, because all entities involved would be accessing and modifying the same resource in parallel and one problem in the main module would create a blocking point for the whole test development process.

2.1.6 Related Patterns

This pattern is an application of the Separation of Concern, a.k.a *Divide and Conquer* design pattern known both in generic software engineering, as well as in test design [9].

2.1.7 References

2.2 Pattern: Prioritization of test objectives

2.2.1 Context

- This pattern addresses test objectives modeling

2.2.2 Problem

Due to resource limitations, often not all test cases can always be developed and/or executed at a time. How to design test objectives, so that key decisions can be taken confidently in the testing process. Key decisions include:

- When can test activities be considered sufficient to provide a level of confidence in the SUT, that is high enough to allow its release?
- Which test cases need to be implemented and executed first and which ones can be left aside for later stage in the testing process?

2.2.3 Solution

As recommended by IEEE 829[22], introduce a prioritization scheme for test objectives in the test model. Prioritization should be provided for a test objective taken individually or for a group of test objectives. Test implementation and test execution can then be planned based on the priority level of the test objectives, to ensure that test cases with highest priority are available on time before product delivery.

2.2.4 Applicable Test Scope

This test pattern is applicable to any test scope

2.2.5 Discussion

Again, the size of the testing project and the time constraints it faces should be taken into account, whenever the application of this pattern is considered.

2.2.6 Known Uses

Prioritization of test cases is used implicitly in several instances, though it is not always supported by the test notation itself. Generally a separate tool is used to manage that aspect of the test process. However, it would be highly beneficial to integrate it into the test modeling process, so that appropriate tool support can be used to provide guidance for more efficiency.

The UTML notation provides the capability to select a priority level to be assigned for each specified test objective and group thereof. Figure 3 displays a screenshot on which the *priority* field is being set for a UTML test objective.

2.2.7 References

[10, 11, 8]

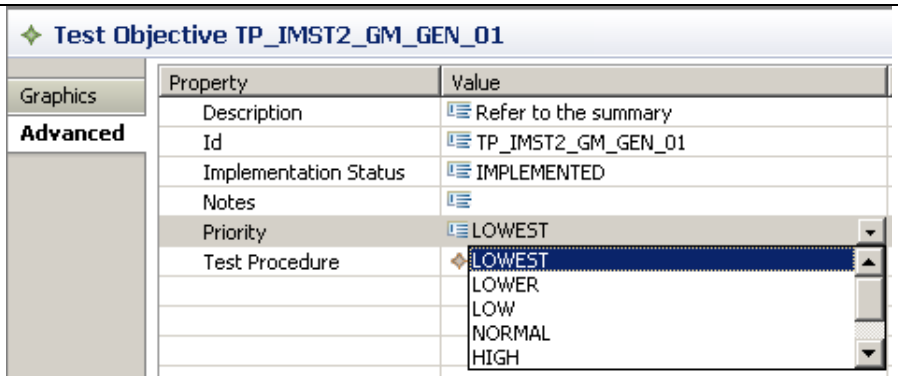

Figure 3. Example of Test Objective Prioritization

2.3 Pattern: Linkage of Test Objectives to System Requirements

2.3.1 Context

- This pattern addresses the modeling of test objectives.

2.3.2 Problem

How to achieve confidence, that the SUT has been tested sufficiently. Forces:

- 100% code coverage is an illusion
- 100% requirements coverage is achievable, but needs to proven.
- Would it be possible at any time to give an estimation of the current coverage of requirements by the specified test objectives? If so, that information might facilitate decision making for releasing the product.

How to achieve traceability between tests and system requirements to enable automatic coverage analysis?

2.3.3 Solution

Provide a mean for linking each test objective to a (set of) requirements or features of the SUT that it addresses. Those requirements could be functional or non-functional. The test objectives could represent a risk in relation to the feature or a mean for verifying that the SUT meets the requirements

2.3.4 Applicable Test Scope

This pattern is applicable to any test scope

2.3.5 Known Uses

Known uses of this pattern include:

- The UTML meta-model's *TestObjective* element defines a reference to a series of requirements the specified test objective covers
- The TPLan [15] notation also provides a similar concept in its syntax definition

- Some model-based testing tools generate test objective descriptions from state diagrams of the system under the test (e.g Conformiq's QTronic too [20]). Since, the test objectives are a result of a transformation process from the system model, a link to system requirements is also possible, provided those requirements can be mapped to certain paths in the state automaton.

2.3.6 Discussion

Benefits of this pattern include:

- The pattern facilitates the selection of test cases to address specific products or features based on the requirements they support.
- Traceability facilitates coverage analysis of the test cases versus the requirements.

One key difficulty in applying this pattern is to ensure that changes to the test model are propagated in both directions of the link to avoid dead links and keep the test model consistent. The test modeling tool should take care of that and update a test objective element accordingly, if one of the covered system requirements is altered (e.g. deleted, moved to another location, renamed etc.). Such a propagation of changes could be facilitated by the usage of the same notation or of the same modeling technology (e.g. EMF, MOF) for those aspects being linked with each other. Otherwise, some serious maintainability issues might emerge.

2.4 Pattern: Linkage of Test Objectives to Failures, Faults and Errors

2.4.1 Context

In spite of all testing efforts, errors in software are inevitable and will eventually occur.

2.4.2 Problem

We want to avoid experiencing and fixing the same errors many times. How can it be ensured, that the information gathered in analyzing and fixing errors identified at the user end or through testing can be exploited for the benefit of future testing activities and for improving the overall quality of the software product under test?

Forces:

- Fixing errors is generally granted higher priority than documenting them.
- Besides, who cares about fixed issues?
- Developers lack of time to do such additional activities. So they tend to postpone them until they pop-up again as higher priorities.

2.4.3 Solution

Provide a mechanism for creating links between the test objectives and the fault management system. The mechanism should fullfil the following requirements:

- The mechanism should be integrated in the test development/management tool to ensure that it can the process does not cost too much additional effort.
- Every time a failure is (inadvertently or deliberately) discovered on a version of the SUT, a test objective should be created in the test objectives model to cover that defect.
- Provide technical means for enforcing that policy automatically online (i.e. in the process of creating the entries in the model repository) or offline (after the elements have been created)
- Automatically integrating the newly added tests in subsequent regression tests would yield additional benefits.

2.4.4 Known Uses

- Testopia [25] is a test case management extension to the well-known bug management tool Bugzilla. However, due to time constraints, we have not used Testopia yet, to analyze to what extent it applies this pattern. It is likely, that similar other tools exist on the market, but we have not got the opportunity to look into those for further analysis yet.

2.4.5 Discussion

The same type of potential issues identified for the *linkage of test objectives to system requirements* pattern (section 2.3) also apply for this pattern.

2.5 Pattern: One-on-One Test Configuration

2.5.1 Context

- This pattern addresses test architecture modeling.
- The SUT can be viewed as one entity providing a small set of entry points and interacting with its environment following a sequential non-concurrent behaviour.
- Functional testing at unit or system level is the goal.

2.5.2 Problem

How to model a static test configuration for achieving testing the SUT with highest possible efficiency.
The following forces come into play for this problem:

- Resources planned for testing are generally and straightforward solutions are always welcome.
- The level of complexity of the test system should be kept as low as possible, to avoid a situation whereby the effort for maintaining it would exceed sensible amount.
- Usage of concurrency in the test system increases the risk of introducing erroneous test behaviour and the cost of the test system, because a coordination mechanism is required to control the choreography of test components.

2.5.3 Solution

Design the test system as one single test component connected to the SUT in a way that it can send impulses to it and verify its reactions to those impulses. One usual way of achieving this is by making the test component an inverse of the SUT. Figure 4 displays two examples resulting from applying that pattern. The upper part of the figure shows a test configuration consisting of a single test component that uses one port both for sending impulses to and receiving responses from the SUT to verify its correct behaviour. On the other hand, the lower part of the figure illustrates a test configuration for an SUT providing 4 different entry points for stimuli and responses. Benefits: Having a single test component implies that synchronization mechanisms based on message exchange or

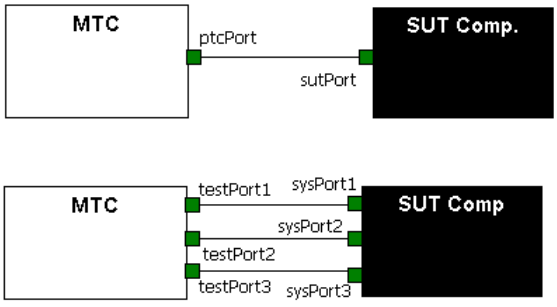

Figure 4. Test architecture Diagram for One-on-One Pattern

other Remote Procedure Control(RPC) or similar mechanisms do not have to be implemented at the testing side. Variables defined in the test component can be used to describe states based on which decisions can be made on the test verdict. Shortcomings: The test component has to emulate the complete behaviour of system component

it replaces. Depending on the level of complexity of that behaviour, this might be more or less difficult to achieve. Furthermore, having a single component makes it difficult to deal with concurrency at the testing side, if required.

2.5.4 Known Uses

This pattern is applied in numerous conformance test suites. For example:

- the collection of IPv6 test suites [31] used e.g. for the IPv6 logo brand
- the IMS benchmark test suite [7] used for performance testing IMS server equipment
- the CORBA component test suite [2] used for integration testing of CORBA components

2.5.5 Discussion

Potential difficulties in handling concurrent behaviour from the SUT and to emulate similar behaviour to stimulate the SUT.

2.5.6 Related Patterns

This pattern is the logical opposite to the *Centralized Test Coordinator* test pattern described in section 2.6. It is also referred to as the *Centralized tester* test pattern [12].

2.5.7 References

[12]

2.6 Pattern: Centralized Test Coordinator

2.6.1 Context

- This pattern addresses test architecture modeling.
- This pattern is more applicable to integration and system testing. It is less the case for unit testing at the class level. However, it can be applied for system testing, whereby a unit testing framework is instrumented for that purpose.

2.6.2 Problem

How to model a test configuration, that is suitable for load- , performance- or conformance testing on an SUT requiring parallel and possibly distributed processing.

2.6.3 Solution

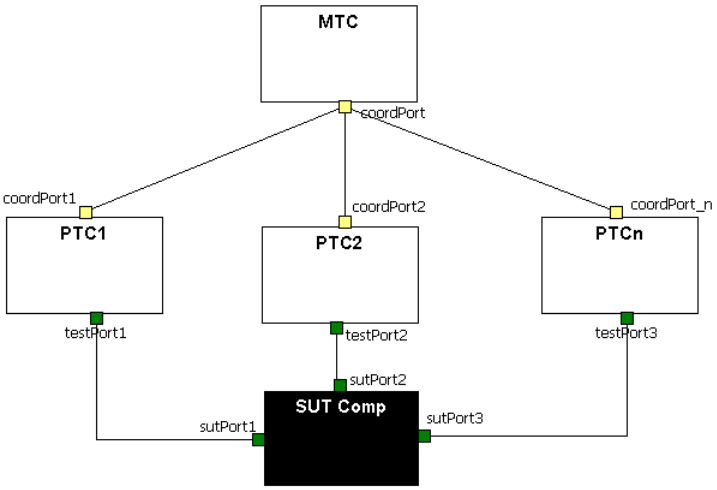

Figure 5. Test architecture Diagram for Centralized Test Coordinator Pattern

As depicted on figure 5, this pattern features a test component acting as test coordinator and thus controlling the life cycle other components it controls. Each of the controlled test components is connected to the controlling component via a connection through which coordination messages can be exchanged to control the components' behaviour. To keep the overhead of processing those coordination messages as low as possible, to not affect the proper test behaviour, coordination messages should be kept as simple as possible in their structure. The real testing activities are performed by the controlled test components, which are directly connected to the SUT.

2.6.4 Known Uses

Several TTCN-3 projects such as [28] involving UTML protocol testing (Siemens) and [6] involving BCMP protocol performance testing.

2.6.5 Discussion

An intelligent coordination pattern is required between the main test component and the parallel test components. The additional load and delays created by that communication should be taken into account while evaluating the SUT component's test results.

2.6.6 Related Patterns

This pattern is the opposite of the *One on One test configuration* pattern defined in section 2.5

2.6.7 References

[12]

2.7 Pattern: Purpose-Driven Test Data Modeling

2.7.1 Context

2.7.2 Problem

How to ensure, that all the test data model elements required for a test suite have been defined ? In large test development projects involving more than one test engineer working on the same test model, there is a need to ensure that redundant data is not defined at many instances in the same test suite, thus negatively affecting readability and maintainability. E.g. in TTCN-3 test suites, too many templates might be defined under different identifiers, although they represent the same test data instances, in terms of functionality. Such redundancy in a test system can affect its understandability and maintainability.

2.7.3 Solution

Assign each defined test data a rule specifying what makes the test data unique and what purpose it fulfills in the test suite. Additional benefit might be obtained by using a machine processable notation for specifying the rule associated to the data instance. For that purpose, an assertion language such as OCL can be used. Based on that rule, before a new test data would be added to the test model, it can be checked automatically, if another test data meeting the associated criteria does not exist yet in the test model and a warning issued accordingly.

2.7.4 Applicable Test Scope

This pattern is more applicable to integration and system testing. For unit testing, the efforts implied would outweigh the potential benefits of applying the pattern.

2.7.5 Known Uses

- The UTML associates each test data instance with a set of constraints it meets
- The Classification Tree Method (CTM) and similar class partitioning approaches for data generation are applications of this pattern.

2.7.6 Discussion

A post-analysis of the test model [27] can also help identifying and addressing this problem.

2.8 Pattern: Reusable Test Data Definitions

2.8.1 Context

- This pattern addresses test data modeling.

2.8.2 Problem

How to facilitate reuse of already defined test data artifacts?

2.8.3 Solution

To facilitate reuse of already defined test data artifacts (i.e. both types and instances) the test modeling notation should provide a mean for referring to existing test data elements in the test model. The relationship between the original test data artifact and the new one can be based on extension or restriction. An extension means, all the rules of the original remain valid, but are extended with new additional rules. E.g. for a data type definition an extension may consist in adding an additional field to the existing structure of the type. On the other hand, a restriction maintains the structure of the original data artifact as-is, but adds new constraints to it. An example of constraint would consist in making mandatory a field that was previously defined as optional in a data type definition.

2.8.4 Applicable Test Scope

This test pattern is applicable to unit, integration and system testing

2.8.5 Known Uses

Mechanisms for extending/restricting existing test data artifacts are provided by several test notations (e.g. XSD, TTCN-3, UTML). Classical object inheritance, as supported by several object-oriented programming languages can also be instrumented to implement a similar result, in situations whereby they are used for test scripting.

2.8.6 Discussion

If the test notation supports a form of inheritance, it will facilitate the application of this pattern.

2.8.7 Related Patterns

This test pattern can use the *Default values*, the *Boundary values* and the *Domain partitioning* testing techniques described in numerous publications to instantiate the initial set of reusable test data definition.

2.9 Pattern: Dynamic Test Data Pool

2.9.1 Context

2.9.2 Problem

Statically defined test data restrict the coverage of the tests, because they increase the risk of ignoring certain areas of the testing domain. Certain tests require for test-data to be generated dynamically, at test execution time, on an "on-demand" basis. This can be very useful in situations whereby it would be too costly to define all test data statically. Furthermore that enhances the quality of the tests, as each set of test will be created very specifically for the objective to be addressed by that test case.

2.9.3 Solution

A dynamic test data pool is an entity which can be called via a predefined API to generate test data dynamically, i.e. during test execution. For that purpose, the data pool is provided a set of criteria, which the generated data is supposed to fulfill, and based on which an appropriate test data instance will be selected or generated to be returned to the calling entity. For expressing the criteria on the test data, a constraint notation such as the OMG Object Constraint Language (OCL) or any other similar notation is recommended to allow automated processing.

2.9.4 Applicable Test Scope

This Pattern is applicable to any test scope

2.9.5 Known Uses

- This pattern is applied by IBM Rational Tester to generate test data based on class partitioning. [30]
- The Classification Tree Method (CTM) [18] follows a strategy similar to this pattern, with each branch of the classification tree representing a constraint fulfilled by the associated data.

2.9.6 Discussion

The mean for defining the selection criteria of test data instances is a critical aspect of this pattern. The chosen notation should base on a clearly define syntax to allow the criteria to be processed automatically while selecting matching data instances. Using natural language instead would significantly reduce the impact of the pattern.

2.9.7 Related Patterns

This pattern is sometime combined with the *class partitioning* pattern, whereby equivalent classes are defined and test data are dynamically generated for each class, based on its defining criteria.

2.10 Pattern: Assertion-Driven Test Behaviour Modeling

2.10.1 Context

- This pattern addresses test behaviour modeling.
- This pattern can be applied to unit, integration and system testing.
- Test models are intended to be used by different types of persons involved in the product development process.

2.10.2 Problem

In many situations, it is very difficult to understand and to communicate what a test case is actually checking on the SUT and how it achieves that task. This adds another level of difficulty to communication between the entities involved in testing or affected by it. How to ensure, that the intent of each test case can be quickly understood, without having to navigate too deeply into the test script's source code?

2.10.3 Solution

While modeling each test case, the focus should always be laid on what behaviour is expected from the SUT for that particular test case. Even if an erroneous behaviour is expected, then the *positive path* for the test case is the one to be visible from the test case's design and implementation. We define the positive path in a test script for a test case as the one leading to a *PASS* verdict. That means, there should be no "positive" *FAIL* verdict. Furthermore, unexpected behaviour in testing should not be modeled explicitly in test behaviour model, but should rather be handled implicitly by some exception handling or similar mechanism, based on the expected behaviour's model. Otherwise the test model loses in readability and maintainability.

2.10.4 Known Uses

- The UTML notation makes extensive use of this pattern to define templates of test actions, which can then be combined to design a complete test behaviour.
- Several TTCN-3 test suites define functions for key actions in the test scenarios and invoke those functions in the test cases, instead of putting all the details of those actions at the highest level of the source code, i.e. the test case level. Key actions include sending an impulse to the SUT, receiving a given response from the SUT, checking that SUT's response meets some defined constraints and assertions etc.
- The TTCN-3 *altstep-default* mechanism can also be viewed as an application of this pattern. A TTCN-3 *default* behaviour is one that is used as alternative whenever an explicitly specified behaviour does not occur. Activating/deactivating a TTCN-3 *altstep-default* switches it on/off as possible alternative behaviour.
- xUnit (JUnit, HTTP-Unit . . .) use this test pattern. In JUnit and frameworks based on JUnit, the test cases mainly consist of assertions to be verified on objects and values from returning methods. If any exception is thrown in the process a *FAIL* or an *ERROR* verdict is set for the test case.

2.10.5 Discussion

2.10.6 Related Test Patterns

This pattern provide the base for all xUnit Test Patterns [24]. Also it is widely used as *Assertion-Based Verification* for various software domains ranging from UML to embedded systems.

2.10.7 References

xUnit Test Patterns [24]

2.11 Pattern: Timing Constraints in Test Behaviour

2.11.1 Context

- This pattern addresses test behaviour modeling
- Unpredictable behaviour can occur in the test execution process, triggered by the SUT, the test system or elements of the test environment.
- Test execution should be automated as far as possible to reduce costs and save time (night runs)

2.11.2 Problem

Situations of deadlock or livelock during test executions should be avoided to ensure that one test case does not block the execution of the remaining test campaign. Exceptional situations in test scenarios involving interactions between test components among themselves or with SUT components must be handled appropriately and take such unpredictable behaviour into account.

2.11.3 Solution

Define timing constraints on test actions involving more than one component. E.g. for each action representing an impulse to an SUT component or an expected response, provide a timing constraint to allow the test system to recover, if the action does not complete smoothly. The timing constraint can be defined via a timer which is started shortly before the action is started and which would trigger an event, if it expires before the action has completed as expected.

2.11.4 Known Uses

- In TTCN-3 a so-called guard-timer can be used to define a timing constraint for an expected signal on a test component. The guard timer's expiration, while waiting for a reaction from the SUT, triggers an event, that can be handled to set the test verdict accordingly.
- Real-Time TTCN-3 [4] proposes to extend the TTCN-3 notation with the concept of this pattern.
- The UTML notation applies this pattern by attaching a timer specification to every specification of an event expected as response from an EUT.

2.11.5 Related Patterns

This pattern is equivalent to the *latency* test pattern mentioned in [4] and used in performance testing of various kinds of servers (e.g. web, application etc.).

3 Evaluation of the Approach: IMS Case Study

3.1 MDTester: A Pattern-Oriented CASE tool for testing

Integrating patterns in a process requires a suitable concept, that would allow the creation of new artifacts in a flexible and efficient way, while at the same time ensuring that the rules defined by the patterns are followed in the creation process or can be verified afterwards. Domain Specific Modeling Languages (DSML) provide a good mean for integrating patterns to a given process. Firstly, because they operate at a level of expression, that

Test Pattern	Implementation Status	Application to Case Study
Separation of concerns	Yes	Yes
Grouping of concerns	Yes	Yes
Naming convention	Yes	Yes
Prioritization of test objectives	Yes	No
Linkage of test objectives to system requirements	Yes	Yes
Linkage of test objectives to failures	Yes	No
One on One	Yes	No
Proxy test component	Yes	Yes
Centralized test coordinator	Yes	Yes
Purpose-driven test data modeling	Yes	Yes
Flexible test data definition	Yes	Yes
Dynamic test data pool	No	No
Focus on expected test behaviour	Yes	Yes
Test component factory	No	No
Time constraints	Yes	Yes

Table 1. Summary of Test Patterns and Status

is abstract enough to express concepts in a solution-independent, but yet formal manner. Secondly, because they can be tailored precisely to define model templates and associated rules, that are specific to the targeted process' domain. Therefore, to evaluate the impact of the patterns we presented in section 2, we defined a UML MOF Meta-model for a DSML dedicated to black box test engineering. The particularity of this DSML is that, tests are modelled based on meta-elements representing the patterns we mentioned earlier.

3.2 The IMS testing case study

Following an MDE process, we developed MDTester, a CASE (Computer-Aided Software Engineering) tool to support pattern-oriented test engineering. The MDTester tool was used to design functional tests for the IP Multimedia Subsystem (IMS) architecture. Table 1 lists all test patterns and their implementation status in the prototype tool, as well as their application to the IMS case study test model.

The impact of model-driven and pattern-oriented test development can be analyzed from a quantitative and a qualitative view point. The purpose of quantitative analysis is to evaluate how productivity is affected by the introduction of the methodology. On the other hand, qualitative analysis aims at measuring the effect on quality factors, both of the process itself and of its output, i.e. the generated test scripts. The goal of the case study was to analyse both the qualitative and the quantitative aspects of that impact and at the same time, to compare the results with those obtained with a "traditional" test development approach.

3.2.1 Quantitative Analysis

A key metric for quantitative analysis of any development process is productivity. Evaluating the productivity of pattern oriented test development is a relatively straightforward task. For that purpose, we simply have to correlate the output (e.g. number of implemented test cases) to the invested effort (e.g. number of person-days/person-months involved) for a project or a series of projects. However, to measure the impact of introducing a new

Project Duration(Days)	Produced Test cases	Productivity (Test cases/-Day)
5	19	3.8

Table 2. Results of Applying Pattern-Oriented Test Engineering to IMS Case Study

approach on that productivity is a less trivial task, because productivity data before and after the introduction of the new approach need to be compared with each other. Ideally, to ensure a fair comparison, at least the following conditions need to be fulfilled:

- Both methodologies should be applied on the same case study: The starting point for both test development approaches should be the same system specification or test plan, targeting the same SUT
- Separate teams should apply the methodology, each on its side in a separate project.
- The same time frame will apply to both projects and results will be collected at the end for evaluation.
- Both teams should have comparable level of expertise in their respective field.

However, we could not provide such an ideal setup for our IMS case study. Therefore we had to base our quantitative comparison on assumptions resulting from statistical analysis of past TTCN-3 test development projects. Table 2 summarizes the results obtained, after applying the pattern-oriented test development methodology on the case study. Taking into account that the project duration was set to 5 person-days and that a total result of 19 test cases were implemented at its end, productivity factor is 19/5 = 3.8 test cases/day. It should be pointed that, this result was obtained with team of designers with a rather low level of testing and modelling expertise. Therefore, it can be assumed that slightly higher results would be obtained with experienced test designers.

To measure the productivity gain generated by our approach, we compare our results with those generally obtained through “traditional” test development approaches. Figure 6 depicts the evolution of productivity gain,

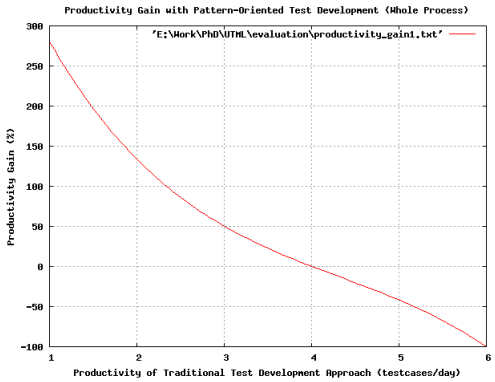

Figure 6. Productivity Gain From Pattern-Oriented Test Development, without taking into account the impact of Test Objectives and Test Procedures

depending on the productivity obtained without pattern-oriented test development. Generally, for TTCN-3 test development, realistic estimations of productivity range between 2 and 5 test cases/day. Therefore,the plot on Figure 6 indicates that, if the existing process allows a production rate of more than 4 test cases/day (including test objectives definition, test procedure design and documentation), then applying our methodology would rather

cause a productivity loss. On the other hand, the productivity could be significantly improved (30 to 90%), when the production rate of the existing methodology is between 2 and 4 test cases/day. Moreover, if we estimate that,

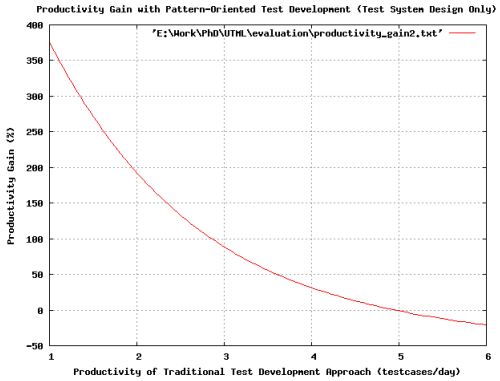

Figure 7. Productivity Gain From Pattern-Oriented Test Development Based on Pure Test System Design

the specification of a test plan (test objectives) and of test procedures consumes 20% of the effort in pattern-oriented test development and are generally not taken into account, when estimating the productivity of the test development process, then the productivity gain is even higher, as depicted on Figure 7.

3.2.2 Qualitative Analysis

Using model-driven approach to test development offers a wide range of qualitative benefits, compared to traditional development approach. Test models offer a higher level of readability, maintainability, documentation and flexibility that plain test scripts and non-formal notations. Furthermore, existing MDE frameworks (e.g. Eclipse EMF, TOPCASED) provide a wide range of functionalities for creating, managing, validating and transforming models, that can be used to provide powerful tool chains to support the process. However, a source of general concern is the quality of the test scripts generated automatically from the process. For our case study, we used the TRex [34] tool to measure the quality of the generated TTCN-3 test scripts. The authors of TRex define a metric called *Template coupling* (ranging between 1 and 3) to measure the maintainability of TTCN-3 scrips. The automatically generated IMS test scripts scored 1.015 on that metrics, indicating the high level of maintainability of those scripts (1.0 is best).

4 Conclusion and Outlooks

This paper has presented first ideas on a pattern language of test design based on a template defined for that purpose. First experiences with that prototype tool chain have shown some promising results. However, model-driven test development has not reached a high level of popularity yet. Therefore, some of the patterns described here can only be considered as mere candidates and will require further analysis with regard to their usability and their consequences. Also, we have presented a case study in which those patterns have been applied to develop tests for IMS. An analysis of the approach through that case study indicates that it can significantly improve the test process, both quantitatively and qualitatively. In the future, we intend to conduct further case studies to analyze the impact of the approach, when developing tests for other domains.

5 Acknowledgements

We would like to thank our shepherds Uwe Zdun and Christian Kohls who were both very helpful in the process of improving this paper through their challenging comments, their patience as well as their interesting and challenging ideas.

References

[1] Paul Baker, Zhen Ru Dai, Jens Grabowski, ystein Haugen, Ina Schieferdecker, and Clay Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer, Berlin, 1 edition, 2007. [cited at p. 1]

[2] Harold J. Batteram, Wim Hellenthal, Willem A. Romijn, Andreas Hoffmann, Axel Rennoch, and Alain Vouffo. Implementation of an open source toolset for ccm components and systems testing. In Roland Groz and Robert M. Hierons, editors, *TestCom*, volume 2978 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2004. [cited at p. 4, 13]

[3] Robert V. Binder. *Testing Object Oriented Systems: Models, Patterns and Tools*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999. [cited at p. 1, 3]

[4] Zhen Ru Dai, Jens Grabowski, and Helmut Neukirchen. Timedttcn-3 a real-time extension for ttcn-3. In *Testing of Communicating Systems*, pages 407–424. Kluwer, 2002. [cited at p. 21]

[5] David E. Delano and Linda Rising. System test pattern language copyright 1996 ag communication systems corporation permission is granted to make copies for plop '96., 1996. [cited at p. 1]

[6] Sarolta Dibuz, Tibor Szabó, and Zsolt Torpis. Bcmp performance test with ttcn-3 mobile node emulator. In *TestCom*, pages 50–59, 2004. [cited at p. 14]

[7] George Din. An ims performance benchmark implementation based on the ttcn-3 language. *Int. J. Softw. Tools Technol. Transf.*, 10(4):359–370, 2008. [cited at p. 4, 13]

[8] Hyunsook Do, Gregg Rothermel, and Alex Kinneer. Prioritizing junit test cases: An empirical assessment and cost-benefits analysis. *Empirical Softw. Engg.*, 11(1):33–70, 2006. [cited at p. 8]

[9] E. Dustin. *Effective Software Testing. 50 Specific Way to Improve Your Testing*. Addison-Wesley, 2003. [cited at p. 1, 7]

[10] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002. [cited at p. 8]

[11] Sebastian Elbaum, Gregg Rothermel, Satya Kanduri, and Alexey G. Malishevsky. Selecting a cost-effective test case prioritization technique. *Software Quality Control*, 12(3):185–210, 2004. [cited at p. 8]

[12] M. Frey et al. Etsi draft report: Methods for testing and specification (mts); patterns for test development (ptd). Technical report, European Telecommunications Standards Institute (ETSI), 2005. [cited at p. 13, 15]

[13] Alain-Georges Vouffo Feudjio. Model-driven functional test engineering for service centric systems. In *Proceedings of The 5th International Conference on Testbeds and Research Infrastructures for the Development of Networks and Communities (TridentCom 2009)*. IEEE, 2009. [cited at p. 4]

[14] D.G. Firesmith. Pattern language for testing object-oriented software. *Object Magazin*, 1996. [cited at p. 1]

[15] ETSI ES 202 553: Methods for Testing and Specification (MTS). Tplan: A notation for expressing test purposes. Technical report, European Telecommunications Standards Institute, Sophia Antipolis, 2007. [cited at p. 9]

[16] Methods for Testing and Specification (MTS). The testing and test control notation version 3; part1: Ttcn-3 core language. Technical report, European Telecommunications Standards Institute (ETSI), 2003. [cited at p. 2, 4]

[17] Dorothy Graham, Erik van Veenendaal, Isabel Evans, and Rex Black. *Foundations of Software Testing: ISTQB Certification*. Int. Thomson Business Press, 2006. [cited at p. -]

[18] Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993. [cited at p. 18]

[19] William E. Howden. Software test selection patterns and elusive bugs. In *COMPSAC '05: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05) Volume 1*, pages 25–32, Washington, DC, USA, 2005. IEEE Computer Society. [cited at p. 1, 3]

[20] Antti Huima. Implementing conformiq qtronic. In *TestCom/FATES*, pages 1–12, 2007. [cited at p. 10]

[21] Andy Hunt and Dave Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, September 2003. [cited at p. 2]

[22] IEEE. Draft ieee standard for software and system test documentation (revision of ieee 829-1998). Technical report, IEEE, 2008. [cited at p. 2, 3, 8]

[23] A. Z. Javed, P. A. Strooper, and G. N. Watson. Automated generation of test cases using model-driven architecture. In *AST '07: Proceedings of the Second International Workshop on Automation of Software Test*, page 3, Washington, DC, USA, 2007. IEEE Computer Society. [cited at p. 1]

[24] Gerard Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Addison-Wesley, 2007. [cited at p. 1, 19, 20]

[25] Mozilla.org. Testopia. Web Page, 2009. [cited at p. 11]

[26] Helmut Neukirchen. *Languages, Tools and Patterns for the Specification of Distributed Real-Time Tests*. PhD thesis, Dissertation, Universität Göttingen, November 2004 (electronically published on http://webdoc.sub.gwdg.de/diss/2004/neukirchen/index.html and archived on http://deposit.ddb.de/cgi-bin/dokserv?idn=974026611 . Persistent Identifier: urn:nbn:de:gbv:7-webdoc-300-2), November 2004. [cited at p. 3]

[27] Helmut Neukirchen and Martin Bisanz. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. In *Proceedings of the 19th IFIP International Conference on Testing of Communicating Systems and 7th International Workshop on Formal Approaches to Testing of Software (TestCom/FATES 2007), June 26-29 2007, Tallinn, Estonia. Lecture Notes in Computer Science (¡a href='http://www.springer.de/comp/lncs'¿LNCS¡/a¿) 4581.*, pages 228–243. Springer, Heidelberg, June 2007. [cited at p. 16]

[28] Andrej Pietschker. Automating test automation. *Int. J. Softw. Tools Technol. Transf.*, 10(4):291–295, 2008. [cited at p. 14]

[29] OMG ptc. Unified modeling language: Testing profile, finalized specification. Technical report, Object Management Group, 2004. [cited at p. 2, 4]

[30] IBM Rational. Use ibm rational tester for soa quality to add dynamic data to a web service test. Web Page, 2009. [cited at p. 18]

[31] Stephan Schulz. Test suite development with ttcn-3 libraries. *Int. J. Softw. Tools Technol. Transf.*, 10(4):327–336, 2008. [cited at p. 4, 5, 13]

[32] Stephan Schulz, Anthony Wiles, and Steve Randall. Tplan-a notation for expressing test purposes. In Alexandre Petrenko, Margus Veanes, Jan Tretmans, and Wolfgang Grieskamp, editors, *TestCom/FATES*, volume 4581 of *Lecture Notes in Computer Science*, pages 292–304. Springer, 2007. [cited at p. 4]

[33] Alain Vouffo-Feudjio and Ina Schieferdecker. Test patterns with ttcn-3. In *FATES*, pages 170–179, 2004. [cited at p. 1, 2]

[34] Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, and Paul Baker. TRex - An Open-Source Tool for Quality Assurance of TTCN-3 Test Suites. In *Proceedings of CONQUEST 2006 – 9th International Conference on Quality Engineering in Software Technology, September 27-29, Berlin, Germany*. dpunkt.Verlag, Heidelberg, September 2006. [cited at p. 24]