SPECIAL SECTION ON ADVANCES IN TEST AUTOMATION – THE EVOLUTION OF TTCN-3

An approach to quality engineering of TTCN-3 test specifications

**Helmut Neukirchen · Benjamin Zeiss ·
Jens Grabowski**

Abstract Experience with the development and maintenance of large test suites specified using the *Testing and Test Control Notation* (TTCN-3) has shown that it is difficult to construct tests that are concise with respect to quality aspects such as maintainability or usability. The ISO/IEC standard 9126 defines a general software quality model that substantiates the term "quality" with characteristics and subcharacteristics. The domain of test specifications, however, requires an adaption of this general model. To apply it to specific languages such as TTCN-3, it needs to be instantiated. In this paper, we present an instantiation of this model as well as an approach to assess and improve test specifications. The assessment is based on metrics and the identification of code smells. The quality improvement is based on refactoring. Example measurements using our TTCN-3 tool TRex demonstrate how this procedure is applied in practise.

Keywords Test specification · TTCN-3 · Quality model · Code smells · Metrics · Refactoring

H. Neukirchen (✉) · B. Zeiss · J. Grabowski
Software Engineering for Distributed Systems Group,
Institute for Computer Science, University of Göttingen,
Lotzestr. 16–18, 37083 Göttingen, Germany
e-mail: neukirchen@cs.uni-goettingen.de

B. Zeiss
e-mail: zeiss@cs.uni-goettingen.de

J. Grabowski
e-mail: grabowski@cs.uni-goettingen.de

1 Introduction

The *Testing and Test Control Notation* (TTCN-3) [16,22] is a mature standard which is widely used in industry and standardisation to specify abstract test suites. Nowadays, large TTCN-3 test specifications with a size of several ten thousand lines of code are developed [2,13–15]. Like any other large software, such large test specifications tend to have quality problems. The roots of these quality problems are manifold, for example inexperienced test developers [2] or software ageing [39]. Usually, statements on quality deficiencies of test suites are made in a subjective manner. However, to obtain a dependable quality assurance for TTCN-3 test suites, an impartial quality assessment for TTCN-3 test specifications is desirable. Hence, a model for test suite quality is required.

In this article, a method and a tool for quality engineering of TTCN-3 test suites are presented. To this aim, we use an adaption of the ISO/IEC 9126 [26] software product quality model which is suitable for test specifications. For the automated quality assessment of TTCN-3 test suites, we apply metrics and code smells measuring attributes of the quality characteristics that constitute the test specification quality model. *Metrics* are a common means to quantify properties of software. More sophisticated deficiencies in source code structures which cannot be identified by using metrics require a pattern-based analysis. These patterns in source code are described by *code smells*. For the improvement of test suites, we use *refactoring*. A refactoring is "*a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior*" [20]. This means, a refactoring is a behaviour preserving transformation which is able to improve internal source code quality. For an automation of the complete quality assessment and improvement process,

Fig. 1 The ISO/IEC 9126-1
model for internal and external
quality
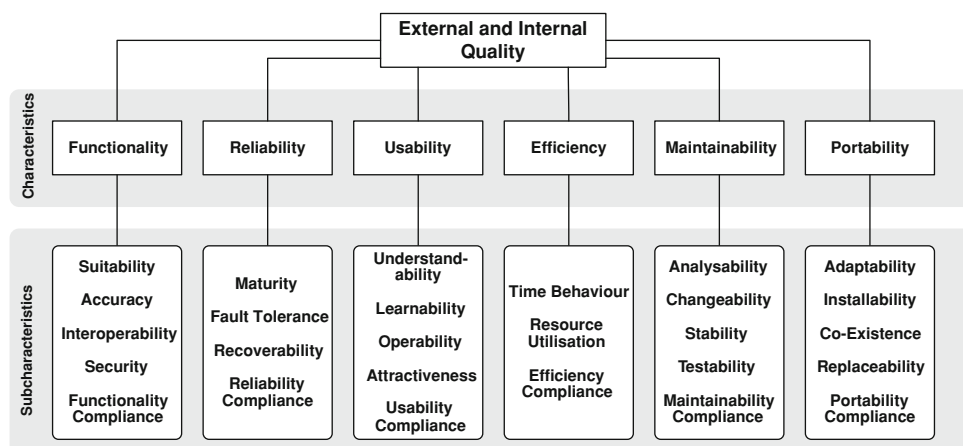
we have developed the TTCN-3 Refactoring and Metrics tool
TRex which is available as open-source software.

This article is structured as follows: Sect. 2 describes the
ISO/IEC 9126 software product quality model and intro-
duces its adaptation to the domain of test specification. Sub-
sequently, Sect. 3 surveys metrics and smells and presents
TTCN-3 specific metrics and code smells. An application
of metrics and smells for the quality assessment of TTCN-3
specifications is demonstrated in Sect. 4. In Sect. 5, the TRex
tool for assessing and improving TTCN-3 test specifications
is presented. Finally, a summary and an outlook are given.

2 Quality of test specifications

Quality models are needed to evaluate and set goals for the
quality of a software product. The ISO/IEC standard 9126
[26] defines a general quality model for software products
that requires an instantiation for each concrete target envi-
ronment (e.g. a programming language). In this section, we
briefly introduce ISO/IEC 9126 and present an adaption to
the domain of test specifications [56].

2.1 Software quality (ISO/IEC 9126)

ISO/IEC 9126 [26] is an international multipart standard pub-
lished by the *International Organization for Standardization*
(ISO) and the *International Electrotechnical Commission*
(IEC). It is based on earlier attempts for defining software
quality [5,30] and presents a software product quality model,
quality characteristics, and related metrics.

Part 1 of ISO/IEC 9126 contains a two-part quality model:
the first part of the quality model is applicable for model-
ling the internal and external quality of a software product,
whereas the second part is intended to model the quality in
use of a software product. These different quality models are
needed to be able to assess the quality of a software prod-
uct at different stages of the software life cycle. Typically,

internal quality is obtained by reviews of specification docu-
ments, checking models, or by static analysis of source code.
External quality refers to properties of software interacting
with its environment. In contrast, *quality in use* refers to the
quality perceived by an end user who executes a software
product in a specific context.

As shown in Fig. 1, ISO/IEC 9126 defines the same generic
model for modelling internal and external quality. This
generic quality model can then be instantiated as a concrete
model for internal or external quality by using different sets
of metrics. The model itself is based on the six characteris-
tics *functionality*, *reliability, usability, efficiency, maintain-
ability*, and *portability*. Each characteristic is structured into
further subcharacteristics.

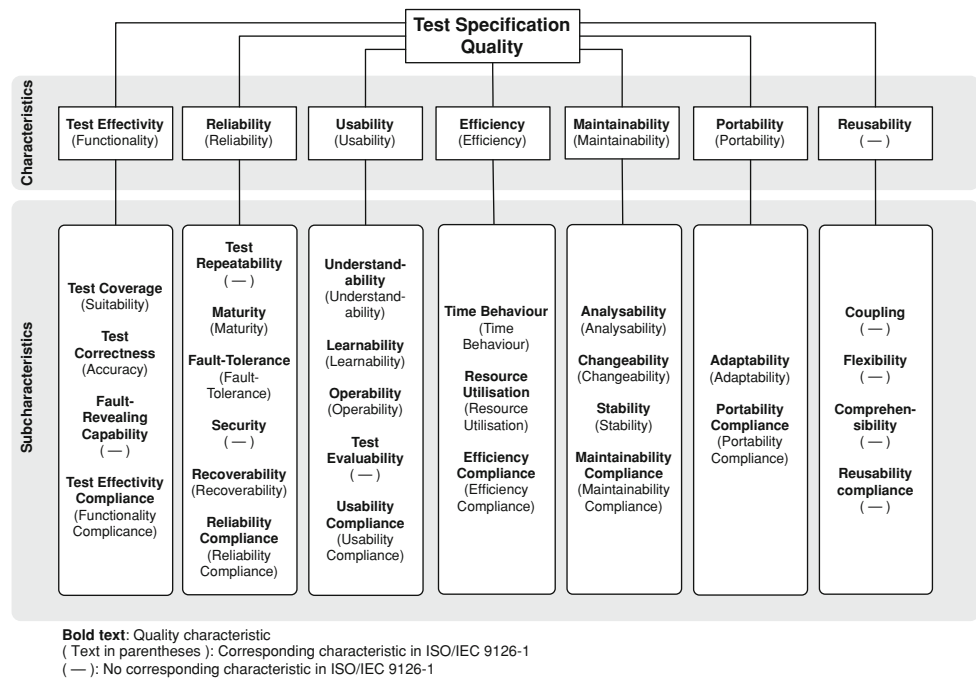The model of quality in use is based on the characteristics
effectiveness, productivity, safety, and *satisfaction* and does
not elaborate on further subcharacteristics. In the further parts
of ISO/IEC 9126, metrics are defined which are intended to
be used to measure the properties of the (sub)characteristics
defined in Part 1. The provided metrics are quite abstract
which makes them applicable to various kinds of software
products, but they cannot be applied without further refine-
ment.

The actual process of assessing the quality of a software
product is not part of ISO/IEC 9126. It is defined in the
ISO/IEC standard 14598 [25]: the assessment requires the
weighting of the different (sub)characteristics and the selec-
tion of appropriate metrics.

2.2 Test specification quality

Our quality model for test specification is an adaptation of
ISO/IEC 9126 to the domain of test specification. While the
ISO/IEC 9126 model deals with internal quality, external
quality, and quality in use, the remainder of this article will
only address internal quality characteristics. Due to our par-
ticipation in standardisation, our primary interest is the qual-
ity of abstract test specifications.

Fig. 2 The test specification quality model

Bold text: Quality characteristic
(Text in parentheses): Corresponding characteristic in ISO/IEC 9126-1
(—): No corresponding characteristic in ISO/IEC 9126-1

An overall view of our quality model for test specifications is shown in Fig. 2. The model is structured into seven characteristics: *test effectivity*, *reliability*, *usability*, *efficiency*, *maintainability*, *portability*, and *reusability*. Each characteristic comprises several subcharacteristics.

Even though most of those characteristics defined in ISO/IEC 9126 can be generously re-interpreted and thus applied for test specifications as well, we preferred to introduce names which are more appropriate in the context of testing. In Fig. 2, the relationship of our model to ISO/IEC 9126 is indicated by providing the corresponding name of the ISO/IEC 9126 characteristics in parentheses. In that figure, test quality characteristics are printed in **bold** letters. Test quality characteristics which have no corresponding characteristic in ISO/IEC 9126, are denoted by the sign (–).

The characteristic *reusability* (right-hand side of Fig. 2) is not considered in the ISO/IEC 9126 model. We added it to our model, because test specifications and parts of them are often reused for different kinds of testing. Thus, design for reusability is an important internal quality criterion for test specifications.

Each characteristic contains a *compliance* subcharacteristic which denotes the degree to which the test specification adheres to potentially existing standards or conventions concerning this characteristic. Since such standards and conventions also exist for test design, a compliance subcharacteristic is also part of each characteristic in our model. However, compliance to standards and conventions will not be covered any further in the following descriptions because they are often company- or project-specific.

In the following, we describe the characteristics and subcharacteristics of our quality model (Fig. 2) in more detail.

Subcharacteristics that are not applicable for test specifications are also reviewed.

2.2.1 Test effectivity

The *test effectivity* characteristic describes the capability of the specified tests to fulfil a given test purpose. The *test effectivity* characteristic corresponds to the *functionality* characteristic in ISO/IEC 9126. It is renamed to emphasise the test specific meaning of this characteristic.

The *test coverage* subcharacteristic relates to test completeness and can be measured on different levels, e.g. the degree to which the test specification covers system requirements, system specification, or test purpose descriptions.

Test correctness refers to the correctness of the test specification with respect to system requirements, system specification or test purposes. Furthermore, a test specification is only correct when it always returns correct test verdicts and when it has reachable end states, i.e. it terminates.

The *fault-revealing capability* is a new subcharacteristic. It has been added, because obtaining a good coverage with a test suite does not make any statement about the capability of a test specification to actually reveal faults. Indicators for increased attention to the fault-revealing capability may be the usage of cause–effect analysis [34] for test creation or usage of mutation testing [6].

The *interoperability* subcharacteristic has been omitted from the test specification quality model. Test specifications are too abstract for *interoperability* to play a role. The *security* subcharacteristic has been moved to the *reliability* characteristic.

2.2.2 Reliability

The *reliability* characteristic describes the capability of a test specification to maintain a specific level of performance under different conditions. In this context, the term "performance" expresses the degree to which needs are satisfied.

The *reliability* subcharacteristics *maturity*, *fault-tolerance*, and *recoverability* of ISO/IEC 9126 apply to test specifications as well.

The new subcharacteristic *test repeatability* refers to the requirement that test results should always be reproducible in subsequent test runs if generally possible. Otherwise, the report of a defect may become imprecise and debugging the *system under test* (SUT) to locate a defect may become hard or even impossible. *Test repeatability* includes the demand for deterministic test specifications.

The *security* subcharacteristic covers issues such as included plain-text passwords that play a role when test specifications are made publicly available or are exchanged between development teams.

2.2.3 Usability

Usability characterises the ease to instantiate and execute a test specification. This explicitly does not include usability in terms of difficulty to maintain or reuse parts of the test specification. These aspects are included in other characteristics of our model.

Understandability covers aspects that establish understanding of a test specification. Documentation and description of the overall purpose of the test specification are key factors for the test engineer to decide whether a test specification is suitable for his or her needs and also to find suitable test selections.

The *learnability* subcharacteristic focuses on understanding for usage. For the proper use of a test suite, the test engineer must understand details of the test configuration, what kind of parameters are involved, and how they affect test behaviour. A comprehensive documentation and the usage of style guides may have positive influence on this quality subcharacteristic.

A test specification has a poor *operability* if it, e.g. lacks appropriate default values, or a lot of external or non-automatable actions are required in the actual test execution. Such factors make it hard to setup a test suite for execution or they make execution time-consuming due to a limited degree of automation.

Test evaluability is a new test-specific subcharacteristic. It refers to the requirement that a test specification must make sure that the provided test results are detailed enough for a thorough analysis. This can be achieved by, e.g. producing meaningful log messages during the test run.

Lastly, the ISO/IEC 9126 characteristic *usability* includes *attractiveness* which we omitted in our quality model. *Attractiveness* is not relevant for test specifications. It may play a role for the user interface of test execution environments and tools, but for plain test specifications, there simply is no user interface involved.

2.2.4 Efficiency

The *efficiency* characteristic relates to the capability of a test specification to provide acceptable performance in terms of speed and resource usage. The ISO/IEC 9126 subcharacteristics *time behaviour* and *resource utilisation* apply without change.

2.2.5 Maintainability

The *maintainability* of a test specification characterises its capability to be modified for error correction, improvement, or adaption to changes in the environment or requirements. This quality characteristic is important, because test specifications are often modified and expanded due to changing product requirements and new product versions.

The *analysability* subcharacteristic covers the ability to examine a test specification for deficiencies. For example, deficiencies may be detected statically by means of code reviews. Well-structured code is a prerequisite for an efficient code review. Further elements that influence the analysability of test code are mandatory style guides or a complete and comprehensive documentation.

The *changeability* subcharacteristic describes the capability of the test specification to enable the implementation of required modifications. Examples for negative impacts on this quality subcharacteristic are unstructured spaghetti code or a test architecture that is not expandable.

Depending on the test specification language used, unexpected side effects due to a modification have negative impact on the *stability* subcharacteristic.

The *testability* subcharacteristic from the ISO/IEC 9126 model does not play any role for test specifications and is therefore removed from our quality model.

2.2.6 Portability

The *portability* plays only a limited role in the context of test specifications, because test specifications are not yet instantiated. Therefore, the subcharacteristics *installability* (ease of installation in a specified environment), *co-existence* (with other test products in a common environment), and *replaceability* (capability of the product to be replaced by another one for the same purpose) are elements of the ISO/IEC 9126 model, but not of our quality model.

However, the *adaptability* subcharacteristic is relevant for our model since test specifications should be capable to be

adapted to different SUTs or test environments. For example, hard-coded SUT addresses or access data in the specification make it hard to adapt the specification for other SUTs.

2.2.7 Reusability

A *reusability* quality characteristic is not part of ISO/IEC 9126. We consider this characteristic to be particularly important for the quality of test specifications, because test suites and parts of them are often reused for different types of tests. For example, the test behaviour of a performance or stress test specification may differ from a functional test, but the test data, such as predefined messages, can be reused between those test suites. As well, parts of a test specification may be reused to test different versions of the SUT. It is noteworthy that the subcharacteristics correlate with the *maintainability* characteristic to some degree.

The *coupling* degree is an important subcharacteristic in the context of reuse. Coupling can occur in-between test behaviour, in-between test data, and between test behaviour and test data. For example, if there is a function call within a test case, the test case is coupled to this function.

The *flexibility* of a test specification relates to its customiseability regarding unpredictable usage. For example, fixed values in a part of a test specification deteriorate its flexibility, and thus a parametrisation likely increases its reusability. Flexibility is furthermore influenced by the length of a specification sub-part, because short parts can usually be more flexible reused in new contexts.

Finally, parts of a specification can only be reused if there is a good understanding of the reusable parts (*comprehensibility* subcharacteristic). Good documentation, comments, and style guides are necessary to achieve this goal.

3 Test quality assessment

The presented test quality model abstracts from how to determine the quality of a test specification with respect to each characteristic and subcharacteristic. Hence, this quality model needs to be instantiated by providing means to measure attributes of a test specification. There are mainly two ways to obtain these attributes: static analysis of a test specification to gather attributes of its internal quality and dynamic analysis on a specification level to gain attributes of its external quality. In the following, quality assessment based on static analysis is presented. Its application is demonstrated in Sect. 4.

3.1 Software metrics

The ISO/IEC 9126 standard suggests to quantify a software product's quality attributes using *software metrics*. As we show later, such metrics are not only applicable for implementations, but also for TTCN-3 test specifications.

According to Fenton et al. [18], the term *software metrics* embraces all activities which involve software measurement. Software measurement can be classified into measures for properties or attributes of *processes*, *resources*, and *products*. For each class, internal and external attributes can be distinguished. *External attributes* refer to how a process, resource, or product relates to its environment (i.e. the ISO/IEC 9126 notion of external quality); *internal attributes* are properties of a process, resource, or product on its own, separate from any interactions with its environment (i.e. the ISO/IEC 9126 notion of internal quality).

Internal product metrics can be structured into *size* and *structural* metrics [18]. Size metrics measure properties of the number of usage of programming or specification language constructs, e.g. the *number of non-commenting source statements*. Structural metrics analyse the structure of a program or specification. The most popular examples are complexity metrics based on control flow or call graphs and coupling metrics.

Concerning metrics for measuring complexity of control structures, one of the most prominent examples is the *cyclomatic complexity* from McCabe [29,51]. The cyclomatic complexity $v(G)$ of a control flow graph G can be defined[1] as: $v(G) = e - n + 2$, where e is the number of edges and n is the number of nodes in G. The informal interpretation is that a linear control flow has a complexity of 1 and each branching increases the complexity by 1, thus $v(G)$ measures the number of branches.

The cyclomatic complexity metric is *descriptive*, i.e. its value can be objectively derived from source code. By additionally using threshold values, this metric becomes also *prescriptive* [17], i.e. it helps to control software quality. For example, when threshold violations of the metric values are analysed, it can help to identify modules with a lot of branching which shall thus be split into several simpler ones [51]. McCabe suggests to use a boundary value of 10. Behaviour with a higher cyclomatic complexity is considered to be too complex and should thus be avoided or restructured.

To make sure that reasonable metrics are chosen, Basili et al. [3] suggest the *Goal Question Metric* (GQM) approach: First, the goals which shall be achieved (e.g. improve maintainability) must be defined. Then, for each goal, a set of meaningful questions that characterise a goal is derived. The answers to these questions determine whether a goal has been met or not. Finally, one or more metrics are defined to gather quantitative data which give answers to each question.

---

[1] Several ways of defining $v(G)$ can be found in literature. The above definition assumes that G has a single entry and a single exit point. In the presence of several exit points, this assumption can be maintained by adding edges from all exit points to a single exit point.

3.1.1 Test metrics

Tests are a special kind of software and thus metrics are also applicable to assess their quality. Most of the known metrics related to tests concern processes and resources, but not products like test suite specifications. From those known metrics which relate to tests as product, most are simple size metrics: Vega et al. [49], for example, propose several internal and external size metrics for TTCN-3 test suites. However, they did not provide the goals and questions related to their metrics, hence it is not clear how these metrics can be interpreted to assess the actual quality of test suites. Sneed [45] provides metrics which abstract from a certain test specification notation. Hence, they are not based on attributes of source code, but on more abstract information. For example, Sneed measures test case reusability by considering the ratio of automated test cases to the overall number of test cases. Some of Sneed's metrics (e.g. test costs) even relate to process or resource attributes. As a result, Sneed's test metrics are not applicable for an assessment of TTCN-3 test specifications on their own.

3.1.2 TTCN-3 metrics

We have investigated metrics for different test specification quality characteristics and came to the conclusion that internal and external product metrics are not only applicable to assess source code of implementations, but as well to assess TTCN-3 test specifications [54–56]. The remainder of this section focuses on internal product metrics for TTCN-3 test specifications.

In addition to the taxonomy of metrics described at the beginning of Sect. 3.1, the metrics which we use to measure the different quality (sub)characteristics of a TTCN-3 test specification can be regarded as either TTCN-3 specific metrics, test specific metrics, or generally applicable (i.e. as well to programming languages) metrics.

TTCN-3 specific metrics　　These kind of metrics take specific concepts of the TTCN-3 language into account. One example is the coupling between test behaviour and test data descriptions which depends on whether behavioural TTCN-3 statements refer to test data using TTCN-3 template references or in-line templates [54]:

Metric 1 (Template coupling) The template coupling TC is defined as:

$$TC := \frac{\sum_{i=1}^{|\mathrm{stmt}|} \mathrm{score}(\mathrm{stmt}(i))}{|\mathrm{stmt}|}$$

where stmt is the sequence of behaviour statements referencing templates in a test suite, |stmt| is the number of statements in stmt, and $\mathrm{stmt}(i)$, $i \in \mathbb{N}$, denotes the ith statement in stmt.

$\mathrm{score}(\mathrm{stmt}(i))$ is defined as follows:

$$\mathrm{score}(\mathrm{stmt}(i)) := \begin{cases} 1 & \text{if } (\mathrm{stmt}(i) \text{ references a template} \\ & \text{without parameters}) \\ & \vee (\mathrm{stmt}(i) \text{ uses wildcards only}) \\ 2 & \text{if } \mathrm{stmt}(i) \text{ references a template} \\ & \text{with parameters} \\ 3 & \text{if } \mathrm{stmt}(i) \text{ uses an in-line template} \end{cases}$$

The template coupling TC measures whether a change of test data requires changing test behaviour and vice versa. The value range is between 1 (i.e. behaviour statements refer only to template definitions or use wildcards) and 3 (i.e. behaviour statements only use in-line templates). For the interpretation of such a coupling score, appropriate boundary values are required. These may depend on the actual usage of the test suite. For example, for good *changeability* a decoupling of test data and test behaviour (i.e. the template coupling score is close to 1) might be advantageous and for optimal *analysability* most templates may be in-line templates (i.e. the template coupling score will be close to 3).

Test specific metrics　　In addition to TTCN-3 specific metrics, we identified metrics which do not relate to TTCN-3 concepts, but to general test specific properties. For example, a test case is typically designed to check a single test purpose. Checking too much functionality in a single test case makes it hard to comprehend the reasons of *fail* test verdicts. Such a test case can be recognised when the ratio between the number of statements that set a verdict and the number of overall statements is unreasonable. Meszaros [31] describes this situation as *test eagerness*.

Metric 2 (Test eagerness) The test eagerness metric TE of a single test case is defined as:

$$TE := 1 - \frac{|vs|}{|\mathrm{stmt}|}$$

where vs is the sequence of all statements that set a verdict in a test case and stmt is the sequence of all statements in the same test case. The range of TE is [0..1]. In general, it is desirable that TE is close to 1 for an average size test case.

General applicable metrics　　The third kind of metrics which we considered, are those which are applicable to source code of implementations as well as to test specifications.

As an example, we evaluated the applicability of McCabe's cyclomatic complexity to the control flow graphs of TTCN-3 behavioural entities, i.e. the **function**, **testcase**, **altstep**, and control part constructs (TTCN-3 keywords are printed in **bold**). We found out that McCabe's metric exhibits the same properties for TTCN-3 test suites as for source code written using general purpose programming languages. Even though TTCN-3 eases the specification of branching using the **alt**

and **altstep** constructs, our measurements show that this does not lead to overly complex control flow graphs [55]. Thus, those behaviours which violate McCabe's upper boundary of ten are actually of low quality and this complexity metric can be used to identify behaviour that is hard to maintain. However, there is one exception: the cyclomatic complexity of control parts is usually in the same order of magnitude as the number of test cases in a test suite. The reason is that a control part often contains many **if** statements querying module parameters to select the test cases to be executed depending on sets of capabilities of the implementation. Hence, it is very probable that the control part of a test suite which consists of more than ten test cases does violate the upper complexity bound. Nevertheless, the structure of such a control part is usually very simple (a linear sequence of **if** statements) and therefore, such control parts cannot be regarded as very error prone or of low quality. Thus for control parts, a metric that is purely based on the nesting level [18] is more appropriate. Alternatively, a possible approach could be to increase the boundary value for control parts by the number of statements which execute a test case and are guarded by a condition [55].

The application of such metrics in order to assess a specific quality characteristic of a test specification requires a tailoring of descriptive metrics into prescriptive ones. Section 4 exemplifies how this can be achieved by introducing such metrics and threshold values tailored to analyse the *changeability* and *analysability* quality subcharacteristics.

3.2 Smells in software

Even though we experienced that metrics are able to detect various issues and can thus be used for an assessment of several quality aspects, some issues cannot by detected by simple metrics. Instead, a complementary, more sophisticated pattern-based approach is needed. This approach is based on so called *smells*. The metaphor of "*bad smells in code*" has been coined by Beck and Fowler. They define a smell as "*certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring*" [20]. Smells are thus indicators of bad quality. According to this definition, defects with respect to program logic, syntax, or static semantics are not smells, because these defects cannot be removed by a refactoring (by definition, a refactoring only improves internal structure, but does not change observable behaviour).

Beck and Fowler present smells for Java source code. They describe their smells using unstructured English text. The most prominent smell is *Duplicated Code*. Code duplication deteriorates in particular the *changeability* quality subcharacteristic: if code that is duplicated needs to be modified, it usually needs to be changed in all duplicated locations.

Smells provide only hints: whether the occurrence of an instance of a certain smell in a source code is considered as a sign of low quality may be a matter that depends on preferences and the context of a project. For the same reason, a list of code structures which are considered as smell is never complete, but may vary from project to project and from domain to domain [12].

The notion of metrics and smells is not disjoint: each smell can be turned into a metric by counting the occurrences of a smell, and sometimes, a metric can be used to locate a smell. The latter is the case for the smell of a long function which can be expressed by a metric which counts the lines of code of a function. However, the above smell of duplicated code and other pathological structures in code require a pattern-based detection approach and cannot be identified using metrics.

3.2.1 Smells in tests

Even though smells were developed with having implementation languages in mind, the idea has also been applied to tests. Van Deursen et al. [7] and Meszaros [31] describe *test smells*. Meszaros distinguishes between three kinds of smells that concern tests (*test smells*): *code smells* are problems that must be recognised when looking at code, *behaviour smells* affect the outcome of tests as they execute, and *project smells* are indicators of the overall health of a project which does not involve looking at code or executing tests. Within this classification, smells of different kinds may affect each other, hence the root cause of a behaviour smell may be a problem in the code. We believe that this classification is reasonable. Most of the test smells identified by Meszaros are behaviour smells, e.g. the smell called *Erratic Test* which refers to tests which are non-deterministic. Some of the test smells presented by Van Deursen et al. are specific to the usage of the *JUnit* Java framework [21] and can thus be considered as code smells while others are more general and can be regarded as behaviour smells. In the remainder, we will restrict our investigations on code smells for TTCN-3.

3.2.2 A TTCN-3 code smell catalogue

Because no TTCN-3 specific code smells have been systematically described, we have developed an initial catalogue of TTCN-3 code smells [4,35]. When investigating candidates for this catalogue, we have relaxed the above definitions of code smells a little bit: we added to that catalogue not only problems in TTCN-3 source code that can be improved by a behaviour preserving refactoring, but as well problems which obviously require a change of the behaviour. One example is an "idle parallel test component" which is created, but never started. In this case, either a TTCN-3 **start** statement needs to be added or the **create** statement needs to be removed. However, we adhered to the above definitions of code smells, in that we did not consider errors in TTCN-3 source code with respect to syntax or static semantics as a smell.

The code smells that are provided by Beck and Fowler [20] were a source for our TTCN-3 code smell catalogue. Even though those smells are intended for Java code, some of them are as well applicable for TTCN-3 code. A further source was our TTCN-3 refactoring catalogue which we have developed earlier [48,53,54]. It already refers briefly to code smell-like quality issues in the motivation of each refactoring.

While the above sources describe smells only in an informal manner, our TTCN-3 code smell catalogue uses a structured representation: each entry is listed in the following format: each smell has a *name*; those smells which are based on other sources have a *derived from* section which lists the corresponding references; a *description* provides a short prose description of the symptom of the smell; the *motivation* part explains why the described code structure is considered of having a low quality; if several variants of a smell are possible by relaxing or tightening certain requirements on a code structure, this is mentioned in an *options* section; one or more actions (typically a refactoring from our TTCN-3 refactoring catalogue) which are applicable to remove a smell are listed in the *related actions* section; finally, a TTCN-3 source code snippet is provided for each smell in the *example* section.

In our catalogue, the names of TTCN-3 code smells are emphasised using *slanted* type and TTCN-3 keywords are printed using **bold** type. To structure our TTCN-3 code smell catalogue, we have divided it into ten sections. This structure is used in the following list which is an overview providing the name and the symptom description of each smell from our TTCN-3 code smell catalogue. So far, we have identified 37 TTCN-3 code smells:

Duplicated code

– *Duplicate statements*: there is a duplicate sequence of statements in the statement block of one or multiple behavioural entities (functions, test cases, and altsteps). Special cases like code duplication in **alt** constructs and conditionals are considered as a separate smell.
– *Duplicate alt branches*: different **alt** constructs contain duplicate branches.
– *Duplicated code in conditional*: duplicated code is found in the bodies of a series of conditionals.
– *Duplicate in-line templates*: there are two or more similar or identical in-line templates.
– *Duplicate template fields*: the fields of two or more templates are identical or very similar.
– *Duplicate component definition*: two or more test components declare identical variables, constants, timers, or ports.
– *Duplicate local variable/constant/timer*: the same local variable, constant, or timer is defined in two or more functions, test cases, or altsteps running on the same test component.

References

– *Singular template reference*: a template definition is referenced only once.
– *Singular component variable/constant/timer reference*: a component variable, constant, or timer is referenced by one single function, test case, or altstep only, although other behavioural entities run on the component as well.
– *Unused definition*: a definition is never referenced.
– *Unused imports*: an import from another module is never used.
– *Unrestricted imports*: a module imports more from another module than needed.

Parameters

– *Unused parameter*: a parameter is never used within the declaring unit. For **in**-parameters, the parameter is never read, for **out**-parameters never defined, for **inout**-parameters never accessed at all.
– *Constant actual parameter value*: the actual parameter values for a formal parameter of a declaration are the same for all references. Hence, this parametrisation is unnecessary.
– *Fully-parametrised template*: all fields of a template are defined by formal parameters. Hence, this template conveys no information on its own.

Complexity

– *Long statement block*: a function, test case, or altstep has a long statement block.
– *Long parameter list*: the number of formal parameters is high.
– *Complex conditional*: a conditional expression is composed of many Boolean conjunctions.
– *Nested conditional*: a conditional expression is unnecessarily nested.
– *Short template*: the body of a template definition is so short that it does not justify the creation of a template declaration.

Default anomalies

– *Activation asymmetry*: a default activation has no matching subsequent deactivation in the same statement block, or a deactivation has no matching previous activation.
– *Unreachable default*: an **alt** statement contains an **else** branch while a default is active.

Test behaviour

– *Missing verdict*: a test case does not set a verdict.
– *Missing log*: **setverdict** is used to set verdict **inconc** or **fail**, but without calling **log**.
– *Stop in function*: a function contains a **stop** statement.

Test configuration

- *Idle PTC*: A *parallel test component* (PTC) is created, but never started.

Coding standards

- *Magic values*: magic values are literals that are not defined as constants or as part of templates. Numeric literals are called Magic Numbers, string literals are called Magic Strings.
- *Bad naming*: an identifier does not conform to a given naming convention.
- *Disorder*: the sequence of elements within a module does not conform to a given order.
- *Insufficient grouping*: a module or group contains too many elements.
- *Bad comment rate*: the comment rate is too high or too low.
- *Bad documentation comment*: a documentation comment does not conform to its format.

Data flow anomalies

- *Missing variable definition*: a variable or **out** parameter is read before its value has been defined. This smell is also known as UR data flow anomaly [19,23].
- *Unused variable definition*: a defined variable or **in**-parameter is not read before it becomes undefined. This smell is also known as DU data flow anomaly [19,23].
- *Wasted variable definition*: a variable is defined and defined again before it is read. This smell is also known as DD data flow anomaly [19,23].

Miscellaneous

- *Over-specific runs on*: A behavioural entity (function, test case, or altstep) is declared to run on a component, but uses only elements of this component's super-component or no elements of the component at all.
- *Goto*: a **goto** statement is used.

To give an impression, how an entry of our TTCN-3 code smell catalogue looks like, the *Duplicate Alt Branches* smell is presented in detail. In addition to the already mentioned style of type setting TTCN-3 keywords in **bold** and names of TTCN-3 code smells in *slanted*, refactoring names from our TTCN-3 refactoring catalogue [48,53,54] are printed in *slanted* type as well. Please refer to the complete TTCN-3 code smell catalogue [4,48] for a detailed description of all TTCN-3 code smells.

Smell: *Duplicate alt branches*

Derived from: TTCN-3 refactoring catalogue [48,53,54] (Motivations for *Extract Altstep*, *Split Altstep*, and *Replace Altstep with Default* refactorings).

Description: different **alt** constructs contain duplicate branches.

```
1  testcase tc_exampleTestCase1() runs on ExampleComponent {
2          timer t_guard;
3          //...
4          t_guard.start(10.0);
5          alt {
6                  [] pt.receive(a_MessageOne) {
7                          pt.send(a_MessageTwo);
8                  }
9                  [] any port.receive {
10                         setverdict(fail);
11                         stop;
12                 }
13                 [] t_guard.timeout {
14                         setverdict(fail);
15                         stop;
16                 }
17         }
18 }
19
20 testcase tc_exampleTestCase2() runs on ExampleComponent {
21         timer t_guard;
22         //...
23         t_guard.start(10.0);
24         alt {
25                 [] pt.receive(a_MessageThree) {
26                         pt.send(a_MessageFour);
27                 }
28                 [] any port.receive {
29                         setverdict(fail);
30                         stop;
31                 }
32                 [] t_guard.timeout {
33                         setverdict(fail);
34                         stop;
35                 }
36         }
37 }
```

Listing 1 Duplicate alt branches

Motivation: code duplication in branches of **alt** constructs should be avoided just as well as any other duplicated code. Especially common branches for error handling can often be handled by default altsteps if extracted into an own altstep beforehand.

Related action(s): Use *Extract Altstep* refactoring to separate the duplicate branches into an own altstep. Consider refactoring *Split Altstep* if the extracted altstep contains branches which are not closely related to each other and refactoring *Replace Altstep with Default* if the duplicate branches are invariably used at the end of **alt** constructs as default branches.

Example: In Listing 1, both test cases contain **alt** constructs with three alternatives. The last two alternatives in both **alt** constructs (lines 9–16 and lines 28–35) are identical and can be extracted into a separate altstep.

Most of our TTCN-3 code smells are detectable using static analysis; however, some of the code smells related to test behaviour can only be detected using a dynamic analysis.

The applicability of each TTCN-3 code smell depends on the specific project and personal preferences. For example,

when developing a library, occurrences of the *Unused Definition* smell are tolerable, because definitions provided by a library are usually not referenced until the library is reused by other TTCN-3 modules.

The smell descriptions can be used to detect either individual occurrences of a smell or for an overall quality assessment by introducing additional metrics that count occurrences of smells. Such complementing metrics will be described in the next section together with results from applying this unified metrics calculation and code smell detection approach to three large TTCN-3 test suites.

4 Application of metrics and smell detection

For the quality assessment of TTCN-3 test suites, we use a unified approach in which smells are also represented by metrics. To demonstrate the practical usefulness of this approach, we have implemented the calculation of metrics for the *maintainability* characteristic of the test specification quality model and in particular its *analysability* and *changeability* subcharacteristics. Using the GQM approach, we chose the subsequently described metrics for the subcharacteristic assessment. They incorporate complexity metrics as described in Sect. 3.1 as well as metrics that are based on counting occurrences of the TTCN-3 code smell defined in Sect. 3.2.

To ease the subsequent definition of prescriptive metrics, we define the violation of an upper bound as follows:

Definition 1 (Upper bound violation) A measurement m that exceeds a threshold value u, violates the upper bound of a metric. The corresponding function *bound* is defined as follows:

$$bound(m, u) := \begin{cases} 0 & \text{if } m < u \\ 1 & \text{if } m \geq u \end{cases}$$

4.1 Analysability metrics

The metric listed in this section concerns the degree to which a test specification can be diagnosed for deficiencies, e.g. a badly structured test suite affects the difficulty of code reviews.

Metric 3 (Complexity violation ratio) The complexity violation ratio CVR represents the ratio between the number of TTCN-3 test cases, functions, and altsteps that exceed a chosen boundary value u of a complexity measure and the overall number n of behavioural entities (test cases, functions, and altsteps). Let *elem* be the sequence of $|elem|$ test cases, functions, and altsteps and let $elem(i)$, $i \in \mathbb{N}$, denote the ith element in *elem*. The function $cm(e)$ denotes the complexity measure of an element e from elem. Then, the complexity

violation ratio CVR is defined as follows:

$$CVR := 1 - \frac{\sum_{i=1}^{|elem|} bound(cm(elem(i)), u)}{|elem|}$$

Since various aspects contribute to the complexity of a test behaviour, several complexity measures $cm(e)$ may be used, e.g. McCabe's cyclomatic complexity [29,55], nesting level [18], or call-depth.

4.2 Changeability metrics

Changeability is the capability of a test specification to enable the implementation of necessary changes. For example, badly structured code or a non-expandable test architecture have a negative impact on this subcharacteristic.

Metric 4 (Code duplication ratio) The code duplication ratio CDR describes how much of the code is duplicated. Let *entities* be a sequence of entities that could possibly be duplicated (such as all branches of **alt** statements in a test suite) and let $entities(i)$, $i \in \mathbb{N}$, denote the ith element in *entities*. With $|entities|$ being the total number of elements in the sequence, the code duplication ratio CDR is then defined as:

$$CDR := 1 - \frac{\sum_{i=1}^{|entities|} dup(entities, i)}{|entities|}$$

where the function $dup(entities, i)$ with $i \in \mathbb{N}$ being a position within the sequence is defined as:

$$dup(entities, i) := \begin{cases} 0 & \text{if } \nexists j \in \mathbb{N} : \\ & entities(j) = entities(i), \quad j < i \\ 1 & \text{otherwise} \end{cases}$$

That means, $dup(entities, i)$ yields 1, if $entities(i)$ is duplicated, and 0 if not.

Metric 5 (Reference count violation ratio) When applying changes to entities which are referenced very often, a test developer needs to check for every reference whether a change may have unwanted side effects or requires follow-up changes. The reference count violation ratio RCVR determines how often an entity is referenced and penalises the violation of a given upper boundary value u. With *defs* being a sequence indexing distinct entity definitions, the reference count violation ratio RCVR is defined as follows:

$$RCVR := 1 - \frac{\sum_{i=1}^{|defs|} bound(ref(defs, i), u)}{|defs|}$$

where the function $ref(defs, i)$, $i \in \mathbb{N}$, denotes the number of references to the ith distinct entity definition:

$$ref(defs, i) := \text{number of references to } defs(i)$$

Metric 6 (Magic value count violation) Magic numbers or magic strings, i.e. literals that are not defined as constant,

decrease *changeability* when used excessively. The magic value count violation metric MVCV is designed to indicate bad quality when the total number of magic values exceeds a given threshold value u. Thus, the result of this metric is Boolean. With m being the total number of existing magic values in a test suite, the magic value count violation MVCV of a test suite is defined as:

$$MVCV := bound(m, u)$$

The metrics 3–6 have been designed to yield a value between 0 (the considered quality aspect is not fulfiled at all) and 1 (the considered quality aspect is fulfilled to 100%). The presented metrics serve as example for the application and do not represent a complete means to measure these characteristics.

4.3 Results

The described metrics for assessing the *maintainability* quality characteristic have been implemented in our TRex tool (see Sect. 5). We applied them to three different TTCN-3 test suites: the SIP test suite [13], the HiperMAN test suite [14], and the IPv6 test suite [15] which are standardised by the *European Telecommunications Standards Institute* (ETSI) and publicly available. The calculated metric values for these test suites are shown in Table 1. In the first third of the table, the number of lines of code and number of test cases are provided to give an impression of the size of the three test suites. The next part of the table contains the values that are used as input to calculate the metrics CVR, CDR, RCVR, and MVCV. The results of these calculations are shown in the lower third of the table.

To obtain a value for the complexity violation ratio CVR, the number of exceeded complexity thresholds and the number of behavioural entities are used. As complexity measure cm, we have chosen the cyclomatic complexity $v(G)$ from McCabe. As discussed before (Sect. 3.1.2), a threshold value $u := 10$ is reasonable. The code duplication ratio CDR is determined with respect to the number of duplicated branches in **alt** statements and the number of total **alt** branches. For the calculation of the reference count violation ratio RCVR, the number of exceeded reference thresholds and the total number of distinct entity definitions are required. As threshold for exceeded referencing, we selected $u := 50$. For deciding whether a magic value count violation occurred, the magic value count is required. As threshold for the magic value count violation MVCV, we used $u := 100$. The latter two threshold values are based on our intuitive perception of how much work a single change should cause. For example, changing a definition requires examining the impact to all its references which may be scattered all over the test suite. Thus, changing a definition manually may become a really costly task when it is referenced too often.

```
1  [] SIPP.receive (INVITE_Request_r_1) {
2          repeat;
3  }
4  [] SIPP.receive {
5          all timer.stop;
6          setverdict (fail);
7          rejectInvite(v_CSeq);
8          stop;
9  }
10 [] TAck.timeout {
11         setverdict (fail);
12         rejectInvite(v_CSeq);
13         stop;
14 }
```

Listing 2 Duplicated alt branches in the SIP test suite

```
1  v_ptc1.start (
2          ptc1_FinalAnswer(
3                  302,
4                  "Moved Temporarily",
5                  loc_CSeq_ptc_s,
6                  v_CallId));
7  v_ptc2.start (
8          ptc2_FinalAnswer(
9                  302,
10                 "Moved Temporarily",
11                 loc_CSeq_ptc2_s,
12                 v_CallId));
```

Listing 3 Magic values in the SIP test suite

The measurements of the ETSI test suites show that the CVR and RCVR metrics yield values which are within reasonable boundaries. However, the CDR measurement of the SIP test suite indicates that a lot of duplicated branches of **alt** statements could be avoided by using altsteps. Listing 2 shows such a generic code fragment (taken from the file SIP_CallControl.ttcn of the SIP test suite) that is repeated multiple times.

Not only does the repetition of such a code fragment increase the size of the overall test suite and reduce its *analysability*, but it also decreases *changeability* as the duplicate fragments must be found and adjusted as well.

Furthermore, the values of the MVCV metric indicate that all test suites make little to no use of constant definitions and constant references for avoiding magic values. Listing 3 (taken from SIP_CallControl.ttcn as well) demonstrates this problem in two subsequent statements: in lines 3 and 9, the magic number 302 is used. Without knowing the details of the SIP protocol, it is hard understand what this number means. Hence, using a constant with a descriptive name would improve the *analysability* of the test suite. Furthermore, the *changeability* of a test suite is increased by using constants instead of magic values.

The exemplary investigation of these results demonstrates that the used metrics are not only descriptive (and thus able to quantify occurrences of TTCN-3 language constructs), but also prescriptive: they help to locate concrete problems and

Table 1 Maintainability metrics of ETSI test suites

Metric	SIP v4.1.1	HiperMAN v2.3.1	IPv6 v1.1
Lines of code	61282	54565	41801
Number of test cases	609	364	286
Number of exceeded complexity thresholds ($cm := v(G)$, $u := 10$)	31	35	9
Number of behavioural entities $\|elem\|$	1478	701	881
Number of duplicate alt branches	938	169	224
Number of total alt branches $\|entities\|$	1535	1034	560
Number of exceeded reference thresholds ($u := 50$)	71	54	49
Number of distinct entity definitions $\|defs\|$	2077	2460	1759
Magic value count m	2214	545	618
Complexity violation ratio CVR ($cm := v(G)$, $u := 10$)	0.02	0.05	0.01
Code duplication ratio CDR (with respect to alt branches)	0.61	0.16	0.40
Reference count violation ratio RCVR ($u := 50$)	0.03	0.02	0.03
Magic value count violation MVCV ($u := 100$)	1.00	1.00	1.00

to make quantified statements on quality aspects of TTCN-3 test suites by combining descriptive metrics with goal information, e.g. threshold values or ratios.

5 The TRex tool

To practically evaluate our method for the assessment and improvement of TTCN-3 test specification quality, we have implemented the TTCN-3 Refactoring and Metrics tool *TRex* [48]. The initial version has been developed in collaboration with Motorola Labs, UK [1]. TRex currently implements assessment and improvement techniques for TTCN-3 based on static analysis and TTCN-3 source code restructuring. More precisely, the tool realises the calculation of internal metrics, automated smell detection, and refactoring as well as *Integrated Development Environment* (IDE) functionality for TTCN-3. The latter is provided by a *TTCN-3 perspective* (Fig. 3) which includes typical state-of-the-art functionality:

- a *navigator view* for project browsing,
- an *editor* with syntax highlighting and syntax checking according to the specification of the textual TTCN-3 core language (v3.1.1),
- a *code formatter*,
- an *outline view* providing a tree representation of the structure of the currently edited file,
- *content assist* which automatically completes identifiers from their prefix and scope.

Furthermore, it is possible to invoke external TTCN-3 compilers from within TRex.

5.1 TTCN-3 metrics functionality

TRex implements a considerable amount of size metrics (such as number of statements or the number of references to definitions) and structural metrics (such as the cyclomatic complexity) including those metrics mentioned in Sect. 4. Some structural metrics require the creation of control flow graphs and call graphs for each TTCN-3 behavioural entity. For a manual inspection, these graphs can be visualised as shown in Fig. 4.

The calculated metrics are displayed in the *metrics view* (Fig. 5a). The metrics are hierarchically organised as a tree which supports different aggregation types (e.g. sum or mean). Hence, it is possible to investigate the calculated metrics at different scopes.

5.2 TTCN-3 code smell detection functionality

A total number of 11 TTCN-3 code smell detection rules have been implemented in TRex. By means of static analysis, TRex is able to find

- *Activation asymmetry* smells,
- template parameter instances of *Constant actual template parameter value*,
- *Duplicate alt branches*,
- *Fully-parametrised template* smells,
- *Magic values* of numeric or string types,
- instances of *Short template* with configurable character lengths,
- instances of *Singular component variable/constant/ timer reference*,
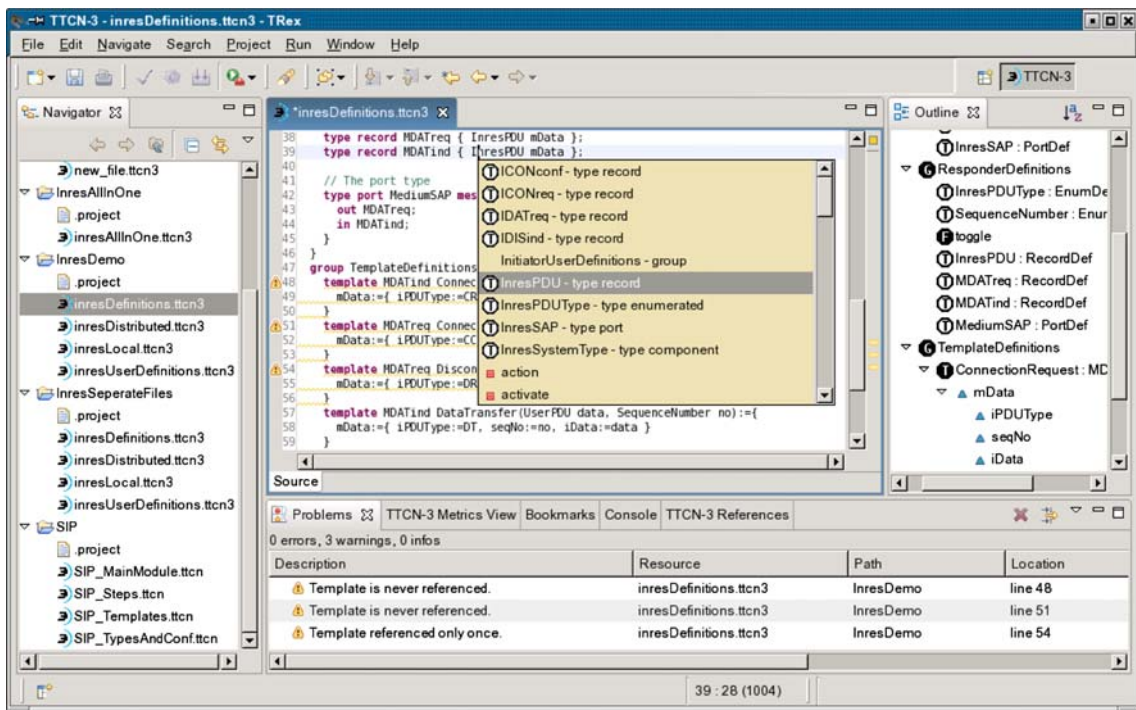- instances of *Singular template reference*,
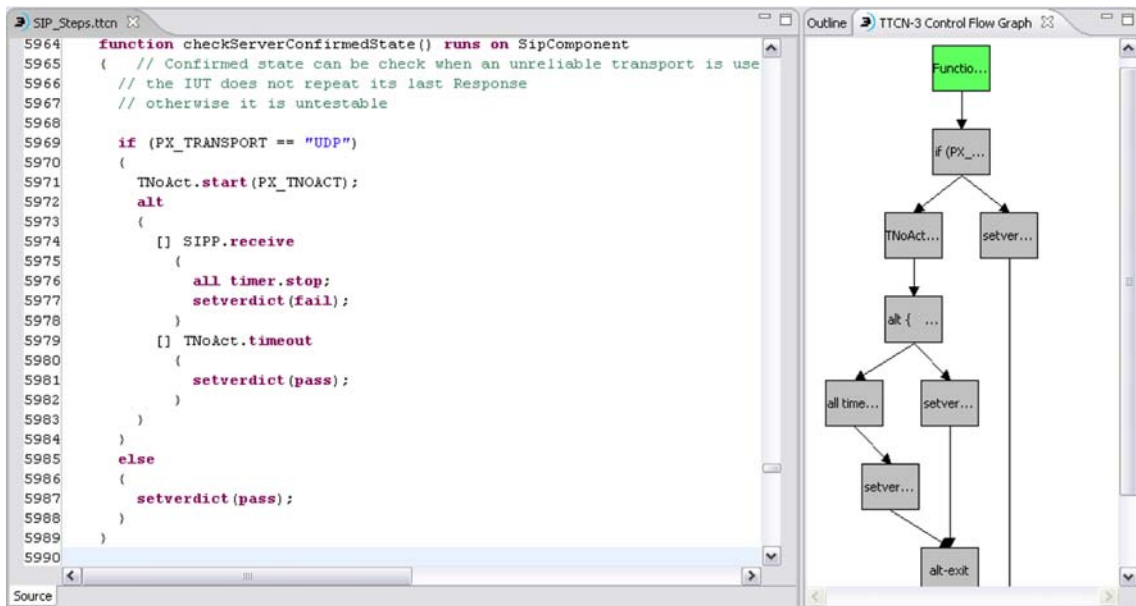
Fig. 3 TRex TTCN-3 perspective

Fig. 4 TRex control flow graph view

- template definitions with *Duplicate template fields*,
- instances of any local *Unused definition*, and
- occurrences of an *Unused definition* of a global template instance.

As stated in Sect. 3, whether a certain code structure is considered as a smell or not, may vary from project to project.

Therefore, TRex supports enabling and disabling individual TTCN-3 code smell detection rules and to store and retrieve these preferences as customised analysis configurations.

The results of the smell detection are displayed in the *Analysis results* view (Fig. 5b). The listed results are organised within a tree. Clicking the entries results in a jump to the corresponding position in the TTCN-3 source code displayed

(a) Metrics view

(b) Smell analysis results view
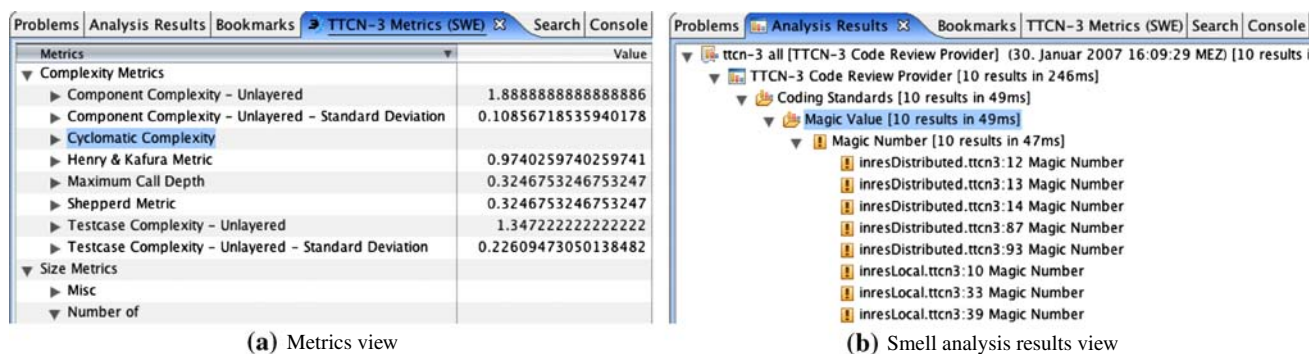
Fig. 5 TRex metrics and smell analysis views

in the editor window. Some rules, for example *Unused definitions*, offer the possibility for invoking so called *Quick Fixes*. Quick Fixes automatically suggest a TTCN-3 source code change to remove a detected smell. In fact, these Quick Fixes invoke refactorings.

5.3 TTCN-3 refactoring functionality

To improve the quality TTCN-3 test suites, TRex supports refactoring of TTCN-3 test suites [1,54]. Refactoring is based on a systematic behaviour preserving restructuring of TTCN-3 source code. So far, we have identified 51 refactorings that are suitable for TTCN-3. We have collected them in our TTCN-3 refactoring catalogue [48,53,54].

The refactoring mechanics, step-by-step instructions of how to apply each refactoring, are used by TRex to automate the application of a refactoring. The TTCN-3 refactorings which are currently implemented in TRex emphasise on improving template definitions. So far, the following refactoring are realised:

- the *Inline Template* refactoring inlines a template,
- the *Extract Template* refactoring turns one or several identical in-line templates into a template definition,
- the *Inline Template Parameter* refactoring inlines a template parameter when all its references use a common actual parameter value,
- the *Merge Template* refactoring replaces several similar or even identical template definitions by a single, possibly parametrised, template definition,
- the *Decompose Template* refactoring decomposes complex templates by referencing to smaller templates,
- the *Replace Template with Modified Template* refactoring simplifies templates that differ only in few field values by using modified templates,
- the *Move Module Constants to Component* refactoring moves constants that are defined on module level into a component definition, if the constants are used inside a

single component only or only by behaviour running on the same component,
- the *Rename* refactoring changes a definition name when the name does not reveal its purpose.

There are two ways to apply refactorings in TRex. The first possibility is that the test engineer specifies the source code locations which are subject to the refactoring. Then, before a refactoring is applied, a refactoring wizard displays a preview page where the transformed and original TTCN-3 source code can be compared side by side. The second possibility is the automated application of refactorings through Quick Fixes. Since each of our TTCN-3 code smells presented in Sect. 3.2.2 has a *related actions* section, it is possible to implement and associate a suitable refactoring to remove the issue. As pointed out at the end of Sect. 5.2, TRex is able to automatically invoke the suitable refactoring to remove a smell by means of a Quick Fix. This way, a manual selection of the source code location to be refactored becomes unnecessary.

The automated assessment and improvement functionality of TRex yields test suites with increased internal quality [54]. As an experiment, we investigated the effect which TRex was able to achieve when being applied to the SIP, the HiperMAN, and the IPv6 ETSI test suites by performing refactorings for removing unused template definitions, replacing template definitions which are only referenced once by in-line templates, and by refactorings for merging similar templates: the reduction of the number of template definitions was between 11% (SIP) and 53% (IPv6). The resulting reduction of the test suite size in terms of lines of code depends on the extent to which template definitions contribute to the overall size of the test suites: the obtained reduction was between 1% (SIP) and 12% (IPv6).

5.4 Implementation

TRex is written in Java and is implemented as a set of plugins for the Eclipse Platform [9]. The Eclipse Platform is well

Fig. 6 The TRex toolchain
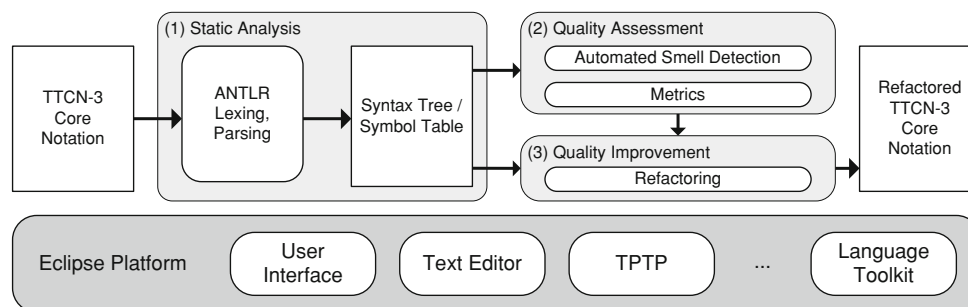
documented and supported. It provides many ready-to-use components such as project and file management, or a flexible graphical user interface making it easy to implement an IDE for new languages. Furthermore, the Eclipse Platform supports a very flexible plug-in architecture using the concept of *extension points*. Through the definition of TRex-specific extension points, it is very easy to add third-party extensions to TRex. Figure 6 illustrates how TRex is built on top of existing Eclipse components and infrastructure and how the different building blocks of TRex interact.

All language-oriented features of TRex are based on a lexer and parser generated using "*ANother Tool for Language Recognition*" (ANTLR) [40]. ANTLR also supports the traversal of *abstract syntax tree*s (ASTs) using tree grammars which are syntactically similar to the lexer and parser grammars. As shown in Block 1 of Fig. 6, additionally a symbol table is used for storing and retrieving information like the type of an identifier. Using the syntax tree resulting from the parsing and the created symbol table, it is possible to realise the static analysis techniques required for the quality assessment (Block 2 of Fig. 6) and quality improvement (Block 3 of Fig. 6) implemented in TRex.

The metrics implementation in TRex offers its own extensible infrastructure. A new metric calculation can be implemented easily by adding a new plug-in which makes use of a special TRex metric extension point. The existing TRex TTCN-3 metric plug-ins calculate metrics using tree grammars which have been been enriched with semantic actions. These actions count, for example, the occurrences of certain language elements, or build and use control flow and call graphs of TTCN-3 entities. To visualise these graphs, the Eclipse *Graphical Editing Framework* (GEF) [10] has been utilised.

The smell detection in TRex uses the static analysis framework offered by the *Eclipse Test & Performance Tools Platform* (TPTP) [11]. In the context of this framework, each smell detection capability is represented by a selectable rule. TPTP provides the underlying programming interface for these rules as well as configuration dialogues to allow custom rule selection and analysis profiles and a view for the result output. Thus, it is possible to concentrate on the actual rule implementation and analysis aspects. The actual smell

detections are based on the syntax tree traversals and symbol table lookups.

The refactoring implementation (Block 3 of Fig. 6) makes use of the *Language Toolkit* (LTK) which is part of the Eclipse Platform. It provides a programming interface for transformations of an Eclipse workspace and thus provides abstract classes and wizards that need to be implemented. The benefit is that certain functionality is already provided, for example, a preview page in the refactoring wizard which displays the differences between the original and refactored code side by side. Based on the syntax tree and symbol table, the necessary changes for the workspace transformation are calculated and applied to the original TTCN-3 source code using a programmatic text editor which is provided by the Eclipse Platform as well. It supports all typical text operations such as copy, paste, move, or delete. The overall formatting of the original TTCN-3 source code is preserved since only the textually changed parts are modified. In some cases, the refactoring implementations make use of the TRex code formatter for the necessary changes to obtain valid TTCN-3 core notation from a transformed syntax tree.

5.5 Related work

Most recent work which realises quality assessment and improvement for source code uses Java as target language. Hence, test-specific quality aspects, e.g. those related to test verdicts or to the determinism of tests, are not considered in these tools. Software that calculates metrics of programming languages like C or Java has been around for decades. A few recent examples of such tools which target the quality assessment of programming languages are the Metrics plug-in for Eclipse [44], CodePro AnalytiX from Instantiations [24], or Telelogic Logiscope [46].

Approaches to automatic detection of issues in source code which are detectable using static analysis and go beyond metrics exist for a long time as well. While, e.g. the Lint tool [28] is older than the notion of *smell*, it detects issues which are nowadays considered as code smell.

Fowler suggests not to automate the localisation of code smells, but rather argues that "*no set of metrics rivals informed human intuition*" [20]. We think this statement is

correct in the sense that it seems impossible to find universally valid threshold values for metrics. After all, reasonable threshold values may differ significantly depending on the preferred coding style, the language used, and many other factors. Nevertheless, we think that it is possible to ease the detection of problems with proper tool support when threshold values are user definable and selected sensibly respecting such factors. To some extent, the selection of threshold values represents this intuitive aspect mentioned by Fowler. Related research in this area proves that automatic smell detection is possible and helpful [32,33,43]. In addition to this research, there are already reasonably mature tools for Java like Find-Bugs [41] or PMD [8] that have similar objectives and are able to detect code smells. However, these tools do neither consider TTCN-3 nor at least more general test specific properties at all.

Refactoring for C++ [38] and Smalltalk [42] has been known for some time, but has actually become popular only more recently after the publication of Fowler's book on refactoring [20] and by the Java refactoring functionality provided in the JetBrains IntelliJ IDEA [27] and the Eclipse JDT [9] IDEs. However, an automated assessment and improvement as provided by TRex is not a default feature of those IDEs. The TRex implementation represents the first publicly available refactoring and metrics tool for TTCN-3 test specifications. The TTCN-3 IDE TTworkbench [47] has recently been extended to provide initial support for refactoring as well.

6 Summary and outlook

In this article, we have presented a procedure and means to assess and improve the quality of TTCN-3 test suites. The approach is based on a quality model for test specifications, which is an adaptation of the ISO/IEC 9126 quality model to the domain of test specification. The quality model addresses the different aspects related to quality by defining several characteristics. The concrete assessment of a quality characteristic is based on metrics. For each characteristic under investigation an appropriate set of metrics has to be selected and applied.

We have instantiated the quality model for the assessment of TTCN-3 test specifications by providing and discussing metrics. These metrics can be used to measure the quality of TTCN-3 specifications and they help to detect quality issues. In addition to metrics, we use code smells which have been collected and published by us in a TTCN-3 code smell catalogue. It is a first attempt at writing down problems related to TTCN-3 source code and we have the intention to allow others to contribute by making it publicly available as a wiki web page [48] and to extend it continuously. Our means to improve the quality of test specifications is refactoring, i.e.

we remove quality issues by restructuring a test specification without changing its behaviour.

Metrics, smell detection, and refactoring for TTCN-3 have been implemented in our TTCN-3 Refactoring and Metrics tool TRex. TRex is available as open-source tool from the project website [48]. We have shown the application of TRex to several standardised TTCN-3 test suites. In this study, we have investigated their maintainability subcharacteristics.

Currently, we are investigating further TTCN-3 code smells, e.g. those that relate to the usage of timers or functions in the guards of **alt** statements. In addition, we are refining the smell-based approach to detect issues in a test specification. At the moment, the smell detections in TRex are implemented using Java. Instead of this hard-coded imperative approach, we are currently developing a declarative method for specifying TTCN-3 code smell patterns using XQuery [50] expressions [36].

Our intention is to implement further metrics and support the quality assessment based on user-specific variants of our quality model. The latter will allow to select appropriate metrics and thresholds for each (sub)characteristics and to give weights to the different (sub)characteristics to obtain a general quality verdict. Furthermore, we have started to investigate means to evaluate whether chosen metrics are reasonable and independent, i.e. orthogonal to each other [52].

Our future work concentrates on the refinement, completion, and implementation of our quality model. At the moment, our model covers only internal quality characteristics. We will start to investigate a generalisation of our model which also includes external quality characteristics, e.g. performance aspects and properties related to test campaigns.

Finally, we intend to instantiate our quality model for tests specified with the *UML 2.0 Testing Profile* (U2TP) [37]. Thus, aspects of graphical design and object orientation will be future challenges of our work on quality assessment and quality improvement for test specifications.

References

1. Baker, P., Evans, D., Grabowski, J., Neukirchen, H., Zeiss, B.: TRex—the refactoring and metrics tool for TTCN-3 test specifications. In: Proceedings of TAIC PART 2006 (Testing: Academic & Industrial Conference—Practice And Research Techniques), Cumberland Lodge, Windsor Great Park, UK, 29–31 August 2006. IEEE Computer Society (2006). doi:10.1109/TAIC-PART.2006.35

2. Baker, P., Loh, S., Weil, F.: Model-driven engineering in a large industrial context—Motorola case study. In: Briand, L., Williams, C. (eds.) Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2–7, 2005. Lecture Notes in Computer Science (LNCS), vol. 3713, pp. 476–491. Springer, Berlin (2005). doi:10.1007/11557432_36

3. Basili, V.R., Weiss, D.M.: A methodology for collecting valid software engineering data. IEEE Trans. Softw. Eng. **10**(6), 728–738 (1984)

4. Bisanz, M.: Pattern-based smell detection in TTCN-3 test suites. Master's thesis, Center for Computational Sciences, University of Göttingen, Germany, ZFI-BM-2006-44 (2006). http://www.swe.informatik.uni-goettingen.de/publications/MB/bisanz_mastersthesis.pdf

5. Boehm, B., Brown, J., Kaspar, J., Lipow, M., MacLead, C., Merrit, M.: Characteristics of Software Quality. North-Holland, Amsterdam (1978)

6. DeMillo, R.A., Lipton, R.J., Sayward, F.G.: Hints on test data selection: help for the practicing programmer. IEEE Comp. **11**(4), 34–43 (1978). doi:10.1109/C-M.1978.218136

7. van Deursen, A., Moonen, L., van den Bergh, A., Kok, G.: Extreme Programming Perspectives, Chap. Refactoring Test Code, pp. 141–152. Addison-Wesley, Boston (2002)

8. Dixon-Peugh, D.: PMD (2007). http://pmd.sourceforge.net

9. Eclipse Foundation: Eclipse (2007). http://www.eclipse.org

10. Eclipse Foundation: Eclipse Graphical Editing Framework (2007). http://www.eclipse.org/gef

11. Eclipse Foundation: Eclipse Test & Performance Tools Platform Project (TPTP) (2007). http://www.eclipse.org/tptp

12. van Emden, E., Moonen, L.: Java quality assurance by detecting code smells. In: Proceedings Ninth Working Conference on Reverse Engineering WCRE 2002, pp. 97–106. IEEE Computer Society Press, New York (2002). doi:10.1109/WCRE.2002.1173068

13. ETSI: Technical Specification (TS) 102 027-3 V4.1.1 (2006–07): SIP ATS & PIXIT; Part 3: Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2006)

14. ETSI: Technical Specification (TS) 102 385-3 V2.2.1 (2006–04): Conformance Testing for WiMAX/HiperMAN 1.2.1; Part 3: Abstract Test Suite (ATS). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2006)

15. ETSI: Technical Specification (TS) 102 516 V1.1 (2006–04): IPv6 Core Protocol; Conformance Abstract Test Suite (ATS) and partial Protocol Implementation eXtra Information for Testing (PIXIT). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2006)

16. ETSI: ETSI Standard (ES) 201 873 V3.2.1: The Testing and Test Control Notation version 3; Parts 1–8. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation series. Z.140 (2007)

17. Fan, C.F., Yih, S.: Prescriptive metrics for software quality assurance. In: Proceedings of the First Asia-Pacific Software Engineering Conference, pp. 430–438. IEEE-CS Press, Tokyo (1994). doi:10.1109/APSEC.1994.465237

18. Fenton, N.E., Pfleeger, S.L.: Software Metrics. PWS Publishing Company, Boston (1997)

19. Fosdick, L.D., Osterweil, L.J.: Data flow analysis in software reliability. ACM Comput. Surv. **8**(3), 305–330 (1976). doi:10.1145/356674.356676

20. Fowler, M.: Refactoring—Improving the Design of Existing Code. Addison-Wesley, Boston (1999)

21. Gamma, E., Beck, K.: JUnit (2007). http://junit.sourceforge.net

22. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An introduction to the testing and test control notation (TTCN-3). Comp. Netw. **42**(3), 375–403 (2003). doi:10.1016/S1389-1286(03)00249-4

23. Huang, J.C.: Detection of data flow anomaly through program instrumentation. IEEE Trans. Softw. Eng. **5**(3), 226–236 (1979). doi:10.1109/TSE.1979.234184

24. Instantiations: CodePro AnalytiX (2007). http://www.instantiations.com/codepro/

25. ISO/IEC: ISO/IEC Standard No. 14598: Information technology—Software product evaluation; Parts 1–6. International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC), Geneva, Switzerland (1999–2001)

26. ISO/IEC: ISO/IEC Standard No. 9126: Software engineering—Product quality; Parts 1–4. International Organization for Standardization (ISO)/International Electrotechnical Commission (IEC), Geneva, Switzerland (2001–2004)

27. JetBrains: IntelliJ IDEA (2007). http://www.jetbrains.com

28. Johnson, S.: Lint, a C Program Checker. Unix Programmer's Manual. AT&T Bell Laboratories, New Jersey (1978)

29. McCabe, T.J.: A complexity measure. IEEE Trans. Softw. Eng. **2**(4), 308–320 (1976)

30. McCall, J., Richards, P., Walters, G.: Factors in software quality. Technical Report RADC TR-77-369, US Rome Air Development Center (1977)

31. Meszaros, G.: XUnit Test Patterns. Addison-Wesley, Boston (2007)

32. Moha, N., Guéhéneuc, Y.G., Leduc, P.: Automatic generation of detection algorithms for design defects. In: 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006), 18–22 September 2006, Tokyo, Japan, pp. 297–300. IEEE Computer Society, New york (2006). doi:10.1109/ASE.2006.22

33. Munro, M.: Product metrics for automatic identification of "Bad Smell" design problems in Java source-code. In: 11th IEEE International Symposium on Software Metrics (METRICS 2005), 19–22 September 2005, Como, Italy. IEEE Computer Society, New York (2005). doi:10.1109/METRICS.2005.38

34. Myers, G.: The Art of Software Testing. Wiley, New York (1979)

35. Neukirchen, H., Bisanz, M.: Utilising code smells to detect quality problems in TTCN-3 test suites. In: Petrenko, A., Veanes, M., Tretmans, J., Grieskamp, W. (eds.) Testing of Communicating Systems/Formal Approaches to Testing of Software 2007, Tallinn, Estonia, June 26–29, 2007. Lecture Notes in Computer Science (LNCS), vol. 4581, pp. 228–243. Springer, Berlin (2007). doi:10.1007/978-3-540-73066-8_16

36. Nödler, J.: An XML-based approach for software analysis—applied to detect bad smells in TTCN-3 test suites. Master's thesis, Center for Computational Sciences, University of Göttingen, Germany, ZFI-BM-2007-36 (2007). http://www.swe.informatik.uni-goettingen.de/publications/JN/noedler-masters-thesis.pdf

37. OMG: UML Testing Profile (Version 1.0 formal/05-07-07). Object Management Group (OMG) (2005)

38. Opdyke, W.: Refactoring object-oriented frameworks. Ph.D. thesis, University of Illinois at Urbana-Champaign, USA (1992)

39. Parnas, D.: Software aging. In: Proceedings of the 16th International Conference on Software Engineering (ICSE), May 16–21, 1994, Sorrento, Italy, pp. 279–287. IEEE Computer Society/ACM Press, New York (1994)

40. Parr, T.: ANTLR parser generator v2 (2007). http://www.antlr2.org

41. Pugh, B.: FindBugs (2007). http://findbugs.sourceforge.net

42. Roberts, D., Brant, J., Johnson, R.: A refactoring tool for smalltalk. theory and practice of object systems **3**(4), 253–263 (1997). doi:10.1002/(SICI)1096-9942(1997)3:4<253::AID-TAPO3>3.3.CO;2-I

43. van Rompaey, B., du Bois, B., Demeyer, S.: Characterizing the relative significance of a test smell. In: Proceedings of the 22nd IEEE International Conference on Software Maintenance (ICSM 2006), Philadelphia, Pennsylvania, September 25–27, 2006, pp. 391–400. IEEE Computer Society (2006). doi:10.1109/ICSM.2006.18

44. Sauer, F.: Eclipse Metrics Plugin (2007). http://metrics.sourceforge.net

45. Sneed, H.M.: Measuring the effectiveness of software testing. In: Beydeda, S., Gruhn, V., Mayer, J., Reussner, R., Schweiggert, F. (eds.) Proceedings of SOQUA 2004 (First International Workshop on Software Quality) and TECOS 2004 (Workshop Testing Component-Based Systems). Lecture Notes in Informatics (LNI), vol. 58. Gesellschaft für Informatik, Köllen Verlag, Bonn (2004)

46. Telelogic: Logiscope (2007). http://www.telelogic.de/products/logiscope/

47. Testing Technologies: TTworkbench (2007). http://www.testingtech.de/products_services/ttwb_intro.php

48. TRex Team: TRex Website (2007). http://www.trex.informatik.uni-goettingen.de

49. Vega, D.E., Schieferdecker, I.: Towards quality of TTCN-3 tests. In: Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling, May'31–June 2, 2006, University of Kaiserslautern, Germany. University of Kaiserslautern, Germany (2006)

50. XQuery 1.0: An XML Query Language. World Wide Web Consortium (W3C) Recommendation 23 January 2007 (2007)

51. Watson, A.H., McCabe, T.J.: Structured testing: a testing methodology using the cyclomatic complexity metricy. NIST Special Publication 500–235, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg (1996)

52. Werner, E., Grabowski, J., Neukirchen, H., Röttger, N., Waack, S., Zeiss, B.: TTCN-3 quality engineering: using learning techniques to evaluate metric sets. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007: Design for Dependable Systems, 13th International SDL Forum, Paris, France, September 18–21, 2007, Proceedings. Lecture Notes in Computer Science (LNCS), vol. 4745, pp. 54–68. Springer, Berlin (2007). doi:10.1007/978-3-540-74984-4_4

53. Zeiss, B.: A refactoring tool for TTCN-3. Master's thesis, Center for Computational Sciences, University of Göttingen, Germany, ZFI-BM-2006-05 (2006). http://www.swe.informatik.uni-goettingen.de/publications/BZ/zeiss_mastersthesis.pdf

54. Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P.: Refactoring and Metrics for TTCN-3 Test Suites. In: Gotzhein, R., Reed, R. (eds.) System Analysis and Modeling: Language Profiles. 5th International Workshop, SAM 2006, Kaiserslautern, Germany, May 31–June 2, 2006, Revised Selected Papers, Lecture Notes in Computer Science (LNCS), vol. 4320, pp. 148–165. Springer, Berlin (2006). doi:10.1007/11951148_10

55. Zeiss, B., Neukirchen, H., Grabowski, J., Evans, D., Baker, P.: 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, pp. 117–128. dpunkt.Verlag, Heidelberg (2006)

56. Zeiss, B., Vega, D., Schieferdecker, I., Neukirchen, H., Grabowski, J.: Applying the ISO 9126 quality model to test specifications—exemplified for TTCN-3 test specifications. In: Bleeck, W.G., Rasch, J. Züllighoven, H. (eds.) Proceedings of Software Engineering 2007 (SE 2007). Lecture Notes in Informatics (LNI), vol. 105, pp. 231–242. Gesellschaft für Informatik, Köllen, Bonn (2007)