Quality assurance for TTCN-3 test specifications‡

Helmut Neukirchen¹, *, †, Benjamin Zeiss¹, Jens Grabowski¹, Paul Baker² and Dominic Evans²

¹Software Engineering for Distributed Systems Group, Institute for Computer Science, University of Göttingen, Germany
²Motorola Labs, Basingstoke, U.K.

SUMMARY

Comprehensive testing of modern communication systems often requires large and complex test suites, which have to be maintained throughout the system life cycle. Industrial experience, with those written using the standardized Testing and Test Control Notation (TTCN-3), has shown that this maintenance is a non-trivial task and its burden can be reduced by means of appropriate concepts and tool support. To this aim, Motorola has collaborated with the University of Göttingen to develop TRex, an open-source TTCN-3 development environment, which notably provides suitable metrics and refactorings to enable the assessment and automatic restructuring of test suites. This article presents concepts like metrics and refactoring for the quality assurance of TTCN-3 test suites and their implementation provided by the TRex tool. These means make it far easier to construct and maintain TTCN-3 tests that are concise and optimally balanced with respect to maintainability quality characteristics. Copyright © 2008 John Wiley & Sons, Ltd.

Received 5 January 2007; Revised 16 August 2007; Accepted 19 September 2007

KEY WORDS: testing; quality assurance; TTCN-3; refactoring; metrics; tool; eclipse

1. INTRODUCTION

The Testing and Test Control Notation (TTCN-3) [1] is a test specification and test implementation language standardized by the European Telecommunications Standards Institute (ETSI) and the

*Correspondence to: Helmut Neukirchen, Institute for Computer Science, University of Göttingen, Lotzestr. 16–18, 37083 Göttingen, Germany.
†E-mail: neukirchen@cs.uni-goettingen.de
‡This article is an extended version of the paper ‘TRex—The Refactoring and Metrics Tool for TTCN-3 Test Specifications’ by Paul Baker, Dominic Evans, Jens Grabowski, Helmut Neukirchen, and Benjamin Zeiss, which was originally presented at TAIC PART 2006 (Testing: Academic & Industrial Conference—Practice And Research Techniques).
While TTCN-3 has its roots in functional black-box testing of telecommunication systems, it is nowadays also used for testing in other domains such as Internet protocols, automotive, aerospace, or service-oriented architectures. TTCN-3 is applicable not only for specifying and implementing functional tests but also for scalability, robustness, or stress tests. Commercial TTCN-3 tools [2–7] and in-house solutions support editing test suites and compiling them into executable code.

Experience within Motorola has shown that not only editing and executing TTCN-3 test suites but also maintenance of TTCN-3 test suites is an important issue [8]. For example, the conversion of a legacy test suite for a UMTS-based component to TTCN-3 resulted in 60,000 lines of code, which were hard to read, (re-)use, and maintain. To address this issue, appropriate concepts and tool support are required.

Since no comprehensive approaches and tools for assessing and improving the quality of TTCN-3 test suites existed, Motorola has collaborated with the University of Göttingen to develop a quality assurance approach for TTCN-3 test suites. As a result, a refactoring and metrics tool, called TReX, has been created. The initial aims of TReX were to (1) enable the assessment of a TTCN-3 test suite with respect to lessons learnt from experience; (2) provide a means of detecting opportunities for avoiding issues; and (3) provide a means for restructuring TTCN-3 test suites to improve them with respect to any existing issue. To let others participate in this tool and to participate in contributions from others, TReX is available under the open source Eclipse Public License.

This article is structured as follows: As a foundation, an overview of TTCN-3 is provided in the next section. Then, in Section 3, metrics for TTCN-3 are discussed. Section 4 introduces refactoring for TTCN-3. Based on both metrics and refactoring, a rule-based automated issue detection and removal strategy is described in Section 5. In Section 6, a description of TReX’s functionality and its implementation is given. Section 7 presents results from applying TReX to publicly available test suites. Related work is discussed in Section 8. Finally, a summary and an outlook conclude this article.

2. AN OVERVIEW OF TTCN-3

The test specification and test implementation language TTCN-3 [1] has the look and feel of a typical general-purpose programming language, i.e. it is based on a textual syntax, referred to as the core notation. Most of the concepts of general-purpose programming languages can be found in TTCN-3 as well, e.g. data types, variables, functions, parameters, visibility scopes, loops, or conditional statements. In addition, test-related concepts are available to ease the specification of test suites. Listing 1 gives an impression of typical TTCN-3 concepts and how a TTCN-3 test suite looks like. In this article, the keywords of TTCN-3 are highlighted using bold face type. Single quotes are used in the text to refer to identifiers and values that can be found in TTCN-3 listings.

A test suite consists of one or more named modules (line 1 of Listing 1). Inside each module, types, values, and behaviour can be defined and a module may import definitions from other modules.
module exampleModule {

type record ExampleType {
    boolean ipv6,
    charstring ipAddress
}

type port ExamplePortType message {
    inout ExampleType
}

type component ExampleComponentType {
    port ExamplePortType examplePort
}

template ExampleType exampleTemplate := {
    ipv6 := false,
    ipAddress := "127.0.0.1"
}

altstep exampleAltStep() runs on ExampleComponentType {
    examplePort. receive (ExampleType:?){
        setverdict (fail);
    }
}

testcase exampleTestCase() runs on ExampleComponentType {
    examplePort.send(exampleTemplate);
    alt {
        examplePort. receive (ExampleType:{false, "127.0.0.2"}) {
            setverdict (pass);
        }
    }
}
}

Listing 1. A TTCN-3 example test suite.

Lines 2–5 show the definition of a record data type. Such data-type definitions can be used to specify the type of the messages that are exchanged between the System Under Test (SUT) and the components of a TTCN-3 test system.

The communication in TTCN-3 takes place via ports. The definition of a port type for message-based communication can be found in lines 7–9. Line 8 specifies that instances of this port type may receive and send messages of the previously defined type, ‘ExampleType’.

TTCN-3 allows the specification of distributed tests, and the entities of distribution are test components that can be used to execute test behaviour in parallel. In lines 11–13, a type for a test component is defined: every test behaviour that runs on a component of this type has access to an instance of a port with the name ‘examplePort’ of type ‘examplePortType’. In addition to port instances, a component may provide instances of other elements, e.g. constants, variables, or timers.

For specifying test data, templates are used in TTCN-3. For example, lines 15–18 declare a template ‘exampleTemplate’ that contains test data for messages of type ‘ExampleType’. Such a template can be used to specify data values to be sent via a port or to specify expected data values that are received via a port. Unlike usual constants, templates support wildcards that can be used to match a set of corresponding values on reception rather than just one concrete value.
The actual test behaviour is specified using the **testcase** construct. An example can be found in lines 26–34. The **runs on** in line 26 specifies that this test case is executed on a test component of type ‘ExampleComponentType’; thus, this test case may access all elements of this component type, i.e. the port ‘examplePort’. In line 27, a **send** operation is used to send test data via the port ‘examplePort’. The data to be sent are specified by the reference to the template ‘exampleTemplate’. Alternative test behaviour that describes how a test case reacts to different observations can be specified using the **alt** construct (lines 28–33). In the example, the first branch (lines 29–31) deals with the case that via port ‘examplePort’ a message of a certain type and value is received. TTCN-3 allows not only referencing of template declarations for the specification of test data but also the creation of templates on-the-fly using a so-called **inline template** notation: The inline template in line 29 specifies a message of type ‘ExampleType’ with values **false** and ‘127.0.0.2’. If a message that matches this inline template has been received, the body of this **alt** branch is executed; in this case, the test verdict **pass** is set (line 30). The second branch of this **alt** construct contains a reference to an **altstep** that contains further alternative branches that are added to this **alt** statement.

The **altstep** in lines 20–24 is used to match any message of type ‘ExampleType’ that is received on the ‘examplePort’. To achieve this, an inline template is used that contains a wildcard (‘?’) instead of concrete values for the fields of the message record. In the context of the **alt** construct in lines 28–33, this wildcard will not match the expected value **false** and ‘127.0.0.1’, since the branches of **alt** statements are evaluated from top to bottom.

Information on further concepts of TTCN-3, such as test configurations for distributed tests, inheritance of templates using the **modifies** keyword, groups that can be used to give a module more structure, the implicit activation of **altsteps** as defaults, or the control part that allows the order in which test cases are executed to be specified, can be found in an introductory article [9], in a textbook [10], on the official TTCN-3 website [11], and in the TTCN-3 standard [1].

The simple example provided in Listing 1 demonstrates some maintenance issues with respect to templates: the actual data that are specified by a template are easier to comprehend if the inline template notation is used, because for a template reference an additional step of looking up the template declaration would be required. On the other hand, inline templates lead to a higher coupling between test behaviour and test data: if a value of a record field in a template specification needs to be changed it may have to be changed in a number of inline templates, whereas when only references to a template are used it would be sufficient to change only one single template declaration. Related trade-offs occur with respect to template declarations that are parameterized: on the one hand, several non-parameterized template declarations may be replaced by a single parameterized template declaration; on the other hand, the corresponding actual parameter values need to be provided from within the test behaviour, which in turn increases the coupling between test behaviour and test data. Hence, neither using only inline templates nor using only references to template declarations is a sufficient approach to improve maintainability; instead, the different aspects of maintainability need to be balanced. Similar considerations are applicable for other language constructs of TTCN-3.

The remainder of this article discusses how to assess maintainability aspects of a TTCN-3 test suite using metrics, how to restructure a test suite using refactorings, and how to balance different maintainability aspects using a rule-based approach.
3. METRICS FOR TTCN-3

For assessing the overall quality of software, metrics can be used. According to Fenton et al. [12], software metrics 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; internal attributes are properties of a process, resource, or product on its own, i.e. separate from any interactions with its environment. Hence, to measure the external attributes of a product, execution of the product is required, whereas for measuring the internal attributes, static analysis is sufficient. Since this article treats quality characteristics such as maintainability of TTCN-3 test specifications, only internal product attributes are considered in the following.

Internal product metrics can be structured into size and structural metrics. Size metrics typically measure the properties of the number of usages of programming or specification language constructs. Structural metrics analyse the structure of a program or specification. The most popular examples are coupling metrics and complexity metrics based on control flow or call graphs.

To assess the overall quality of software, usually averages of metrics are applied. By additionally considering the metrics of individual language elements, it is possible to identify locations of inappropriate usage of programming or specification language constructs. For example, by counting the number of references to each definition, issues like unused definitions can be identified. However, some issues cannot be detected by simple metrics, but need a pattern-based approach. These kinds of issues are called bad smells or code smells [13]. Examples are duplicated code or parameterized definitions, which are always referenced using the same actual parameter values. First results of an issue detection approach that is based on locating patterns of TTCN-3 code smells look promising [14,15]. The remainder of this section focuses on quality assessment based on metrics.

To be able to assess the quality of TTCN-3 test suites, it must first be determined which constituents of quality are relevant. The problems at Motorola related to the maintainability quality characteristics, in particular the corresponding subcharacteristics analysability and changeability, were of interest. Changeability relates to how easily modifications can be implemented in a test specification, analysability refers to how easily a test specification can be diagnosed for deficiencies or for parts to be modified to be identified. For a good analysability, in particular the readability of a test specification must be high. To support a more sophisticated quality model for test specifications, a first proposal that follows the ISO/IEC standard 9126 [16] has been developed [17].

For assessing the quality of TTCN-3 test suites in terms of analysability and changeability and for locating issues, an initial set of appropriate TTCN-3 metrics has been developed [18]. To ensure that these metrics have a clear interpretation, their development was guided by the Goal Question Metric approach from Basili et al. [19]: First the goals to achieve were specified, e.g. Goal 1: ‘Improve changeability of TTCN-3 source code’ or Goal 2: ‘Improve analysability of TTCN-3 source code’. Then, for each goal a set of meaningful questions was derived that characterizes it, e.g. for Goal 1: ‘Are many changes to test behaviour required if values inside the test data change?’; for Goal 2: ‘Are unnecessary indirections used?’ and ‘Are there any unused definitions?’. Finally, one or more metrics were defined to gather quantitative data that provide answers to each question, e.g. coupling metrics are used to answer the question of Goal 1 and counting the number of references for answering the questions of Goal 2.
The resulting set of metrics not only uses well-known metrics for general-purpose programming languages but also defines new TTCN-3-specific metrics. As a first step, some basic size metrics and one coupling metric are used:

- **Number of lines of TTCN-3 source code** including blank lines and comments, i.e. physical lines of code [12].
- **Number of test cases**, including **Number of references to a test case**.
- **Number of functions**, including **Number of references to a function**.
- **Number of altsteps**, including **Number of references to an altstep**.
- **Number of port types**, including **Number of references to a port type**.
- **Number of component types**, including **Number of references to a component type**.
- **Number of data type definitions**, including **Number of references to a data-type definition**.
- **Number of templates**, including **Number of references to a template** and **Number of parameterized templates**.
- **Template coupling**, which is to be computed as follows:

\[
\text{Template coupling} := \sum_{i=1}^{n} \frac{\text{score}(\text{stmt}(i))}{n}
\]

where \( \text{stmt} \) is the sequence of behaviour statements referencing templates in a test suite, \( n \) is the number of statements in \( \text{stmt} \), and \( \text{stmt}(i) \) denotes the \( i \)th statement in \( \text{stmt} \). \( \text{score}(\text{stmt}(i)) \) is defined as follows:

\[
\text{score}(\text{stmt}(i)) :=
\begin{cases}
1 & \text{if } \text{stmt}(i) \text{ references a template without parameters,} \\
& \text{e.g. ‘examplePort.receive(exampleTemplate)’} \\
& \text{or uses wildcards only} \\
& \text{e.g. ‘examplePort.receive(ExampleType:?)' } \\
2 & \text{if } \text{stmt}(i) \text{ references a template with parameters,} \\
& \text{e.g. ‘examplePort.receive(exampleTemplate(1))’} \\
3 & \text{if } \text{stmt}(i) \text{ uses an inline template,} \\
& \text{e.g. ‘examplePort.receive(ExampleType:[true, 1])’}
\end{cases}
\]

**Template coupling** measures the dependence of test behaviour and test data in the form of TTCN-3 template definitions, i.e. 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 inline 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 inline templates (i.e. the template coupling score will be close to 3).

While most of these metrics mainly describe the overall quality of test suites (an example is the **Template coupling** metric), some of them can also be used to improve a test suite by identifying the
location of individual issues. The application of these metrics to improve a test suite is illustrated in Section 5.

In addition to the above size and coupling metrics, the authors of this article have also investigated the applicability of complexity metrics to TTCN-3 test suites and found out that, e.g. the well-known Cyclomatic complexity metric from McCabe [20] exhibits the same properties for TTCN-3 test suites as for general-purpose programming languages.

4. REFACTORING FOR TTCN-3

Refactoring is defined as ‘a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behavior’ [13]. This means refactoring is a remedy against software ageing [21]. While refactoring can be regarded as cleaning up the source code, it is more systematic and thus less error prone than arbitrary code clean-up, because each refactoring provides a checklist of small and simple transformation steps that are often automated by tools.

The essence of most refactorings is independent from a specific programming language. However, a number of refactorings make use of particular constructs of a programming language, or of a programming paradigm in general, and are thus only applicable to source code written in this language.

Examples for simple refactorings are: renaming a variable to give it a more meaningful name, encapsulating fields of a class by replacing direct field accesses by calls to corresponding getter and setter accessor methods, or extracting a group of statements and moving it into a separate function. More complex refactorings are often based on simpler refactorings. For example, converting a procedural design into an object-oriented design requires the conversion of record types into data classes, encapsulating the public fields of the data classes, and extracting and moving statements from procedures into methods of the data classes.

Even though refactoring has a long tradition in Smalltalk, the first detailed written work on refactoring was the PhD thesis of Opdyke [22], who treats refactoring of C++ source code. Refactoring has finally been popularized by Fowler and his book ‘Refactoring—Improving the Design of Existing Code’ [13], which contains a catalogue of 72 refactorings that are applicable to Java source code.

Opdyke [22] and Fowler [13] also address the problem of how to ensure that a refactoring does not change the observable behaviour of the modified software. While Opdyke assumes that an automated tool performs the actual refactoring by applying transformation steps that are proven to be behaviour preserving, Fowler suggests a manual approach in which the validation is realized by repeated execution of existing unit tests. Therefore, Fowler’s refactoring transformation steps contain instructions to run tests to validate that the observable behaviour has not been changed. The approach described in this article follows the suggestion of Opdyke by implementing the transformation steps of a refactoring in an automated tool.

In the following, a refactoring catalogue for TTCN-3 [18] is presented. The presentation of the refactorings is inspired by Fowler’s refactoring catalogue for Java [13]. Hence, the same fixed format for describing the refactorings is used: each refactoring being described by its name, a summary, a motivation, mechanics, and an example. The name of a refactoring is always written in italics. The mechanics section contains systematic checklist-like instructions of how to perform the refactoring. These step-by-step instructions will be used to automate the application of a
refactoring by implementing them in a tool as described in Section 6. In the mechanics section of each refactoring description, the term source is used to refer to the code that is addressed by a refactoring and thus usually removed or simplified, and the term target to refer to the code that is created as a result of a refactoring. The example section of each refactoring description illustrates the refactoring by showing TTCN-3 core notation excerpts before and after the refactoring is applied.

The TTCN-3 refactoring catalogue is divided into refactorings for test behaviour, refactorings for data descriptions, and refactorings that improve the overall structure of a test suite. This classification is used in Sections 4.1 and 4.2.

4.1. Language-independent refactorings applicable to TTCN-3

For the creation of the TTCN-3 refactoring catalogue, it has first been investigated which of the 72 refactorings from Fowler [13] are also relevant for TTCN-3. Even though these refactorings were intended to be used for Java source code, some of them are language independent or can be reinterpreted in such a way that they are applicable to TTCN-3 test specifications. For their reinterpretation, it is necessary to replace the notion of Java methods by TTCN-3 functions or testcases. While TTCN-3 is not an object-oriented language, some of the Java refactorings are nevertheless applicable if the notion of Java classes and fields is replaced by TTCN-3 component types and variables, constants, timers, and ports local to a component, respectively. As a result, 28 refactorings are applicable to TTCN-3 (Table I). Where necessary, the names of these refactorings have been changed to reflect their reinterpretation for TTCN-3. In this case, the original name used by Fowler is given in square brackets.

No refactorings that are solely suitable for data description can be obtained by reinterpreting Fowler’s refactorings, since data description relates mainly to the notion of TTCN-3 templates that do not exist in Java. However, some of Fowler’s refactorings such as Inline Method or Add and Remove Parameter are quite generic and may also be reinterpreted for TTCN-3 templates. Where the mechanics of these refactorings differ significantly when applied to templates, they have been considered as TTCN-3-specific refactorings and are thus described in the following section.

4.2. Language-specific refactorings for TTCN-3

In addition to the language-independent refactorings, restructuring of TTCN-3 test suites can be leveraged by considering language constructs that are specific to TTCN-3. Currently, these refactorings take advantage of TTCN-3 altsteps, templates, grouping, modules and importing from modules, test components, restricted subtypes, logging, and creating distributed test cases.

Until now, 23 TTCN-3-specific refactorings have been identified. The summaries of these refactorings are as follows:

Refactorings for test behaviour:

- Extract Altstep: One or more alternative branches of an alt statement occur several times in a test suite and are thus moved into an altstep on its own.
- Split Altstep: Altsteps that contain branches that are not closely related to each other are split to maximize reuse potential.
QUALITY ASSURANCE FOR TTCN-3 TEST SPECIFICATIONS

Table I. Fowler refactorings applicable to TTCN-3.

<table>
<thead>
<tr>
<th>Refactorings for test behaviour</th>
<th>Refactorings for improving the overall structure of a test suite</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Consolidate Conditional Expression</td>
<td>• Add Parameter</td>
</tr>
<tr>
<td>• Consolidate Duplicate Conditional Fragments</td>
<td>• Extract Extended Component</td>
</tr>
<tr>
<td>• Decompose Conditional</td>
<td>[Extract Subclass]</td>
</tr>
<tr>
<td>• Extract Function [Extract Method]</td>
<td>• Extract Parent Component</td>
</tr>
<tr>
<td>• Introduce Assertion</td>
<td>[Extract Superclass]</td>
</tr>
<tr>
<td>• Introduce Explaining Variable</td>
<td>• Introduce Local Port/Variable/Constant/Timer</td>
</tr>
<tr>
<td>• Inline Function [Inline Method]</td>
<td>[Introduce Local Extension]</td>
</tr>
<tr>
<td>• Inline Temp</td>
<td>• Introduce Record Type Parameter</td>
</tr>
<tr>
<td>• Remove Assignments to Parameters</td>
<td>[Introduce Parameter Object]</td>
</tr>
<tr>
<td>• Remove Control Flag</td>
<td>• Parameterize</td>
</tr>
<tr>
<td>• Replace Nested Conditional with Guard Clauses</td>
<td>Testcase/Function/Altstep</td>
</tr>
<tr>
<td>• Replace Temp with Query</td>
<td>[Parameterize Method]</td>
</tr>
<tr>
<td>• Separate Query From Modifier</td>
<td>• Pull Up Port/Variable/Constant/Timer</td>
</tr>
<tr>
<td>• Split Temporary Variable</td>
<td>[Pull Up Field]</td>
</tr>
<tr>
<td>• Substitute Algorithm</td>
<td>• Push Down Port/Variable/Constant/Timer</td>
</tr>
<tr>
<td></td>
<td>[Push Down Field]</td>
</tr>
<tr>
<td></td>
<td>• Replace Magic Number with Symbolic Constant</td>
</tr>
<tr>
<td>• Replace Altstep with Default: Altsteps that are referenced in more than one alt statement are removed from the alt statements and activated as default altsteps.</td>
<td></td>
</tr>
<tr>
<td>• Add Explanatory Log: Add a log statement to explain why a test case was aborted or a non-pass verdict was assigned.</td>
<td></td>
</tr>
<tr>
<td>• Distribute Test: Transform a non-concurrent test case into a distributed concurrent test case.</td>
<td></td>
</tr>
</tbody>
</table>

Refactorings for improving the overall structure of a test suite:

- **Extract Module/Move Declarations to Another Module:** Move parts of a module into a newly created module or into another existing module to improve structure and reusability.
- **Group Fragments:** Add additional structure to a module by putting code fragments into groups.
- **Restrict Imports:** Restrict import statements to obtain smaller inter-module interfaces and less processing load for TTCN-3 tools.
- **Prefix Imported Declarations**: Prefix imported declarations to avoid possible name clashes.
- **Parameterize Module**: Parameterize modules to specify environment-specific parameters at tool level.
- **Move Module Constant to Component**: A declaration of a constant at module level used exclusively in the context of a single component is moved into the component declaration.
- **Move Local Variable/Constant/Timer to Component**: A local variable, constant, or timer is moved to a component when used in different functions, testcases, or altsteps that run on the same component.
- **Move Component Variable/Constant/Timer to Local Scope**: A component variable, constant, or timer is moved to a local scope when only used in a single function, testcase, or altstep.
- **Generalize Runs On**: Relax **runs on** specification by using a more general component type.

**Refactorings for data descriptions:**

- **Inline Template**: A template that is used only once is inlined.
- **Extract Template**: Inlined templates that are used more than once are extracted into a template definition and referenced.
- **Replace Template with Modified Template**: Templates of structured or list type with similar content values that differ only by a few fields are simplified by using modified templates.
- **Merge Template**: Replace several template declarations of the same type which use different or even the same values for the same fields with one single template.
- **Parameterize Template**: Several templates of the same type, which merely use different field values, are replaced by a single parameterized template.
- **Remove Duplicate Template**: Replace several template declarations that have the same body with a single template.
- **Inline Template Parameter**: A formal parameter of a template that is always given the same actual value is inlined.
- **Decompose Template**: Complex template declarations are decomposed into smaller templates using references.
- **Subtype Basic Types**: Range-constrained subtypes are used instead of basic types in order to detect code flaws more easily.

The remainder of this article focuses on refactoring for data descriptions, since most of the maintenance problems at Motorola were related to the use of templates. To give an impression of how the entries of the TTCN-3 refactoring catalogue are presented, those three refactorings that are mainly used in Section 5 are described in detail. Please refer to the complete TTCN-3 refactoring catalogue [23,24] for a detailed description of all refactorings.

### 4.2.1. Refactoring: Inline Template

**Summary**: A template that is used only once is inlined.

**Motivation**: In some cases, a template declaration is referenced only once throughout the whole test suite. Using the inline notation instead of a reference for this template may improve readability as the test engineer does not have to search through the test suite to find the corresponding declaration.
and it may furthermore shorten the code length. Candidates for this refactoring are simple templates since readability is typically only improved as long as the inlined template can be written in a single line.

**Mechanics:**

- Identify the source template declaration. It should be used only once throughout the code and should be simple. Otherwise, the use of this refactoring is not appropriate.
- Find the code location where the source template declaration is referenced. This is the source reference.
- Replace the source reference with the value list inline notation of the source template:
  - When a normal template is inlined, its notation has the standard notation of an inline template.
  - When a modified template is inlined, its notation must be adjusted to reflect the notation of modified inline templates.
  - When a parameterized template is inlined, the actual parameter values of the reference must be inlined into this value list notation.
- Remove the source template declaration. If the source template declaration is located in a different module than the source reference, the corresponding import statement in the module of the source reference should be adjusted to exclude the source template declaration.

**Example:** Listing 2 shows an excerpt of an unrefactored example. It contains a declaration of the record type ‘ExampleType’ (lines 1–4) and the source template declaration ‘exampleTemplate’ (lines 6–9). In the test case (lines 11–13), a message is sent by using a reference to the source template declaration ‘exampleTemplate’ (line 12). As this is the only reference to the template, the **Inline Template** refactoring is applicable and this template reference is the source reference that will be replaced using the inline template notation.

Listing 3 demonstrates the result of applying **Inline Template**. The source template declaration has been inlined into the send statement (line 7) replacing the source reference. Since this was the only reference, the source template declaration is not needed anymore and has been removed. No imports must be adjusted because the source template declaration and the source reference are both in the same module.

```java
1 type record ExampleType {  
2 boolean ipv6,  
3 charstring ipAddress  
4 }  
5  
6 template ExampleType exampleTemplate := {  
7 ipv6 := false,  
8 ipAddress := "127.0.0.1"  
9 }  
10  
11 testcase exampleTestCase() runs on ExampleComponent {  
12 examplePort.send(exampleTemplate);  
13 }
```

Listing 2. Inline Template (unrefactored).
4.2.2. Refactoring: Inline Template Parameter

Summary: A formal parameter of a template that is always given the same actual value is inlined.

Motivation: Templates are typically parameterized to avoid multiple template declarations that differ only in a few values. However, as test suites grow and change over time, the usage of its templates may change as well. As a result, there may be situations when all references to a parameterized template have one or more actual parameters with the same values. This can also happen when the test engineer is overly eager: a template is parameterized as it is believed that it might be useful for future use, but it later turns out to be unnecessary. In any case, there are template references with unneeded parameters creating code clutter and more complexity than useful. Thus, the template parameter should be inlined and removed from all references.

Mechanics:

- Verify that all template references to the parameterized source template declaration have a common actual parameter value. The parameter with the common actual parameter value is the source parameter. Record the common value.
  - If more than one formal parameter is always referenced with a common actual parameter value, it is easier to inline them together. Therefore, perform each step that concerns the source parameter for all concerned parameters at once.

- Copy the source template declaration within the same scope and group. Give the copied declaration a temporary name. This is the target template declaration.

- In the target template declaration body, replace each reference to the source formal parameter with the value noted in the first step. In the target template declaration signature, remove the formal parameter corresponding to the source parameter.

- Remove the source template declaration.

- Rename the target template declaration using the name of the source template declaration.

- Find all references to the target template declaration. Remove the source parameter from the actual parameter list of each reference.

- Consider usage of the Rename refactoring to improve the target template declaration name.

Example: Listing 4 contains the parameterized template ‘exampleTemplate’ in lines 6–9. All references to this template use the same actual parameter value (lines 12 and 13). Hence, the corresponding parameter ‘addressParameter’ in line 6 is inlined.
4.2.3. Refactoring: Parameterize Template

Summary: Several templates of the same type, which merely use few different field values, are replaced by a single parameterized template.

Motivation: Occasionally, there are several template declarations of the same type that are basically similar but vary in few values at the same fields. These template declarations are candidates for parameterization. Instead of keeping all of them, they are replaced with a single template declaration where the variations are handled by template parameters. Such a change removes code duplication, improves reusability, and increases flexibility. If the names of the source templates convey important information that would get lost by using a single template, their names may be kept by delegating them to the target template. If all template fields are completely the same, use the Remove Duplicate
Template refactoring. If the template declarations are similar, but the values vary in different fields, the Replace Template with Modified Template refactoring may be a better choice.

**Mechanics:**

- First check that none of the source templates are referenced by another template using modifies and that all of the source templates are located in the same scope and group. Otherwise, the mechanics of this refactoring are not applicable.
- Create the parameterized target template declaration within the same scope and group as the source templates by copying one source template declaration. Give the target template a name that reflects the meaning of the non-parameterized template values.
- Introduce a formal parameter to the target template declaration for each field in which the source template values differ:
  - Add the formal parameter to the target template declaration signature. The type of the formal parameter can be obtained from the type of the respective template field.
  - Inside the target template body, replace the value of the varying template field with a reference to the added formal parameter.
- Now, decide whether you would like to keep the source template names for delegation or only the parameterized target template.
  - If you would like to remove the source templates completely and keep only the parameterized target template:
    - Repeat the following steps for all references to the source template declarations:
      - Replace the reference with a reference to the parameterized target template. As actual parameter values, use the field values from the originally referenced template declaration corresponding to the parameterized fields in the target template.
    - Remove the source template declarations. They should not be referenced anymore.
  - If you would like to keep the source templates and want to just delegate to the parameterized target template:
    - Repeat the following steps for all source template declarations:
      - Replace the template body with a reference to the parameterized target template. As actual parameter values, use the values from the original template body fields corresponding to the parameterized fields in the target template.

**Example:** Listing 6 shows an excerpt of an unrefactored example. The source template declarations ‘firstTemplate’ (lines 6–9) and ‘secondTemplate’ (lines 11–14) differ only in the values of ‘ipAddress’.

The resulting code after applying the variant of Parameterize Template where the source templates are completely removed is shown in Listing 7. A new target template declaration ‘parameterizedTemplate’ (lines 6–9) is created, which has a parameter for the varying ‘ipAddress’ field in the source template declarations. The references to ‘firstTemplate’ (line 12) and ‘secondTemplate’ (line 13) are replaced with ‘parameterizedTemplate’ and their corresponding ‘ipAddress’ values as parameters.
5. RULE-BASED ISSUE DETECTION

The application of refactorings can be steered by rules. Several rules have been identified that help to improve the quality of TTCN-3 test specifications with respect to test data, i.e. template definitions. Most of these rules can be directly related to the metrics described in Section 3 and to the refactorings presented in Section 4. Some rules follow general maintainability quality characteristics, whereas others support the quality subcharacteristics analysability and changeability. Rules supporting different quality subcharacteristics may contradict each other and therefore cannot be applied together. For example, the analysability subcharacteristic can be improved by inlining all template definitions, i.e. the test engineer does not need to search for the definitions of the referenced test data. On the other hand, inlining templates that are referenced several times decreases the
changeability of a test specification, because it leads to code duplication. Therefore, a test engineer (or a tool) has to select the rules according to the quality subcharacteristic that is to be improved.

Rules that improve the maintainability quality characteristic in general:

**Rule 1**: A template definition that is not referenced (metric value: \( \text{Number of references to a template} = 0 \)) should be removed.

**Rule 2**: A template definition that is only referenced once (metric value: \( \text{Number of references to a template} = 1 \)) should be inlined and its definition should be removed (application of Inline Template refactoring that, for parameterized templates, includes the inlining of parameters).

**Rule 3**: A template definition in which all fields receive their values by means of parameters should be inlined and its definition removed (application of Inline Template refactoring).

**Rule 4**: Unused parameters of a template definition (e.g., parameters that are not used in assignments to fields) should be removed altering the template definition (application of Remove Parameter refactoring).

Rules that improve the quality subcharacteristic analysability:

**Rule 5**: For a template definition that is referenced multiple times and that has formal parameters that do not adhere to Rule 3 or 4, the following rules apply:

(a) If all instantiations of a template are the same, i.e., all formal parameters are given the same values each time the template is referenced, then the formal parameters are removed and the assigned elements are defined explicitly (application of Inline Template Parameter refactoring).

(b) If instantiations of a template vary, i.e., all formal parameters are given different values, formal parameters account for values of 50% or more of the fields within the template definition, and then the template shall be inlined and its definition shall be removed (application of Inline Template refactoring).

**Rule 6**: A template definition that is referenced multiple times (metric value: \( \text{Number of references to a template} > 1 \)) should be inlined and its definition should be removed (application of Inline Template refactoring). This rule maximizes the Template coupling score.

Rules that improve the quality subcharacteristic changeability:

**Rule 7**: A template definition without parameters that is referenced multiple times (metric value: \( \text{Number of references to a template} > 1 \)) should not be altered. This rule leads to a Template coupling score close to 1.

**Rule 8**: If two or more template definitions exist for the same type, then the following rules could apply:

(a) If template values only differ for the same template fields and these differing fields account for a certain percentage (assume 30%) of the overall fields for the template definition, then the templates can be reduced to a single parameterized definition (application of Parameterize Template refactoring).

(b) If template values differ for different template fields, then nothing is done as the test engineer would have to choose which field to parameterize upon.
For a fully automated quality improvement approach, the rules for improving the maintainability quality characteristic in general need to be applied first and, afterwards, either the rules for improving the quality subcharacteristic analysability or those for improving the quality subcharacteristic changeability are applicable. However, for a more selective and interactive approach, it is also possible to apply these rules individually.

The presented rules can only give an impression of how metrics can steer the refactoring process. Currently, these rules are refined and new rules are defined which also support the refactoring of test behaviour and the TTCN-3 module structure. This includes the definition of further metrics to underpin the rules and the analysis of the influence of the rule ordering.

6. TRex

To automate the quality assurance of TTCN-3 test specifications, the TTCN-3 refactoring and metrics tool TRex has been developed. TRex is able to calculate metrics, perform refactorings, and apply rules as described in Sections 3–5.

6.1. Functionality of the TRex tool

The TRex tool is implemented as a set of Eclipse plug-ins; therefore, everyone who has experience with the Eclipse Platform [25], e.g. by using the popular Java Development Tools (JDT), will immediately feel comfortable with TRex. The TTCN-3 perspective of TRex (Figure 1) allows editing of TTCN-3 core notation as well as assessing and improving the quality of TTCN-3 test suites.

TRex provides editing facilities known from a typical Integrated Development Environment (IDE). These include a Navigator view for project browsing, an editor with syntax highlighting and checking according to the TTCN-3 core language specification v3.1.1, an Outline view providing a tree representation of the TTCN-3 structure for the currently edited file, a Problems view displaying any issues found in a test suite, Content Assist that automatically completes identifiers from their prefix and scope, a code formatter, a reference finder that displays all references to a given element, and the possibility to jump to the declaration of a given reference. In addition, to allow the edited tests to be compiled and run against either a real or an emulated SUT, TRex provides an interface to call external TTCN-3 compilers.

6.1.1. Refactoring

To support the automated application of the refactorings described in Section 4, the transformation steps of a refactoring, which are described in its corresponding mechanics section, have been implemented in TRex. Those refactorings from the TTCN-3 refactoring catalogue which were most important to improve the maintainability of Motorola’s test suites have been implemented first. So far the implementation of the Inline Template, Extract Template, Decompose Template, Replace Template with Modified Template, Merge Templates, Inline Template Parameter, Parameterize Template, Move Module Constants to Component, and Rename refactorings has been completed.
For example, the Inline Template refactoring allows a template reference to be transformed into its semantically equivalent inline template notation. The application of this refactoring is particularly reasonable if a template is referenced only once (Rule 2 from Section 5). When applying this refactoring to a template reference, TRex opens a wizard dialogue that offers configuration for the Inline Template refactoring. As shown in Figure 2, it is possible to remove the declaration of a template if it was referenced only once and the code formatter may additionally be used to obtain...
a pretty-printed inline template. Before a refactoring is actually applied, the refactoring wizard displays a preview of all resulting changes (Figure 3).

These refactorings are typically semi-automated, since the test engineer still has to identify locations where they should be applied (as known from JDT for example). However, as shown in the following section, TRex can also automatically identify such locations.

6.1.2. Metrics and rule-based issue detection

TRex is able to automatically calculate the metrics presented in Section 3. The calculated metric values of a TTCN-3 test suite are displayed in the TTCN-3 Metrics view as shown in Figure 4(a). By using a tree view, different levels of aggregation of the metrics values are possible. In addition to displaying the calculated metric values to the test engineer, TRex uses them as input for the rules described in Section 5. These rules have been implemented as well and allow TRex to identify problematic code fragments and to suggest suitable refactorings. These suggestions are displayed in the Problems view as warnings (Figure 4(c)) and can either be treated merely as indicators that should be taken into account while working on the test suite, or an associated Quick Fix can be invoked via the context menu to perform a suggested refactoring automatically (Figure 4(b)).

6.2. Implementation of the TRex tool

Building an IDE on Eclipse is attractive from the developer’s point of view, as it is well documented and supported and provides many ready-to-use components. Such components include project and file management (workspace) and a graphical user interface (workbench), which can be configured to match the typical layout of an IDE. In fact, the majority of TRex’s functionality is built upon
abstract implementations provided by Eclipse. The TRex tool chain is shown in Figure 5: the Eclipse Platform provides the basic IDE infrastructure. The TRex components build on top of the Eclipse Platform. They are explained in the subsequent sections.

6.2.1. Static analysis

The foundation for most functionality in TRex is the TTCN-3 parser and the resulting syntax tree. For building up the syntax tree for a test suite ANother Tool for Language Recognition (ANTLR) [26] is used, a parser generator that supports lexing, parsing, and syntax tree creation and traversal. For tree traversal, ANTLR uses tree grammars, e.g. the pretty printer uses a tree grammar enriched with semantic actions for the syntax reconstruction and code formatting.

Most of the advanced functionality of TRex requires additional information for TTCN-3 identifiers, such as the identifier’s type, or the syntax tree node of its declaration. To easily find this information, a symbol table was implemented. The syntax tree and the symbol table provide the basis upon which most of TRex’s present functionality is realized, e.g. the metrics and refactoring suggestions.
implementations both use them. As shown in Block 1 of Figure 5, the lexer creates a token stream from the TTCN-3 core notation, which is used by the parser for syntactical validation and for creating the syntax tree and the symbol table.

### 6.2.2. The refactoring implementation

The refactoring implementations make use of the Eclipse Language Toolkit, which provides abstract classes for semantic preserving workspace transformations and customizable wizard pages for user interaction. The benefit of such wizard pages is, for example, an integrated preview pane that can be used to compare the original source to the refactored source side by side.

Block 2 in Figure 5 depicts how the automated refactorings are realized. On the basis of the static analysis step (Block 1), the workspace transformations can be calculated once the concerned syntax tree node (or nodes respectively) has been found through a data structure that stores identifiers along with their text file offsets and once the user enters any required information in the refactoring wizard.

The transformation of the workspace resources, i.e. the text files containing the TTCN-3 source files, is realized with a programmatic text editor provided by the Eclipse Platform. It supports copy, paste, move, delete, insert, and replace operations. These operations are used to weave only the textually changed parts into the original TTCN-3 source files. Therefore, most of the original formatting is preserved. In some cases an intermediate step involving a syntax tree transformation may become necessary in order to calculate the required changes. In this case, the TTCN-3 core notation to be weaved into the original TTCN-3 source files is obtained by the pretty printer. Applying multiple changes to a single file is supported by the programmatic editor by automatically tracking changing offset positions.
6.2.3. Metrics and rule-based issue detection

Metrics are measured immediately after the syntax tree for a test suite has been built or updated (Block 3 in Figure 5). The tree is then fully traversed; all definitions that metrics will be calculated for (e.g. altstep, function, or template) are recorded and all communication operations (e.g. send or receive) are processed to derive Template coupling scorings. References to all of these are calculated in a further pass of the syntax tree, hence giving enough information for the basic size metrics (described in Section 3) to be calculated. Then all templates found in the first step are processed one by one against the analysis rules, using the previously calculated referencing information as well as further inspection of their structure. For calculating the complexity metrics mentioned in Section 3, control flow graphs and call graphs are created for each test behaviour. Based on these graphs, metrics such as McCabe’s Cyclomatic Complexity are calculated.

Once this has been completed, the rule findings are associated with the templates in the form of customized Eclipse marker objects that are automatically displayed in the Problems view (Figures 1 and 4(c)). Quick Fixes are resolved for each of them, based on extended attributes which indicate the detected situation and hence some corresponding suggestion(s) from the rule set.

7. APPLYING TREX

To evaluate the concepts implemented in TRex, three different Abstract Test Suites (ATSs), i.e. TTCN-3 test suites, from ETSI have been refactored by applying the automatically generated rule-based refactoring suggestions: the Session Initiation Protocol (SIP) ATS [27], the Internet Protocol Version 6 (IPv6) ATS [28], and the Worldwide Interoperability for Microwave Access (WiMAX)/High Performance Metropolitan Area Network (HiperMAN) Subscriber Station ATS [29]. Some basic size measures of these test suites can be found in Table II. For this article, the ETSI test suites have been chosen over internal Motorola test suites since they are publicly available. The process of applying TRex had a focus on improvements of template definitions concerning the maintainability quality characteristic and in particular the changeability subcharacteristic, i.e. unused template definitions have been removed (Rule 1 from Section 5), singly referenced template definitions have been inlined (Rule 2), and similar templates have been merged (Rule 8a). By removing unused template definitions and inlining singly referenced ones, code clutter is reduced. By merging templates changeability is enhanced due to greater flexibility and by reducing the volume of the test suite source code through parameterization the analysability quality subcharacteristic may also benefit.

<table>
<thead>
<tr>
<th>Table II. Size of ETSI test suites.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>SIP 4.1.1</td>
</tr>
<tr>
<td>----------------</td>
</tr>
<tr>
<td>Lines of code</td>
</tr>
<tr>
<td>Number of templates</td>
</tr>
<tr>
<td>Number of test cases</td>
</tr>
</tbody>
</table>

The charts in Figures 6 and 7 visualize the effect of those refactorings on the ATSs in terms of the *Number of lines of TTCN-3 source code* metric (physical lines of code) and the *Number of templates* metric. The measurements indicate that results certainly depend on how much a test suite has already been optimized with respect to the factors mentioned. For the SIP ATS, for example, the effect is much less noticeable—especially in terms of lines of code, since the template definitions do not constitute the majority of the test suite volume. The effect on the IPv6 ATS, on the other hand, is clearly visible. The number of template definitions could be reduced to less than half the
number, and more than 5000 lines of code could be saved. It should be noted that the number of inlined templates can be neglected in this case. The impact originates from removing unused and merging similar or duplicate templates. The WiMAX/HiperMAN ATS yields a similar result. The number of templates could be reduced by approximately a third and the test suite size could be reduced by more than 2000 lines of code.

When taking into account that these are the results of merely three refactoring rules which have been applied, a higher number of implemented rules and refactorings, also supporting behavioural and structural quality aspects, is likely to have an even more noticeable impact on the test suite source code.

8. RELATED WORK

Most of the known metrics related to tests concern processes and resources, but not products, i.e. test suites. Two principal publications are known which relate to product metrics for test suites: Sneed [30] provides size metrics for test suites, as well as metrics for measuring the complexity and quality of suites. However, these metrics are too abstract to take the peculiarities of TTCN-3 into account. Vega et al. [31] list some internal and external size metrics, which they suggest could be applied to TTCN-3 tests suites, but it is not clear how these metrics can be interpreted to assess the actual quality of the suite. In contrast, the TTCN-3 metrics presented in this article have a practical relevance since they are used to steer the application of refactorings or to measure the effect of the refactoring process.

Existing work on refactoring deals mainly with the refactoring of source code from general-purpose programming languages and very little is published on the refactoring of test specifications. Concerning TTCN-3 and its predecessor, the Tree and Tabular Combined Notation (TTCN-2) [32], three publications [33–35] deal with transformations that can be regarded as refactoring. Schmitt [34] and Wu-Hen-Chang et al. [35] propose solutions for the automatic restructuring of test data descriptions. Even though different approaches are chosen and Schmitt treats the constraints of TTCN-2, whereas Wu-Hen-Chang et al. deal with TTCN-3 templates, both apply semantics-preserving operations to the test data description. In fact, these operations are refactorings. They are based on concepts which are available in both test languages to specialize, parameterize, and reference test data descriptions. Deiß [33] improves the TTCN-3 code generated by an automated conversion of a TTCN-2 test suite by applying some refactoring-like transformations. For example, TTCN-3 altsteps that contain only an else branch starting with a send statement are transformed into a more appropriate TTCN-3 function. While these publications treat only a limited set of test refactorings, the TTCN-3 refactoring catalogue presented in this article goes beyond the existing work by being more extensive and by providing for each refactoring detailed step-by-step instructions and examples for their application.

Tool support for calculating metrics and for automating refactorings exists for general-purpose programming languages, e.g. the Eclipse JDT [25] provides automated refactorings for Java. Concerning test specification languages, TRex was the first publicly available tool to provide automated refactoring and metric calculation for TTCN-3. In the meantime, the TTCN-3 IDE TTworkbench [7] has been extended to provide initial support for refactoring.
9. SUMMARY AND OUTLOOK

In this article, an approach to the general assessment and quality improvement of TTCN-3 test suites has been presented. It comprises the definition of a set of TTCN-3 metrics which are used for quality assessment. For quality improvement, a TTCN-3 refactoring catalogue is provided, which can be applied to TTCN-3 test suites. Furthermore, to enable the automation of this quality improvement process, the approach is supplemented with a set of rules that interpret metric values to detect issues and to make suggestions for appropriate refactoring.

These concepts have been implemented in the TTCN-3 refactoring and metrics tool TRex. The applicability of the presented approach has been demonstrated by using TRex to improve the quality of several standardized TTCN-3 test suites in terms of maintainability and, in particular, changeability. In addition to the application of TRex in industry, TRex is also being used as a tool for academic research and teaching. TRex is open source and is freely available at its Web site [23].

Further metrics, refactorings, rules, and analyses for TTCN-3 test suites are the subject of current research. As already mentioned in Section 3, metrics are not always sufficient to detect issues, but an additional pattern-based approach may be needed for locating issues. As a first result from this work, a pattern-based TTCN-3 code smell catalogue has been developed [14,15] and is currently being investigated in more detail.

Quality assurance may be directed at several different quality (sub) characteristics, some of which may even contradict each other. For example, in Sections 3 and 5, the analysability quality subcharacteristic has been considered in terms of readability. However, other aspects, which were not considered, contribute to analysability as well. Hence, a general quality model for test specifications in the spirit of the ISO/IEC software product quality model [16] is required. Such a quality model may then be instantiated by giving weights to the different (sub) characteristics and choosing appropriate metrics and threshold values. Thus, a first attempt at defining such a model has just been started [17].

Future research will include dynamic analysis of TTCN-3 test suites. This allows the calculation of external attributes of TTCN-3 test suites: some properties, in particular those related to TTCN-3 defaults which can be activated and deactivated during runtime, cannot be assessed statically, but require a dynamic approach. Furthermore, a dynamic approach allows an exhaustive simulation to generate all possible traces of two variants of a test suite: by comparing all possible traces, it is possible to validate behavioural equivalence between unrefactored and refactored test suites when refactorings are not performed by a tool, but in an error-prone manual way.

Finally, it seems worthwhile to apply the presented quality assessment and improvement approach also to other test specification languages, e.g. to the UML 2.0 Testing Profile (U2TP) [36].

REFERENCES


