SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2008; **18**:1–28 (DOI: 10.1002/stvr.379)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr.379

Quality assurance for TTCN-3 test specifications[‡]

Helmut Neukirchen[1,*,†], Benjamin Zeiss[1], Jens Grabowski[1],
Paul Baker[2] and Dominic Evans[2]

[1] *Software Engineering for Distributed Systems Group, Institute for Computer Science, University of Göttingen, Germany*
[2] *Motorola 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 standardised *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 standardised 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).

International Telecommunication Union (ITU). 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 not only applicable for specifying and implementing functional tests, but also for scalability, robustness, or stress tests. Commercial TTCN-3 tools [2, 3, 4, 5, 6, 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, hard to (re-)use, and hard to 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) 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.

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.

```
 1 module exampleModule {
 2   type record ExampleType {
 3     boolean ipv6,
 4     charstring ipAddress
 5   }
 6
 7   type port ExamplePortType message {
 8     inout ExampleType
 9   }
10
11   type component ExampleComponentType {
12     port ExamplePortType examplePort
13   }
14
15   template ExampleType exampleTemplate := {
16     ipv6 := false ,
17     ipAddress := "127.0.0.1"
18   }
19
20   altstep exampleAltStep() runs on ExampleComponentType {
21     [ ] examplePort.receive(ExampleType:?) {
22       setverdict ( fail );
23     }
24   }
25
26   testcase exampleTestCase() runs on ExampleComponentType {
27     examplePort.send(exampleTemplate);
28     alt {
29       [ ] examplePort.receive(ExampleType:{false, "127.0.0.2"}) {
30         setverdict (pass);
31       }
32       [ ] exampleAltStep();
33     }
34   }
35 }
```

Listing 1. A TTCN-3 example test suite

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' and 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 is 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 not only allows 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, like 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 **altstep**s 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].

Already the simple example provided in Listing 1 demonstrates some maintenance issues with respect to templates: the actual data that is specified by a template is 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 parametrised: on the one hand, several non-parametrised template declarations may be replaced by a single parametrised 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 external attributes of a product, execution of the product is required, whereas, for measuring internal attributes, static analysis is sufficient. Since this article treats quality characteristics like maintainability of TTCN-3 test specifications, only internal product attributes are considered in the remainder.

Internal product metrics can be structured into *size* and *structural* metrics. Size metrics typically measure properties of the number of usage 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 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 parametrised 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 easy modifications can be implemented in a test specification, analysability refers to how easy 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 which 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* (GQM) 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 characterises 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 provides 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 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 parametrised templates*.
- *Template coupling*, which is to be computed as follows:

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

Where $stmt$ is the sequence of behaviour statements referencing templates in a test suite, n is the number of statements in $stmt$, and $stmt(i)$ denotes the ith statement in $stmt$. $score(stmt(i))$ is defined as follows:

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

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 source code, it is more systematical and thus less error prone than arbitrary code clean-up, because each refactoring provides a checklist of small and simple transformation steps which 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, to encapsulate the public fields of the data classes, and to extract and move 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 popularised by Fowler and his book "*Refactoring – Improving the Design of Existing Code*" [13] which contains a catalogue of 72 refactorings which 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 which are proven to be behaviour preserving, Fowler suggests a manual approach in which the validation is realised 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 *slanted* type. 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 which is addressed by a refactoring and thus

Table I. Fowler refactorings applicable to TTCN-3

Refactorings for test behaviour	Refactorings for improving the overall structure of a test suite
• *Consolidate Conditional Expression* • *Consolidate Duplicate Conditional Fragments* • *Decompose Conditional* • *Extract Function* [*Extract Method*] • *Introduce Assertion* • *Introduce Explaining Variable* • *Inline Function* [*Inline Method*] • *Inline Temp* • *Remove Assignments to Parameters* • *Remove Control Flag* • *Replace Nested Conditional with Guard Clauses* • *Replace Temp with Query* • *Separate Query From Modifier* • *Split Temporary Variable* • *Substitute Algorithm*	• *Add Parameter* • *Extract Extended Component* [*Extract Subclass*] • *Extract Parent Component* [*Extract Superclass*] • *Introduce Local Port/Variable/Constant/Timer* [*Introduce Local Extension*] • *Introduce Record Type Parameter* [*Introduce Parameter Object*] • *Parametrise Testcase/Function/Altstep* [*Parameterize Method*] • *Pull Up Port/Variable/Constant/ Timer* [*Pull Up Field*] • *Push Down Port/Variable/Constant/ Timer* [*Push Down Field*] • *Replace Magic Number with Symbolic Constant* • *Remove Parameter* • *Rename* [*Rename Method*] • *Replace Parameter with Explicit Functions* [*Replace Parameter with Explicit Methods*] • *Replace Parameter with Function* [*Replace Parameter with Method*]

usually removed or simplified and the term *target* to refer to code which 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 which 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 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 which 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* which do not exist in Java. However, some of Fowler's refactorings like *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 next 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 which are specific to TTCN-3. Currently, these refactorings take advantage of TTCN-3 altsteps, templates, grouping, modules and importing from modules, test components, restricted sub-types, 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 which are not closely related to each other are split to maximise reuse potential.
- *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.
- *Add Explanatory Log:* Add a **log** statement to explain why a test case aborted or a non-**pass** verdict was assigned.
- *Distribute Test:* Transform a non-concurrent test case into a distributed concurrent test case.

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.
- *Parametrise Module:* Parametrise 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 which 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.
- *Generalise 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.
- *Parametrise Template:* Several templates of the same type, which merely use different field values, are replaced by a single parametrised template.
- *Remove Duplicate Template:* Replace several template declarations which have the same body with one single template.
- *Inline Template Parameter:* A formal parameter of a template which 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 more easily detect code flaws.

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 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 it 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 parametrised 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.

```
 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)

```
type record ExampleType {
  boolean ipv6,
  charstring ipAddress
}

testcase exampleTestCase() runs on ExampleComponent {
  examplePort.send(ExampleType:{false, "127.0.0.1"});
}
```

Listing 3. Inline Template (refactored)

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.

4.2.2. *Refactoring: Inline Template Parameter*

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

Motivation: Templates are typically parametrised 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 parametrised 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 parametrised as it is believed 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 parametrised 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 name of 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 parametrised 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.

After applying the *Inline Template Parameter* refactoring (Listing 5), the string value ‘127.0.0.1’ is inlined into the template body of ‘exampleTemplate’ (Line 8), and the corresponding formal parameter of the template (Line 6) and the corresponding actual parameter of each reference to ‘exampleTemplate’ (lines 12 and 13) are removed.

```
type record ExampleType {
  boolean ipv6,
  charstring ipAddress
}

template ExampleType exampleTemplate(charstring addressParameter) := {
  ipv6 := false,
  ipAddress := addressParameter
}

testcase exampleTestCase() runs on ExampleComponent {
  examplePort.send(exampleTemplate("127.0.0.1"));
  examplePort.receive(exampleTemplate("127.0.0.1"));
}
```

Listing 4. Inline Template Parameter (unrefactored)

```
type record ExampleType {
  boolean ipv6,
  charstring ipAddress
}

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

testcase exampleTestCase() runs on ExampleComponent {
  examplePort.send(exampleTemplate);
  examplePort.receive(exampleTemplate);
}
```

Listing 5. Inline Template Parameter (refactored)

4.2.3. *Refactoring: Parametrise Template*

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

Motivation: Occasionally, there are several template declarations of the same type which are basically similar, but vary in few values at the same fields. These template declarations are candidates for parametrisation. 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 which 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 parametrised 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-parametrised 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 like to keep the source template names for delegation or whether you like to keep only the parametrised target template.
  - If you like to remove the source templates completely and keep only the parametrised target template:
    - Repeat the following steps for all references to the source template declarations:
      - Replace the reference with a reference to the parametrised target template. As actual parameter values, use the field values from the originally referenced template declaration corresponding to the parametrised fields in the target template.
    - Remove the source template declarations. They should not be referenced anymore.
  - If you like to keep the source templates and want just delegate to the parametrised target template:
    - Repeat the following steps for all source template declarations:
      - Replace the template body with a reference to the parametrised target template. As actual parameter values, use the values from the original template body fields corresponding to the parametrised 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 *Parametrise Template* where the source templates are completely removed is shown in Listing 7. A new target template declaration ‘parametrisedTemplate’ (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 ‘parametrisedTemplate’ and their corresponding ‘ipAddress’ values as parameters.

```
type record ExampleType {
  boolean ipv6,
  charstring ipAddress
}

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

template ExampleType secondTemplate := {
  ipv6 := false,
  ipAddress := "127.0.0.2"
}

testcase exampleTestCase() runs on ExampleComponent {
  examplePort.send( firstTemplate );
  examplePort.receive(secondTemplate);
}
```

Listing 6. Parametrise Template (Unrefactored)

```
type record ExampleType {
  boolean ipv6,
  charstring ipAddress
}

template ExampleType parametrisedTemplate(charstring addressParameter) := {
  ipv6 := false,
  ipAddress := addressParameter
}

testcase exampleTestCase() runs on ExampleComponent {
  examplePort.send(parametrisedTemplate("127.0.0.1"));
  examplePort.receive(parametrisedTemplate("127.0.0.2"));
}
```

Listing 7. Parametrise Template (Refactored)

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 cannot therefore 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 referenced test data. On the other hand, inlining templates which 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 which is not referenced (Metric value: *Number of references to a template = 0*) should be removed.

Rule 2: A template definition which is only referenced once (Metric value: *Number of references to a template = 1*) should be inlined and its definition should be removed (Application of *Inline Template* refactoring which, for parametrised 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 which 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 which is referenced multiple times and which has formal parameters that do not adhere to Rules 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 the values of 50% or more of the fields within the template definition, then the template shall be inlined and its definition be removed (Application of *Inline Template* refactoring).

Rule 6: A template definition which is referenced multiple times (Metric value: *Number of references to a template* > 1) should be inlined and its definition should be removed (Application of *Inline Template* refactoring). This rule maximises the *Template coupling* score.

Rules that improve the quality subcharacteristic changeability

Rule 7 A template definition without parameters which is referenced multiple times (Metric value: *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 parametrised definition (Application of *Parametrise 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 parametrise 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 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 support as well 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, to perform refactorings, and to apply rules as described in sections 3–5.

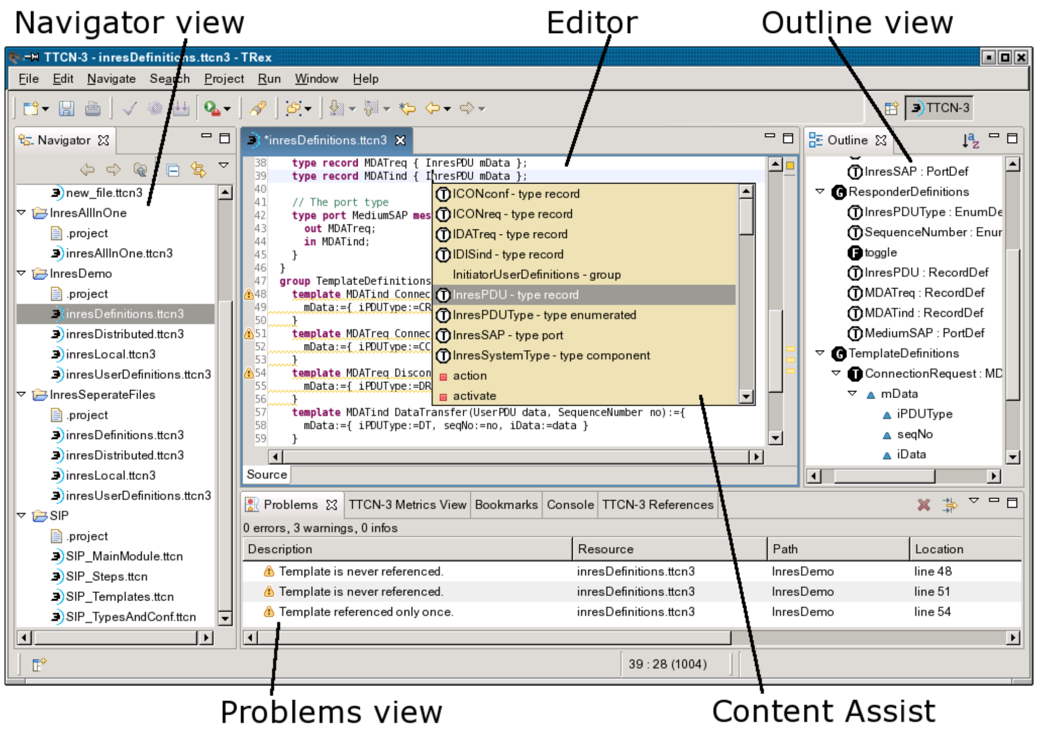
6.1. Functionality of the TRex tool

The TRex tool is implemented as a set of Eclipse plug-ins and 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* which automatically completes identifiers from their prefix and scope, a code formatter, a reference finder which 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 emulated system under test, 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



```
    type record MDATreq { InresPDU mData };
    type record MDATind { InresPDU mData };

    // The port type
    type port MediumSAP mes
      out MDATreq;
      in MDATind;
    }
  }
  group TemplateDefinitions
    template MDATind Connec
      mData:={ iPDUType:=CR
    }
    template MDATreq Connec
      mData:={ iPDUType:=CC
    }
    template MDATreq Discon
      mData:={ iPDUType:=DR
    }
    template MDATind DataTransfer(UserPDU data, SequenceNumber no):={
      mData:={ iPDUType:=DT, seqNo:=no, iData:=data }
    }
```

0 errors, 3 warnings, 0 infos

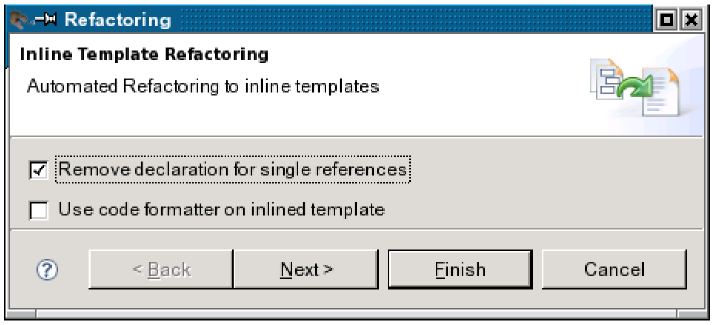
Description	Resource	Path	Location
Template is never referenced.	inresDefinitions.ttcn3	InresDemo	line 48
Template is never referenced.	inresDefinitions.ttcn3	InresDemo	line 51
Template referenced only once.	inresDefinitions.ttcn3	InresDemo	line 54

Figure 1. The TRex TTCN-3 perspective

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*, *Parametrise 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 only referenced once (Rule 2 from Section 5). When applying this refactoring to a template reference, TRex opens a wizard dialogue which 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 next section, TRex can also automatically identify such locations.



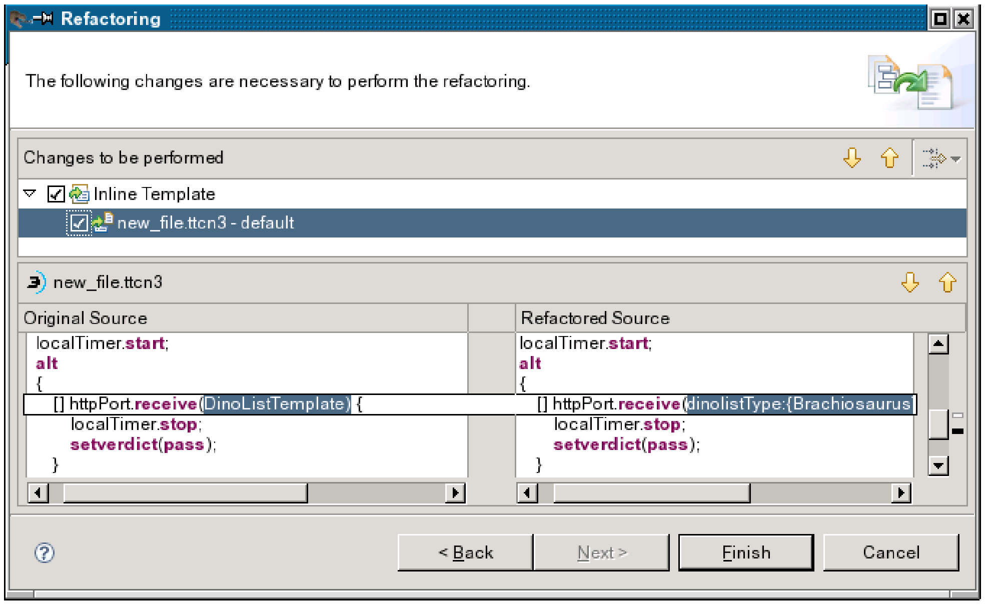
Inline Template Refactoring

Automated Refactoring to inline templates

☑ Remove declaration for single references

☐ Use code formatter on inlined template

Figure 2. Wizard for the configuration of the *Inline Template* refactoring



The following changes are necessary to perform the refactoring.

Changes to be performed

☑ Inline Template

☑ new_file.ttcn3 - default

new_file.ttcn3

Original Source

```
localTimer.start;
alt
{
  [] httpPort.receive(DinoListTemplate) {
     localTimer.stop;
     setverdict(pass);
  }
```

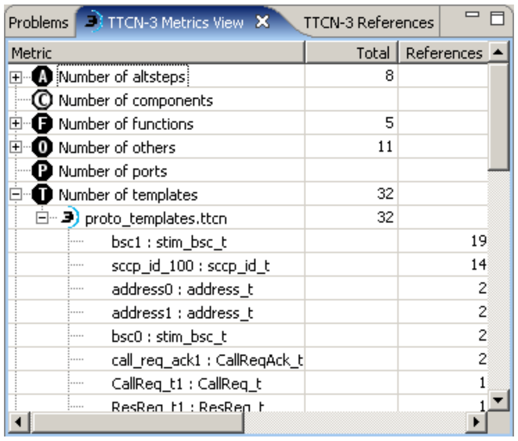
Refactored Source

```
localTimer.start;
alt
{
  [] httpPort.receive(dinolistType:{Brachiosaurus
     localTimer.stop;
     setverdict(pass);
  }
```

Figure 3. Wizard for the preview of the *Inline Template* refactoring

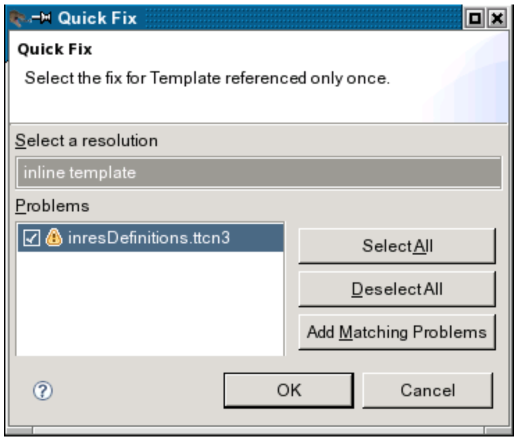
6.1.2. Metrics and rule-based issue detection

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 4a. 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



Metric	Total	References
Number of altsteps	8	
Number of components		
Number of functions	5	
Number of others	11	
Number of ports		
Number of templates	32	
proto_templates.ttcn	32	
bsc1 : stim_bsc_t		19
sccp_id_100 : sccp_id_t		14
address0 : address_t		2
address1 : address_t		2
bsc0 : stim_bsc_t		2
call_req_ack1 : CallReqAck_t		2
CallReq_t1 : CallReq_t		1
ResReq_t1 : ResReq_t		1

(a) TTCN-3 Metrics view



Quick Fix

Quick Fix

Select the fix for Template referenced only once.

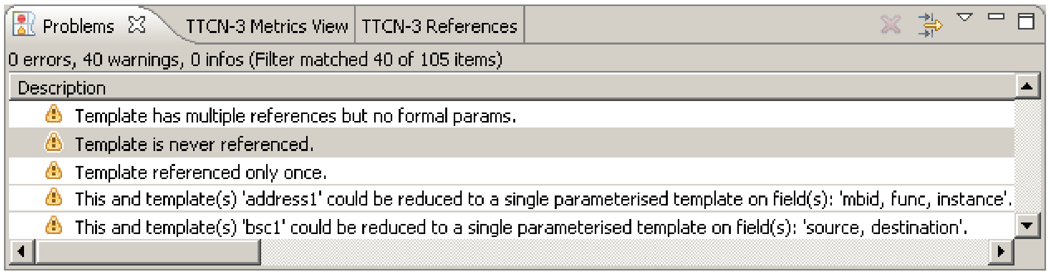
Select a resolution

inline template

Problems

inresDefinitions.ttcn3

(b) TRex's Quick Fix suggestion



0 errors, 40 warnings, 0 infos (Filter matched 40 of 105 items)

Description

Template has multiple references but no formal params.

Template is never referenced.

Template referenced only once.

This and template(s) 'address1' could be reduced to a single parameterised template on field(s): 'mbid, func, instance'.

This and template(s) 'bsc1' could be reduced to a single parameterised template on field(s): 'source, destination'.

(c) TRex's rule-based refactoring suggestions

Figure 4. TRex's metrics-based functionality

as warnings (Figure 4c) and can either be treated merely as indicators that should be taken into account whilst working on the test suite, or an associated *Quick Fix* can be invoked via the context menu to perform a suggested refactoring automatically (Figure 4b).

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.

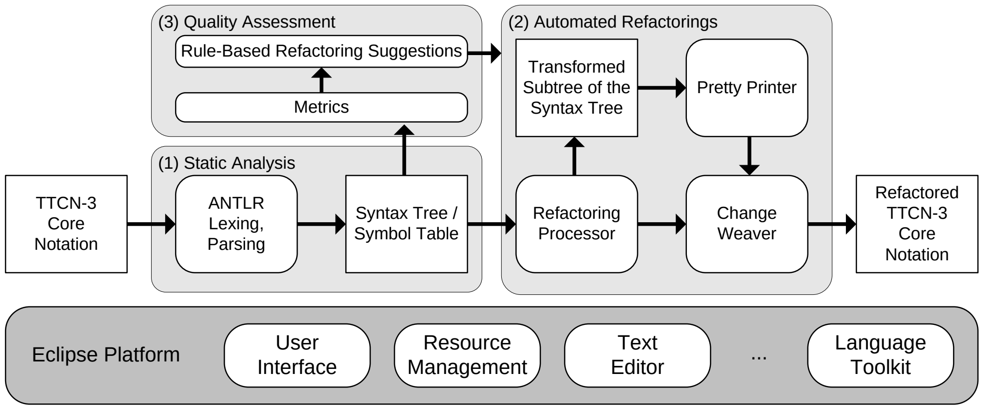


Figure 5. The TRex tool chain

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 which 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 realised, e.g. the metrics and refactoring 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* (LTK) which provides abstract classes for semantic preserving workspace transformations and customisable wizard pages for the 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 realised. 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 which stores identifiers along with their text file offsets and once the user entered 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 realised 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 and call graphs are created for each test behaviour. Based on these graphs, metrics like McCabe's Cyclomatic Complexity are calculated.

Once this has completed, the rule findings are associated with the templates in the form of customised Eclipse *marker* objects which are automatically displayed in the *Problems view* (figures 1 and 4c). *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 Suite*s (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

Table II. Size of ETSI test suites

	SIP 4.1.1	IPv6 1.1	HiperMAN 2.2.1
Lines of code	61282	41801	22803
Number of templates	383	148	456
Number of test cases	609	286	72

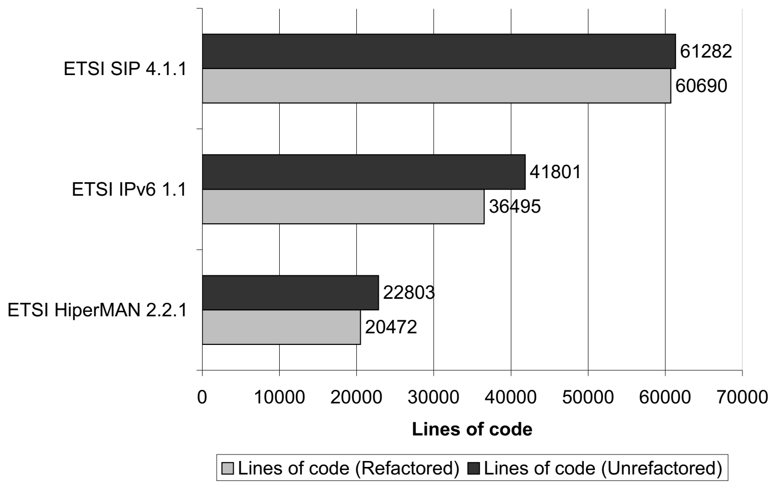


Figure 6. Lines of code before and after applying the refactorings

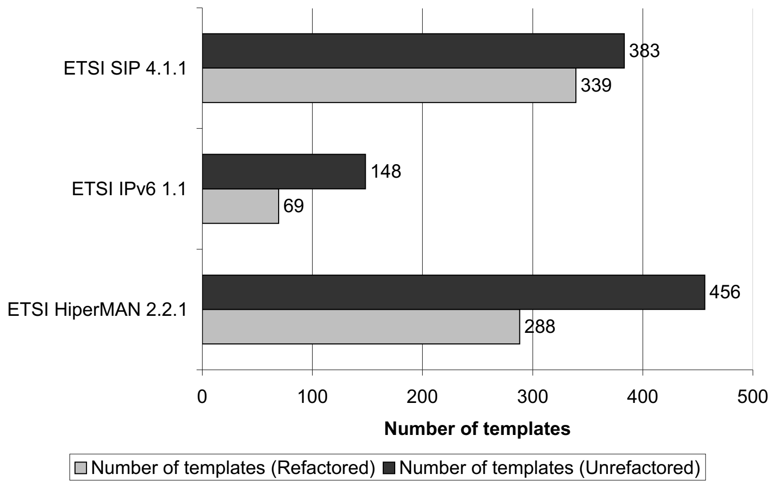


Figure 7. Number of templates before and after applying the refactorings

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 parametrisation, the analysability quality subcharacteristic may also benefit.

The charts in figures 6 and 7 visualise 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 optimised 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 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 the 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 about 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 the quality of a 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, 34, 35] deal with transformations which 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 the concepts, which are available in both test languages to specialise, parametrise, 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 which only contain 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 for 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 which interpret metric values to detect issues and to make suggestions for appropriate refactorings.

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 standardised 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 freely available at its website [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

1. 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.
2. Danet TTCN-3 Toolbox. `http://www.danet.com/index.php?id=ttcn-3-toolbox&L=6`. [18 September 2007].
3. Elvior MessageMagic. `http://messagemagic.elvior.ee`. [18 September 2007].
4. Métodos y Tecnología ExhaustiF/TTCN. `http://www.mtp.es/productos.php?id=1`. [18 September 2007].

5. OpenTTCN Oy OpenTTCN Tester for TTCN-3. `http://www.openttcn.com/Sections/Products/OpenTTCN3`. [18 September 2007].
6. Telelogic Tau/Tester. `http://www.telelogic.com/corp/products/tau/tester/index.cfm`. [18 September 2007].
7. Testing Technologies TTworkbench. `http://www.testingtech.de/products/ttwb_intro.php`. [18 September 2007].
8. Baker P, Loh S, Weil F. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. *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, Briand L, Williams C (eds.), Springer: Berlin, 2005; 476–491. DOI: 10.1007/11557432_36.
9. Grabowski J, Hogrefe D, Réthy G, Schieferdecker I, Wiles A, Willcock C. An introduction to the testing and test control notation (TTCN-3). *Computer Networks* Jun 2003; **42**(3):375–403. DOI: 10.1016/S1389-1286(03)00249-4.
10. Willcock C, Deiß T, Tobies S, Keil S, Engler F, Schulz S. *An Introduction to TTCN-3*. Wiley: New York, 2005.
11. TTCN-3 Website. `http://www.ttcn-3.org`. [18 September 2007].
12. Fenton NE, Pfleeger SL. *Software Metrics*. PWS Publishing Company: Boston, 1997.
13. Fowler M. *Refactoring – Improving the Design of Existing Code*. Addison-Wesley: Boston, 1999.
14. Bisanz M. Pattern-based Smell Detection in TTCN-3 Test Suites. Master's Thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-44 Dec 2006.
15. Neukirchen H, Bisanz M. Utilising Code Smells to Detect Quality Problems in TTCN-3 Test Suites. *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, Petrenko A, Veanes M, Tretmans J, Grieskamp W (eds.), Springer: Berlin, 2007; 228–243. DOI: 10.1007/978-3-540-73066-8_16.
16. 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.
17. 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. *Proceedings of Software Engineering 2007 (SE 2007)*, *Lecture Notes in Informatics (LNI)*, vol. 105, Bleeck WG, Rasch J, Züllighoven H (eds.), Gesellschaft für Informatik, Köllen Verlag: Bonn, 2007; 231–242.
18. Zeiss B, Neukirchen H, Grabowski J, Evans D, Baker P. Refactoring and Metrics for TTCN-3 Test Suites. *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, Gotzhein R, Reed R (eds.), Springer: Berlin, 2006; 148–165. DOI: 10.1007/11951148_10.
19. Basili VR, Weiss DM. A Methodology for Collecting Valid Software Engineering Data. *IEEE Transactions on Software Engineering* 1984; **10**(6):728–738.
20. McCabe T. A Complexity Measure. *IEEE Transactions on Software Engineering* 1976; **2**(4):308–320.
21. Parnas D. Software Aging. *Proceedings of the 16th International Conference on Software Engineering (ICSE), May 16-21, 1994, Sorrento, Italy*, IEEE Computer Society/ACM Press, 1994; 279–287.
22. Opdyke W. Refactoring Object-Oriented Frameworks. PhD Thesis, University of Illinois at Urbana-Champaign, USA 1992. URL `citeseer.ist.psu.edu/article/opdyke92refactoring.html`.
23. TRex Website. `http://www.trex.informatik.uni-goettingen.de`. [18 September 2007].
24. Zeiss B. A Refactoring Tool for TTCN-3. Master's Thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-05 Mar 2006.
25. Eclipse Foundation. Eclipse. `http://www.eclipse.org`. [18 September 2007].
26. Parr T. ANTLR parser generator v2. `http://www.antlr2.org`. [18 September 2007].
27. 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 Jul 2006.
28. 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 Apr 2006.
29. ETSI. Technial 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 Apr 2006.
30. Sneed HM. Measuring the Effectiveness of Software Testing. *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, Beydeda S, Gruhn V, Mayer J, Reussner R, Schweiggert F (eds.), Gesellschaft für Informatik, Köllen Verlag: Bonn, 2004.

31. Vega DE, Schieferdecker I. Towards Quality of TTCN-3 Tests. *Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling, May 31–June 2, 2006, University of Kaiserslautern, Germany*, Gotzhein R (ed.), 2006; 65–77.
32. ETSI. Technical Report (TR) 101 666 (1999-05): Information technology – Open Systems Interconnection Conformance testing methodology and framework; The Tree and Tabular Combined Notation (TTCN) (Ed. 2++). European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France 1999.
33. Deiß T. Refactoring and Converting a TTCN-2 Test Suite. Presentation at the TTCN-3 User Conference 2005, June 6-8, 2005, Sophia-Antipolis, France. `http://www.ttcn-3.org/TTCN3UC2005/program/Wednesday 8th June/03- Session IV- Transitioning, Test system deployment/Deiss_Conversion.pdf` May 2005. [18 September 2007].
34. Schmitt M. Automatic Test Generation Based on Formal Specifications – Practical Procedures for Efficient State Space Exploration and Improved Representation of Test Cases. PhD Thesis, University of Göttingen, Germany Apr 2003.
35. Wu-Hen-Chang A, Viet DL, Batori G, Gecse R, Csopaki G. High-Level Restructuring of TTCN-3 Test Data. *Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers*, *Lecture Notes in Computer Science (LNCS)*, vol. 3395, Grabowski J, Nielsen B (eds.), Springer: Berlin, 2005; 180–194. DOI: 10.1007/b106767.
36. OMG. UML Testing Profile (Version 1.0 formal/05-07-07). Object Management Group (OMG) Jul 2005.