Refactoring for TTCN-3 Test Suites

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

[1] Software Engineering for Distributed Systems Group, Institute for Informatics, University of Göttingen, Lotzestr. 16-18, D-37083 Göttingen, Germany
`{zeiss,neukirchen,grabowski}@cs.uni-goettingen.de`

[2] Motorola Labs, Jays Close, Viables Industrial Estate, Basingstoke, RG22 4PD, UK
`{vnsd001,Paul.Baker}@motorola.com`

Abstract. Experience with the development and maintenance of test suites has shown that the *Testing and Test Control Notation* (TTCN-3) provides very good concepts for adequate test specification. However, experience has also demonstrated that during either the migration of legacy test suites to TTCN-3, or the development of large TTCN-3 test specifications, users have found it is difficult to construct TTCN-3 tests that are concise with respect to readability, usability, and maintainability. To address these issues, this paper investigates refactoring for TTCN-3; systematically restructuring a test suite without changing its behaviour. Complementary metrics are suggested to assess the readability and maintainability of TTCN-3 test suites. For automation, a tool called TRex has been developed that supports refactoring and metrics for TTCN-3.

1 Introduction

The maintenance and migration of legacy test suites is an important issue for industry. For example, within Motorola test suites developed with a high coupling between value and behaviour specification can lead to a large maintenance burden [1]. A single change to a data type can result in the need to change many tests. The *Testing and Test Control Notation* (TTCN-3) [2, 3] contains concepts that can alleviate such issues, such as *templates*. However, experience has demonstrated that it is not always obvious how to use such concepts in a manner that can maximise the readability, usability, and maintainability of TTCN-3. In addition, Motorola teams have encountered problems migrating their test suites to TTCN-3. In doing so, they develop tools that perform simple translations of legacy test suites to TTCN-3. This can often result in non-optimal TTCN-3 code. 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 then leads to another maintenance burden.

To this end, Motorola has collaborated with the University of Göttingen to develop a tool, called *TRex*, for assessing attributes and subsequent restructuring of a TTCN-3 test suite. The current aims for TRex are 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 to avoid any issues, and (3) a means for restructuring TTCN-3 test suites to improve them with respect to any existing issues. The actual restructuring is performed by applying *refactorings*. For software development, refactoring [4] is a proven means to restructure software with the aim of improving its quality. We suggest to apply refactoring also to TTCN-3 test suites.

This paper is structured as follows: In the next chapter, foundations on refactoring and a survey on related work are presented. Chapter 3 contains the main contribution of this paper; our catalogue of 49 refactorings for TTCN-3. In Chapter 4 we give an overview of our activities into automating the application of these refactorings using our TRex tool; making an assessment of TTCN-3 test suites based on metrics we have defined and employing a rule based approach to derive applicable refactorings. Finally, we conclude with a summary and outlook.

2 Foundations

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*" [5]. This means refactoring is a remedy against software ageing [6]. 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. Due to the simplicity of the steps, the effects of the changes are predictable. Sometimes, steps even appear to be awkward, but in fact such steps help to figure out the consequences of a refactoring as soon as possible and maintain the correctness of software not only before and after, but even within a refactoring.

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 to convert 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 the evolutionary software development community around *Smalltalk*, the first detailed written work on refactoring was the PhD thesis of Opdyke [7] who treats refactoring of *C++* source code. Refactoring has finally been popularised by Fowler and his book "*Refactoring*" [5] which contains a catalogue of 72 refactorings which are applicable to *Java* source code.

2.1 Related Work

Existing work on refactoring deals mainly with the refactoring of source code and little is known on the refactoring of test specifications. Probably the most frequent refactoring of tests occurs in agile software development processes. For example, the *Extreme Programming* approach [8], where the implementation and the unit test suite, which is realised using the same programming language as the implementation (e.g. the *JUnit* framework [9] for unit testing Java implementations), are both subject of refactoring. However, only one publication is known which treats refactoring of unit tests on their own: van Deursen et al. [10] suggest to automate also the creation of external resources, to check equality of two Java objects not by comparing the results of their `toString()` methods, but to implement and use the more robust `equals()` method instead, and to provide an explanatory message when a test fails. While the latter refactoring is also applicable to TTCN-3, the other refactorings are specific to unit testing which is not the primary target of TTCN-3.

Concerning TTCN-3 and its predecessor, the *Tree and Tabular Combined Notation* (TTCN-2) [11], three publications [12–14] deal with transformations which can be regarded as refactoring. Schmitt [12] and Wu-Hen-Chang et al. [13] 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 [14] 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.

2.2 Validating the Equivalence of Tests

Opdyke [7] and Fowler [5] 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 which is applicable if no such tool exists. Each entry in his refactoring catalogue provides so called *mechanics*: concise, systematic step-by-step instructions for humans of how to carry out the refactoring. To validate that refactoring did not change the observable behaviour, Fowler presumes that an adequate suite of automated tests exists. If the implementation passes that test suite before and after the refactoring, it is assumed that its observable behaviour was not affected by the refactoring.

When refactoring tests manually, van Deursen et al. [10] suggest running the test suite which is subject of a refactoring against the same implementation before and after the refactoring; checking that the same verdict is returned in both cases. However, this is not sufficient since not all paths of the test suite may be executed. Instead, *bisimulation* [15] of both the original and refactored test suites is required to validate their equivalence, i.e. that they yield the same verdict for the same behaviours of an implementation.

3 A Refactoring Catalogue for TTCN-3

The presentation of our refactorings for TTCN-3 is inspired by Fowler's refactoring catalogue for Java [5]. Hence, we use the same fixed format for describing our refactorings: each refactoring is 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. In that section, we use the term "source" to refer to the code which is addressed by a refactoring and thus usually removed or simplified and the term "target" to refer to code which is created as a result of a refactoring. The example section illustrates the refactoring by showing TTCN-3 core notation excerpts before and after the refactoring is applied.

The mechanics sections provided in this refactoring catalogue can be exploited in two ways: the refactorings can be applied manually or automated by building a tool based on the experience distilled in the step-by-step instructions. Since manual refactoring is error prone, the mechanics also contain the "compile" and "validate" instructions. The compile step is used to check whether syntax and static semantics of the test case are still valid. The validate step means to start the bisimulation process to validate that the original and refactored test suite still behave equivalently. To detect possible mistakes during refactoring as soon as possible, compile and validate steps are suggested as soon and as often as they are applicable. As discussed in Section 2.2, we suggest to automate the application of refactorings using our TRex tool which is described in Section 4.

To ease usage of our refactoring catalogue, we have divided our refactorings 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 3.1 and 3.2.

3.1 Language Independent Refactorings Applicable to TTCN-3

We investigated which of the 72 refactorings from Fowler [5] are also relevant for TTCN-3. Even though these refactorings were intended for Java, some of them are language independent or can be reinterpreted in a way that they are applicable to TTCN-3. 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*, *timer*, and *ports* local to a component respectively. Furthermore, whenever Fowler's mechanics instruct to "test" the refactored implementation, the refactored test suite needs to be validated.

Under these circumstances, we found that 28 refactorings are applicable to TTCN-3. Where necessary, we have changed the name of these refactorings to reflect their reinterpretation for TTCN-3. In this case, the original name used by Fowler is given in square brackets. The list of these refactorings is as follows:

Refactorings for Test Behaviour

- *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*.

Refactorings for Improving the Overall Structure of a Test Suite

- *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*][3],
- *Replace Parameter with Explicit Functions* [*Replace Parameter with Explicit Methods*],
- *Replace Parameter with Function* [*Replace Parameter with Method*].

[3] Note that while Fowler refers only to renaming a method, not only the corresponding TTCN-3 constructs **testcase** and **function** qualify for renaming, but also variables, types, templates, constants, ports, timer, components, modules, groups and altsteps are reasonable subjects of the *Rename* refactoring.

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 differs significantly when applied to templates, we have considered them as TTCN-3 specific refactorings and describe them in the next section.

3.2 TTCN-3 Specific Refactorings

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, our refactorings take advantage of TTCN-3 altsteps, templates, grouping, modules and importing from modules, components, restricted sub-types, logging, and creating concurrent test cases.

Those refactorings which refer to templates and to adding an explanatory log message include some of the known transformations surveyed in Section 2.1. However, we go beyond the existing work by being more extensive and by providing for each refactoring detailed step-by-step instructions and examples for their application.

Until now, we identified 21 TTCN-3 specific refactorings. 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 testcase 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.
- *Parametrise Template:* Several templates of the same type, which merely use different field values, are replaced by a single parametrised 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.

In the following, we will focus on refactorings for data descriptions, since most of the maintenance problems at Motorola were related to the use of templates. To give an impression of how our TTCN-3 refactoring catalogue looks, we present two refactorings in detail: *Inline Template Parameter* and *Parametrise Template*. Please refer to our complete TTCN-3 refactoring catalogue [16] for a detailed description of all refactorings.

3.2.1 *Parametrise Template*

Summary: Several templates of the same type, which merely use 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 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 maintainability and increases flexibility. 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:
- Create the parametrised target template signature. It is of the same type as the source templates. Introduce a parameter for each field in which the source template values differ. The target template declaration's name should reflect the meaning of the non-parametrised values.
- Copy one source template body to the parametrised target template declaration and replace the varying parts with their newly introduced template parameters.
- Compile.
- Repeat the following steps for all references to the source template declarations:
    - Replace the source template reference with a reference to the parametrised target template. As parameter values, use the field values from the originally referenced template declaration corresponding to the parametrised values in the target template.
    - Compile and validate.
- Remove the source template declarations from the code. They should not be referenced anymore.
- Compile and validate.

Example: Listing 1.1 shows the unrefactored example. The source template declarations firstTemplate (lines 6–9) and secondTemplate (lines 11–14) differ only in the values of ipAddress.

Listing 1.1. Parametrise Template (Unrefactored)

```
 1 type record ExampleType {
 2   boolean ipv6,
 3   charstring ipAddress
 4 }
 5
 6 template ExampleType firstTemplate := {
 7   ipv6 := false,
 8   ipAddress := "127.0.0.1"
 9 }
10
11 template ExampleType secondTemplate := {
12   ipv6 := false,
13   ipAddress := "134.72.13.2"
14 }
15
16 testcase exampleTestCase() runs on ExampleComponent {
17   pt.send( firstTemplate );
18   pt.receive( secondTemplate );
19 }
```

Listing 1.2. Parametrise Template (Refactored)

```
 1 type record ExampleType {
 2   boolean ipv6,
 3   charstring ipAddress
 4 }
 5
 6 template ExampleType parametrisedTemplate( charstring addressParameter ) := {
 7   ipv6 := false,
 8   ipAddress := addressParameter
 9 }
10
11 testcase exampleTestCase() runs on ExampleComponent {
12   pt.send( parametrisedTemplate( "127.0.0.1" ) );
13   pt.receive( parametrisedTemplate( "134.72.13.2" ) );
14 }
```

The resulting code after applying *Parametrise Template* is shown in Listing 1.2. 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 IP addresses as parameters.

3.2.2 *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: he parametrises templates as he thinks it might be useful, 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 values is the source parameter. Record the common value.
  - If you have more than one common actual parameter value in all references, it is easier to inline them together. Therefore, perform each step that concerns the source parameters for each source parameter at once.

– Copy the source template declaration and give the copied declaration a temporary name. It is the target template declaration.

– In the target template declaration body, replace each reference to the source parameter with the value noted in the first step. In the target template declaration signature, remove the parameter corresponding to the source parameter.
– Compile.
– 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.
– Compile and validate.
– Consider usage of the *Rename* refactoring to improve the target template declaration name.

Example: Listing 1.3, 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.

Listing 1.3. Inline Template Parameter (Unrefactored)

```
 1  type record ExampleType {
 2    boolean ipv6,
 3    charstring ipAddress
 4  }
 5
 6  template ExampleType exampleTemplate( charstring addressParameter ) := {
 7    ipv6 := false,
 8    ipAddress := addressParameter
 9  }
10
11  testcase exampleTestCase() runs on ExampleComponent {
12    pt.send( exampleTemplate( "127.0.0.1" ) );
13    pt.receive( exampleTemplate( "127.0.0.1" ) );
14  }
```

After applying the *Inline Template Parameter* refactoring (Listing 1.4), the string value "127.0.0.1" is inlined into the template body of exampleTemplate (Line 8), 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.

Listing 1.4. Inline Template Parameter (Refactored)

```
 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    pt.send( exampleTemplate );
13    pt.receive( exampleTemplate );
14  }
```

4 Automation of TTCN-3 Refactoring

In the following we describe how the restructuring of TTCN-3 test suites can be automated. To locate inappropriate usage of TTCN-3 we use so called *bad smells* or *code smells*, a kind of anti-pattern [17]. Examples for code smells include duplicated code, overly long testcases, or templates which are never referenced. Some of them can only be detected by pattern recognition, but some of them may also be detected by calculating metrics. We have started with a metrics-based approach (Section 4.1) which is also suitable for a general assessment of TTCN-3 test suites. Based on these metrics we provide rules of when to apply which refactoring. Our TRex tool (Section 4.2) calculates these metrics, applies the rules to suggest appropriate refactorings, and automatically performs the individual steps of a refactoring.

4.1 TTCN-3 Metrics

Since quantitative methods like metrics have proved to be a powerful means to control processes in the other sciences, computer science practitioners and theoreticians introduced similar approaches into software development. A software metric is a measure of some property of a piece of software or its specifications.

Software metrics can be structured into *linguistic*, *structural*, and *hybrid* metrics. Linguistic metrics measure properties of the usage of a programming or specification language. Well-known linguistic metrics are the Halstead metrics [18]. Examples of the Halstead metrics are *number of operators*, *number of operands*, *program volume*, or *program level*. Structural metrics analyse the structure of a program or specification. The most popular examples of structural metrics are the McCabe metrics [19]. They are based on the control flow graph of a program and measure properties of this graph, such as the *cyclomatic number*. Hybrid metrics combine linguistic and structural metrics.

Halstead and the McCabe metrics are mainly developed for procedural programs, but there also exist metrics for more modern program paradigms like object oriented programs. A popular example of such metrics is the Chidamber & Kemerer metrics suite [20] which measures properties like *depth of inheritance tree*, *coupling between objects*, or *lack of cohesion in methods*.

We started to investigate metrics to measure the quality of TTCN-3 test suites. For this, we want to use (and possibly adapt) the well-known metrics mentioned above, but also define new TTCN-3 specific metrics. In a first step, we implemented some basic linguistic metrics in the TRex tool. These are:

- *Number of non-comment lines of TTCN-3 source code.*
- *Number of test cases*, including *Number of references to each test case.*
- *Number of functions*, including *Number of references to each function.*
- *Number of altsteps*, including *Number of references to each altstep.*
- *Number of port types*, including *Number of references to each port type.*
- *Number of component types*, including *Number of references to each component type.*

– *Number of data type definitions*, including *Number of references to each data type*.
– *Number of template definitions*, including *Number of references to each template* and *Number of parametrised templates*.
– *Template coupling*, which will be computed as follows:

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

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

$$\mathit{score}(\mathit{stmt}(i)) := \begin{cases} 1, & \text{if } \mathit{stmt}(i) \text{ references a template without parameters,} \\ & \text{e.g. } \mathsf{MyPort.}\mathbf{send}\mathsf{(MyTemplateRef)} \\ & \text{or uses wildcards only, e.g. } \mathsf{MyPort.}\mathbf{send}\mathsf{(MyType:?)} \\ 2, & \text{if } \mathit{stmt}(i) \text{ references a template with parameters,} \\ & \text{e.g. } \mathsf{MyPort.}\mathbf{send}\mathsf{(MyTemplateRef(1, "a"))} \\ 3, & \text{if } \mathit{stmt}(i) \text{ uses an inline template,} \\ & \text{e.g. } \mathsf{MyPort.}\mathbf{receive}\mathsf{(MyType:\{n:=1, s:="a"\})} \end{cases}$$

Template coupling measures the dependence of test behaviour and test data in the form of 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 maintainability a decoupling of test data and test behaviour (i.e. the template coupling score is close to 1) might be advantageous and for optimal readability most templates may be inline templates (i.e. the template coupling score will be close to 3).

With appropriate boundary values for the different metrics, we want to identify places in TTCN-3 specifications which need refactoring. At the moment we have a rough idea of suitable values and have started to analyse real-world test suites to further improve our estimates. Some metrics may even allow an entirely automatic refactoring to take place.

We have found some rules that obviously help to improve the quality of TTCN-3 test suites with respect to template definitions. Most of these rules can be directly related to metrics and refactorings:

Rule 1: A template definition which is not referenced (Metric value: *Number of References to the Template = 0*) should be removed.
Rule 2: A template definition which is only referenced once (Metric value: *Number of References to the 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: If a user wants to achieve “optimal readability” (i.e. maximise the *Template Coupling Score*), a template definition which is referenced multiple times (Metric value: *Number of References to the Template* > 1) should be inlined and its definition should be removed (Application of *Inline Template* refactoring).

Rule 4: If a user wants to achieve “good maintainability” (i.e. a *Template Coupling Score* close to 1, a template definition without parameters which is referenced multiple times (Metric value: *Number of References to the Template* > 1) should not be altered.

Rule 5: 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 6: Unused parameters of a template definition (e.g. parameters which are not used in assignments) should be removed altering the template definition (Application of *Remove Parameter* refactoring).

Rule 7: For a template definition which is referenced multiple times and which has formal parameters that do not adhere to Rules 5 or 6 the following rules apply:

(a) If all instantiations of a template are the same, i.e. all formal parameters are given the same values, 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 and the user wants “optimal readability”, then the template shall be inlined and its definition be removed (Application of *Inline Template* refactoring).

Rule 8: If the user aims for “good maintainability” and 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 we currently do nothing as the user would have to choose which field to parametrise upon.

The rules presented above can only give an impression of how metrics can steer the refactoring process. We are currently refining the rules and defining new rules for the refactoring of test behaviour and the TTCN-3 module structure. This includes the definition of further metrics to underpin the rules, analysis of the influence of the rule ordering, and the investigation of options such as “good maintainability or “optimal readability” which are informally mentioned above. (E.g. using inline templates optimises readability only up to a certain size of template, or the fact that parametrised templates promote reuse, but not necessarily maintainability or readability.)

4.2 Tool Support

We have implemented a first version of *TRex*, the *TTCN-3 Refactoring and Metrics* tool. Based on the rules defined in the previous section, refactorings are suggested automatically by TRex and the user is given the option to apply them to one or more template or reference. Otherwise the user needs to identify the places where a refactoring is to be applied. In some cases, additional information needs to be provided, e.g. the desired new name for the *Rename* refactoring. In any case, all further steps are then performed automatically. This significantly reduces the risk of changing the behaviour of a test suite. Automated refactoring has been successfully applied to source code of implementation languages, e.g. using the Java Development Tools of the Eclipse platform [21].

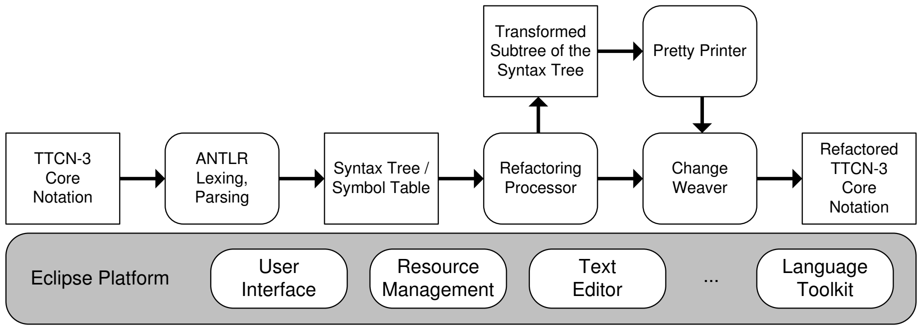

Fig. 1. A Simplified View of Our TTCN-3 Refactoring Tool

The TRex tool analyses data flow and inspects declarations, references, and scopes of TTCN-3 language constructs. As shown in Figure 1, TRex is implemented as a plug-in for the Eclipse platform which provides infrastructure for user interfaces, handling of workspace resources, text editing, and a language toolkit for basic language independent support of semantic preserving workspace transformations. For building up the syntax tree for a test suite we use *'ANother Tool for Language Recognition'* (ANTLR) [22], a parser generator which supports lexing, parsing, and syntax tree creation and traversal. The actual refactoring is performed on the basis of the syntax tree[4], the symbol table, and the TTCN-3 core notation. A refactoring processor calculates the changes necessary for transforming the source code. This can be done directly on the source code based on

[4] An alternative approach would be to build up a TTCN-3 meta model [23] representation of a TTCN-3 test suite and to use this representation instead of the syntax tree.

the information obtained from the syntax tree and the symbol table, otherwise an intermediate subtree transformation step is necessary. In this step, one or more syntax subtrees are transformed and the corresponding core notation is obtained by a TTCN-3 pretty printer. These changes are weaved into the original TTCN-3 core notation using a programmatic text editor (which is provided by the Eclipse platform). The original formatting is therefore mostly preserved.

The Metrics are built up, based on the list in Section 4.1, by traversing the syntax tree and counting the number of basic elements and their references, whilst also processing each communication statement to generate the Template Coupling score. During this traversal we also apply the rules for detecting suitable areas for refactorisation to every template. From this we generate a table for the 'Problems' view of TRex, listing each detection as a warning, with one or more appropriate refactorings supplied as 'Quick Fix' options for the user to apply automatically.

5 Summary and Outlook

We presented a catalogue of 49 refactorings which can be used to restructure existing TTCN-3 test suites without changing their observable behaviour. The aim of our refactorings is to improve readability, extensibility, modularity, reusability, complexity, maintainability, and efficiency of test suites. Each of our refactorings provides detailed step-by-step instructions on how to perform the actual transformations and is accompanied by TTCN-3 examples which illustrate their application. In this paper, we gave an overview of our TTCN-3 refactoring catalogue and presented some examples from the full version [16]. Furthermore, we outlined an initial set of metrics to assess the quality of test suites and described how they can be used to automate the refactoring process.

We implemented the TRex tool, which calculates metrics, makes suggestions for applying refactorings, and automatically performs the specific steps of a refactoring. TRex is already proving to be a very useful environment for the editing, assessment, and restructuring of TTCN-3 test suites.

Currently we are working on a case study to obtain boundary values for our metrics and to demonstrate the benefits of our refactorings on a large scale. Future research aims at extending our metrics and rules of when to apply which refactoring. Furthermore, we will study pattern-based code smells in addition to our metrics-based approach. We also plan to implement tool support for the validation of manual refactoring by providing a TTCN-3 bisimulation tool which allows the equivalence of TTCN-3 test suites to be checked. Finally, we intend to release TRex as open source software. TRex will be publicly available at `http://www.trex.informatik.uni-goettingen.de`.

References

1. Baker, P., Loh, S., Weil, F.: Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In Briand, L.C., Williams, C., eds.: MoDELS. Volume 3713 of Lecture Notes in Computer Science (LNCS)., Springer (2005) 476–491

2. ETSI: European Standard (ES) 201 873-1 V3.1.1 (2005-06): The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France (2005)
3. Grabowski, J., Hogrefe, D., Réthy, G., Schieferdecker, I., Wiles, A., Willcock, C.: An Introduction into the Testing and Test Control Notation (TTCN-3). Computer Networks **42**(3) (2003)
4. Mens, T., Tourwe, T.: A Survey of Software Refactoring. IEEE Transactions on Software Engineering **30**(2) (2004) 126–139
5. Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley (1999)
6. Parnas, D.L.: Software Aging. In: Proceedings of the 16th International Conference on Software Engineering (ICSE), May 16-21, 1994, Sorrento, Italy., IEEE Computer Society/ACM Press (1994) 279–287
7. Opdyke, W.F.: Refactoring Object-Oriented Frameworks. PhD thesis, University of Illinois at Urbana-Champaign, USA (1992)
8. Beck, K.: Extreme Programming Explained. Addison Wesley (2000)
9. Gamma, E., Beck, K.: JUnit. `http://junit.sourceforge.net/` (2006)
10. v. Deursen, A., Moonen, L., v. d. Bergh, A., Kok, G.: Refactoring Test Code. In Marchesi, M., Succi, G., eds.: Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes in Software Engineering. (2001)
11. 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)
12. 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 (2003)
13. Wu-Hen-Chang, A., Viet, D.L., Batori, G., Gecse, R., Csopaki, G.: High-Level Restructuring of TTCN-3 Test Data. In Grabowski, J., Nielsen, B., eds.: Formal Approaches to Software Testing: 4th International Workshop, FATES 2004, Linz, Austria, September 21, 2004, Revised Selected Papers. Volume 3395 of Lecture Notes in Computer Science (LNCS)., Springer (2005) 180–194
14. 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 (2005)
15. Milner, R.: A Calculus of Communicating Systems. Volume 92 of Lecture Notes in Computer Science (LNCS). Springer (1980)
16. Zeiss, B.: A Refactoring Tool for TTCN-3. Master's thesis, Institute for Informatics, University of Göttingen, Germany, ZFI-BM-2006-05 (2006)
17. Brown, W.J., Malveau, R.C., McCormick, H.: Anti-Patterns. Wiley (1998)
18. Halstead, M.H.: Elements of Software Science. Elsevier, New York (1977)
19. McCabe, T.: A Complexity Measure. IEEE Transactions of Software Engineering **2**(4) (1976) 308–320
20. Chidamber, S.R., Kemerer, C.: A Metric Suite for Object-Oriented Design. IEEE Transactions of Software Engineering **20**(6) (1994) 476–493
21. Eclipse Foundation: Eclipse. `http://www.eclipse.org` (2006)
22. Parr, T.: ANTLR parser generator. `http://www.antlr.org` (2006)
23. Schieferdecker, I., Din, G.: A Meta-model for TTCN-3. In Núñez, M., Maamar, Z., Pelayo, F.L., Pousttchi, K., Rubio, F., eds.: Applying Formal Methods: Testing, Performance and M/ECommerce, FORTE 2004 Workshops, Toledo, Spain, October 1-2, 2004. Volume 3236 of Lecture Notes in Computer Science (LNCS)., Springer (2004) 366–379