TRex – An Open-Source Tool for Quality Assurance of TTCN-3 Test Suites

Benjamin Zeiss[1]

Helmut Neukirchen[1]

Jens Grabowski[1]

Dominic Evans[2]

Paul Baker[2]

[1]Software Engineering for Distributed Systems Group,
Institute for Informatics, University of Göttingen,
Lotzestr. 16-18, 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

The comprehensive test of modern communication systems leads to large and complex test suites which have to be maintained throughout the system life-cycle. Experience with those written in the standardised Testing and Test Control Notation (TTCN-3) has shown that the maintenance of test suites is a non-trivial task and its burden can be reduced with appropriate tool support. To this aim, we have developed the TRex tool, published as open-source under the Eclipse Public License, which supports the assessment and automatic restructuring of TTCN-3 test suites by providing suitable metrics and refactorings. Besides presenting TRex and its functionality, the main contribution of this paper is the discussion of complexity metrics for TTCN-3 test suites.

1 Introduction

The Testing and Test Control Notation (TTCN-3) [ETS05, GHR+03] is a test specification and test implementation language standardised by the European

Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU). While TTCN-3 has its roots in functional black-box testing of telecommunication systems, it is nowadays also used in other domains such as Internet protocols, automotive, aerospace, or service-oriented architectures. TTCN-3 can be used not only for specifying and implementing functional tests, but also for scalability, robustness or stress tests. TTCN-3 is based on a textual core notation and several presentation formats.

Commercial TTCN-3 tools [Ope06, Tel06, Tes06] and in-house solutions support editing test suites and compiling them into executable code. By implementing the interfaces of the standardised TTCN-3 runtime environment, these tools also allow TTCN-3 test campaigns to be managed and executed.

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 which requires tool support [BLW05]. 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.

Currently, no tools for assessing and improving the quality of TTCN-3 test suites exist. To this end, Motorola has collaborated with the University of Göttingen to develop a TTCN-3 refactoring and metrics tool, called TRex. 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 to avoid any issues, and (3) a means for restructuring TTCN-3 test suites to improve them with respect to any existing issues. To let others participate in our tool and to participate in contributions from others, we have made TRex a general open-source quality assurance and quality improvement tool for TTCN-3 test suites.

This paper is structured as follows: In Section 2, we will give a short overview of TRex's functionality and its refactoring capabilities. Then, in Section 3, we will present the usage of metrics to assess TTCN-3 test suites, in particular our latest results on the applicability of complexity metrics. Finally, we conclude with a summary and outlook.

2 Overview of the TRex tool

TRex is implemented[1] as an Eclipse plug-in and therefore, anyone who has experience with the Eclipse Platform [Ecl06], e.g. by using the popular Java Development Tools (JDT), will immediately feel comfortable with TRex. The TTCN-3 perspective of TRex 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

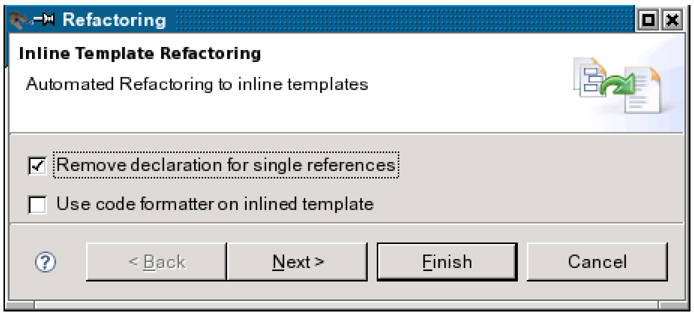
[1] A description of the implementation of TRex is available in a previous paper [BEG$^+$06].

with syntax highlighting and checking according to the latest 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, 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, the Telelogic Tau G2/Tester [Tel06] analyser and compiler *t3cg* is integrated into the TRex environment.

As a powerful means for improving the quality of TTCN-3 test suites, TRex is able to restructure test suites in an automated way. This is achieved by using refactoring which 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"* [Fow99]. While refactoring is well known for implementation languages like Java, it has not been systematically studied for TTCN-3. Thus, we developed a catalogue of 49 refactorings applicable for TTCN-3 [Zei06, ZNG+06]. In TRex, we have begun implementing those refactorings which we believe would improve the maintainability of Motorola's test suites and have so far completed the *Inline Template*, *Inline Template Parameter*, *Merge Templates*, and *Rename* refactorings. As an example, we describe the *Inline Template* refactoring in detail.

For specifying test data, TTCN-3 uses *templates*. A template may either be defined as a named entity or "on-the-fly" using an inline template notation. The first way promotes re-use since a template definition may be referenced at several locations. In contrast, test behaviour may be easier to understand if it uses inline templates, since inline templates define test data at the same location where it is actually used for sending or receiving.

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. 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 1, it is possible to remove the declaration of a template if it was referenced only once and the Formatter may additionally be used to obtain a pretty-printed template. Before a refactoring is


Refactoring

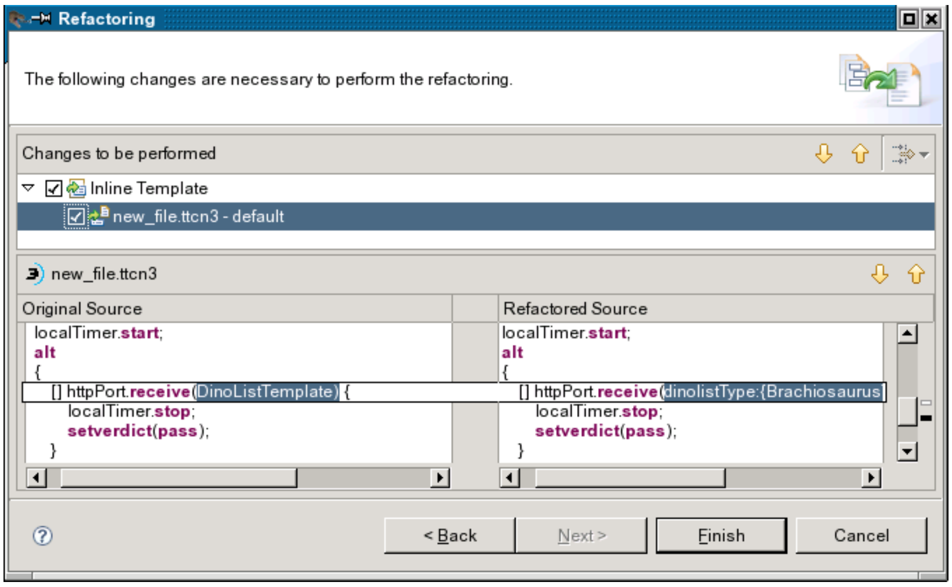
Inline Template Refactoring

Automated Refactoring to inline templates

☑ Remove declaration for single references

☐ Use code formatter on inlined template

Fig. 1: Wizard for the configuration of the Inline Template refactoring



The following changes are necessary to perform the refactoring.

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);
  }
```

Fig. 2: Wizard for the preview of the Inline Template refactoring

actually applied, the refactoring wizard displays a preview of all resulting changes (Figure 2).

These refactorings are typically semi-automated, since the user 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 using metrics.

3 Metrics

Metrics are a powerful means to assess software and to control software development. According to Fenton et al. [FP97], 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.

Internal product metrics can be structured into *size* and *structural* metrics. Size metrics 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. In the remainder, we consider only internal product metrics for TTCN-3 test suites.

Most of the known metrics related to tests concern processes and resources, but not products, i.e. test suites. From those known metrics which relate to tests as product, most are simple size metrics: Vega et al. [VSD06], for example, propose several internal and external size metrics for TTCN-3 test suites. Only Sneed [Sne04] is known to include metrics for measuring the complexity of testcases as well as the quality of testcases. However, Sneed's complexity and test quality metrics use only size metrics as input, i.e. they cannot be regarded as structural metrics.

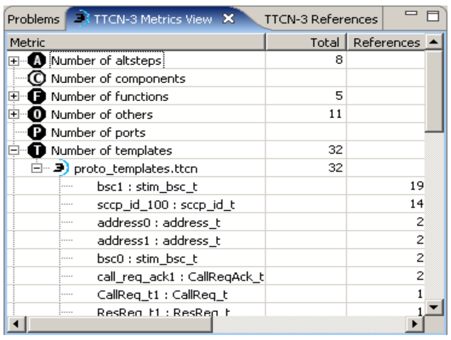
Concerning metrics for measuring complexity of control structures, the most prominent complexity metric is the *Cyclomatic Complexity* from McCabe [McC76, WM96]. The Cyclomatic Complexity $v(G)$ of a control flow graph G can be defined[2] as: $v(G) = e-n+2$, where e is the number of edges and n is the number of nodes in G. The informal interpretation is that a linear control flow has a complexity of 1 and each branching increases the complexity by 1, i.e. $v(G)$ measures the number of branches. The Cyclomatic Complexity metric is descriptive, enabling the identification of software that is error-prone, hard to read, hard to understand, hard to modify, and hard to maintain. But, it is also prescriptive, helping to control and improve software, for example by identifying complex modules which should be split into several simpler ones [WM96]. McCabe suggests to use a boundary value of 10, i.e. behaviour with a higher Cyclomatic Complexity is considered to be too complex and should thus be avoided or refactored.

3.1 Size metrics

In TRex, we use size metrics to identify locations where it might be beneficial to apply a particular refactoring. Figure 3(a) shows the TTCN-3 Metrics view for basic size metrics (e.g. *Number of ...* and *References to ...* for various definitions). Based on these metrics, we have defined several rules by which the templates of a TTCN-3 test suite can be analysed, e.g. looking at number of references, use of parameters, commonalities, etc. From these, TRex is able to identify problematic code fragments and to suggest suitable refactorings. For example, templates which are never referenced can be removed; templates which are only referenced once can be inlined. These suggestions are displayed in the Problems view as warnings (Figure 3(c)) 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 3(b)). A more detailed description of our

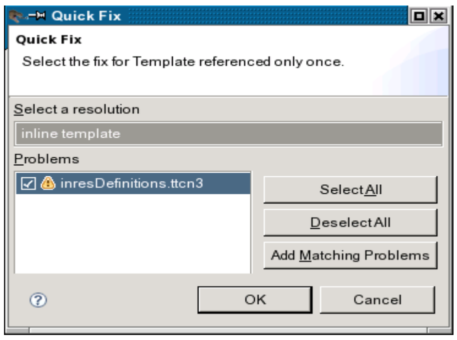
[2] Several ways of defining $v(G)$ can be found in literature. The above definition assumes that G has a single entry and a single exit point. In the presence of several exit points, this assumption can be maintained by adding edges from all exit points to a single exit point.

rules and of the above basic size metrics is available in our previous paper on TTCN-3 refactoring [ZNG+06].



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

Select the fix for Template referenced only once.

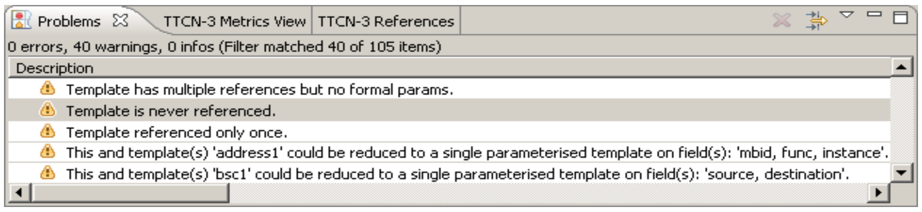
Select a resolution

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

Fig. 3: TRex's size metrics and related functionality

3.2 Structural metrics

To assess the overall quality of TTCN-3 test suites, we investigated several structural metrics and implemented them in TRex.

Coupling metrics

We designed a coupling metric called *Template Coupling* to measure 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 (behaviour statements refer only to template definitions or use wildcards) and 3 (behaviour statements only use inline templates). For good maintainability, a decoupling of test data and test behaviour (i.e. Template Coupling close to 1) is advantageous and for improved readability, most templates shall be inline templates (i.e. Template Coupling close to 3). A more detailed discussion of this metric is provided in a previous paper [ZNG+06].

Complexity metrics

For assessing the control flow complexity of TTCN-3 test suites, we use McCabe's Cyclomatic Complexity v(G). For calculating v(G), TRex builds up the control flow graph G of each behaviour specification (*function*, *testcase*, *altstep*, *control part*). The control flow graph constituents created for statements like *if*, *while*, *for*, etc. are straight forward as suggested by McCabe. However, the *alt* statement (and *altstep* respectively), which allows alternative test behaviour to be described, is unique to TTCN-3. Its semantics are that incoming messages and events are observed until one of the alternative branches matches. Thus, the resulting control flow graph of an *alt* or *altstep* statement resembles the control graph structure of a *select case* construct or *switch case* construct respectively. As a result, each alternative branch increases the Cyclomatic Complexity by 1. Figure 4 provides an example of an *alt* statement and its corresponding control flow graph visualised by TRex in the TTCN-3 Control Flow Graph view using the Eclipse Graphical Editing Framework (GEF) [GEF06].

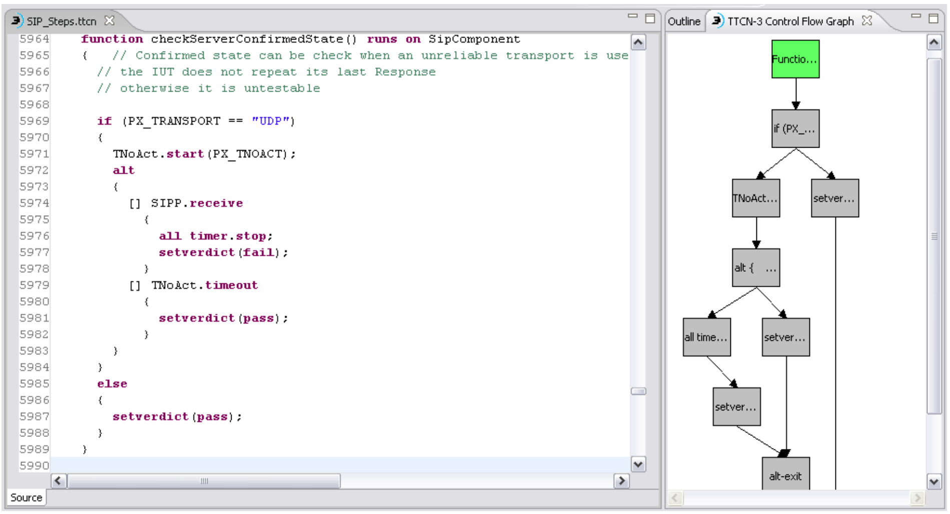
TRex calculates the Cyclomatic Complexity v(G) of each *function*, *testcase*, *altstep*, and *control part*. To allow an overall assessment of the quality of a test suite, we additionally calculate arithmetic averages $\overline{v}$ (G) and their standard deviations σ .

$\overline{v}$ *(All behaviour)*: The average Cyclomatic Complexity over all functions, testcases, altsteps and control parts of a test suite.

$\overline{v}$ *(All behaviour except control part)*: The average Cyclomatic Complexity over all functions, testcases and altsteps.

$\overline{v}$ *(All behaviour except control part and linear behaviour)*: The average Cyclomatic Complexity over all functions, testcases and altsteps that have a Cyclomatic Complexity greater than one.

$\overline{v}$ *(Language elements)*: The average Cyclomatic Complexity over the specified language elements such as functions, testcases, altsteps, or control parts.



```
function checkServerConfirmedState() runs on SipComponent
{   // Confirmed state can be check when an unreliable transport is use
  // the IUT does not repeat its last Response
  // otherwise it is untestable

  if (PX_TRANSPORT == "UDP")
  {
    TNoAct.start(PX_TNOACT);
    alt
    {
      [] SIPP.receive
        {
          all timer.stop;
          setverdict(fail);
        }
      [] TNoAct.timeout
        {
          setverdict(pass);
        }
    }
  }
  else
  {
    setverdict(pass);
  }
}
```

Fig. 4: alt statement and corresponding control flow graph

$\bar{v}$ (Behaviour running on components): The average Cyclomatic Complexity of all functions, testcases and altsteps with a *runs on* clause grouped according to the components on which they run on. The averages can be calculated on different levels: per component and over all components. The considered value in this paper is the latter.

$\bar{v}$ (Testcases including referenced behaviour): The Cyclomatic Complexity of a testcase including all behaviour called or started by a testcase. To be able to calculate this metric, TRex creates the call graph for each testcase. This call graph consists of all functions and altsteps which are directly and indirectly called by a testcase as well as any defaults which are activated from nodes of this call graph. The averages can be calculated on different levels: per testcase and over all testcases. The considered value in this paper is the latter.

Table 1 presents selected measurements applying these metrics to two different test suites: one published by ETSI [ETS04] for the SIP protocol and one which is currently under development at ETSI for the IPv6 protocol. The upper part of Table 1 includes a few basic size metrics to give an impression about the quantities of the measured behaviour.

Table 1: SIP and IPv6 test suite metrics

Metric	**SIP**	$\sigma_{\bar{v}}$	**IPv6**	$\sigma_{\bar{v}}$
Lines of code	42397		46163	
Number of control parts	1		3	
Number of functions	785		643	
Number of functions with runs on	230		616	
Number of functions without runs on	555		27	
Number of testcases	528		295	
Number of altsteps	10		11	
Number of components	2		10	
$\bar{v}$ (All behaviour)	2.43	0.41	2.11	0.29
$\bar{v}$ (All behaviour except control part)	2.01	0.05	1.83	0.06
$\bar{v}$ (All behaviour except control part and linear behaviour)	4.12	0.11	3.44	0.14
$\bar{v}$ (Control part)	542	n/a	90.33	89.3
$\bar{v}$ (All functions)	1.80	0.07	2.13	0.08
$\bar{v}$ (Functions with runs on)	3.57	0.16	2.06	0.07
$\bar{v}$ (Functions without runs on)	1.07	0.04	3.70	1.00
$\bar{v}$ (Testcases)	2.32	0.09	1.03	0.03
$\bar{v}$ (Altsteps)	2.80	0.25	5.82	1.17
$\bar{v}$ (Behaviour running on components)	2.70	0.08	2.24	0.10
$\bar{v}$ (Testcases including referenced behaviour)	2.59	0.01	3.61	0.02

The average Cyclomatic Complexities have been measured in various groupings. In both test suites, the main control parts ($\overline{v}$ *(Control part)*) tend to be extreme outliers. The reason is that the control parts contain many *if* statements to select the test cases to be executed depending on sets of capabilities of the implementation. Therefore, we provide also $\overline{v}$ *(All behaviour except control part)*. In addition, we noticed that both test suites contain a lot of behaviour with the minimal Cyclomatic Complexity of 1, i.e. they contain linear control flows only. To get an impression of the Cyclomatic Complexity without these two extremes, we calculate as well $\overline{v}$ (*All behaviour except control part and linear behaviour)*: it is roughly twice as high as $\overline{v}$ *(All behaviour)*. The Cyclomatic Complexity of functions have been measured in different variants as well: in TTCN-3, behaviour running on a component is often concerned with sending and receiving messages and hence, the nature of behaviour running on a component ($\overline{v}$ *(Functions with runs on)*) and behaviour not running on a component ($\overline{v}$ *(Functions without runs on)*) might be different. Concerning the metrics $\overline{v}$ *(Behaviour running on components)* and $\overline{v}$ *(Testcases including referenced behaviour)*, it should be noted that Table 1 does not present the averages for each individual component or testcase respectively, but instead averages over all of them. As a result, components on which, for example, only little behaviour is running are weighted in the same way as components on which a lot of behaviour is running. TRex allows these metrics to be displayed per component or per function respectively, too.

The standard deviations do not yield big surprises. There is only one control part in the SIP test suite. Hence, there is no standard deviation. The control part deviation of the IPv6 test suite reflects the fact that there is a single complex main control part which can be controlled through module parameters and two others with a linear control flow. Hence, there is a high standard deviation and the corresponding complexity value is lower than the one of the single SIP control part.

It is remarkable that except for the control part, the Cyclomatic Complexity of individual behaviour is mostly below the boundary value of 10 suggested by McCabe.[3] Initially, we expected that the Cyclomatic Complexities of test cases would violate this boundary value, due to frequent usage of *alt* and *altstep* statements which may contain many alternative branches. Even McCabe already discusses the exemption of *switch case* constructs from the boundary value of 10 [WM96]. However, our results suggest that boundary values need not be changed for TTCN-3 test suites except for the control part.

The fact that the complexity of control parts is in the order of one to two magnitudes larger than the traditional boundary value, might indicate that TTCN-3 lacks language constructs suitable for the control part. Even though the introduction of such language constructs would not change the Cyclomatic Complexity of the corresponding control flow graph, such instructions would nevertheless improve internal quality properties like readability and maintainability. While TTCN-3 does not have such simplifying

[3] In Table 1, all Cyclomatic Complexities, except the ones for the control part, are below the boundary due to averaging.

constructs yet, it may be reasonable to handle the control part as a special case. A possible approach could be to increase the boundary value by the number of statements which execute a test case and are guarded by a condition.

4 Summary and outlook

We presented TRex, a general open-source quality assessment and quality improvement tool for TTCN-3 test suites. TRex is implemented as an Eclipse plug-in and provides metrics and refactoring for TTCN-3. Special rules, which interpret metric values, support an automatic refactoring of TTCN-3 test suites. Furthermore, we investigated several metrics to assess the overall quality of TTCN-3 test suites. Concerning the Cyclomatic Complexity, we found out that the traditional boundary value of 10 is still valid for TTCN-3 test suites, except for the control part.

Future versions of TRex will include enhanced editing functionality and further refactoring, analyses, and metrics for TTCN-3 test suites. For example, we like to design a metric to assess the cohesion of TTCN-3 modules: the best practise in TTCN-3 to separate data and behaviour into different modules, requires cohesion metrics which differ completely from known cohesion metrics for implementation languages. Furthermore, we only discussed complexity metrics regarding readability and maintenance properties. Constructs such as the *interleave* statement, alternatives, or even expressions would have to be modelled differently in the control flow graph if the semantical complexity is examined: the *interleave* statement is in fact a shorthand for several interleaved *alt* statements; *alt* and *altsteps* constructs are usually implemented as a loop in which snapshots are taken; Boolean expressions are based on shortcut evaluation which leads to further branches in the control flow graph. Other interesting properties, e.g. those related to TTCN-3 defaults which can be activated and deactivated during runtime, cannot be assessed statically, but require a dynamic analysis which we would like to implement in the future.

TRex is published as an open-source tool under the Eclipse Public License and is freely available at its website [TRe06]. We invite the TTCN-3 community to use the tool, share their experience, and participate in the future development of TRex.

References

[BEG+06] P. Baker, D. Evans, J. Grabowski, H. Neukirchen, and B. Zeiß. TRex – The Refactoring and Metrics Tool for TTCN-3 Test Specifications. In Proceedings of TAIC PART 2006 (Testing: Academic & Industrial Conference – Practice And Research Techniques), Cumberland Lodge, Windsor Great Park, UK, 29th-31st August 2006. IEEE Computer Society, August 2006.

[BLW05] P. Baker, S. Loh, and F.Weil. Model-Driven Engineering in a Large Industrial Context – Motorola Case Study. In L. Briand and C. Williams, editors, Model Driven Engineering Languages and Systems: 8th International Conference, MoDELS 2005, Montego Bay, Jamaica, October 2-7, 2005, volume 3713 of Lecture Notes in Computer Science (LNCS), pages 476–491. Springer, May 2005.

[Ecl06] Eclipse Foundation. Eclipse. http://www.eclipse.org, 2006.

[ETS04] ETSI TS 102 027-3: 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, October 2004. Version 2.0.

[ETS05] ETSI European Standard (ES) 201 873 V3.1.1 (2005-06): The Testing and Test Control Notation version 3; Parts 1–7. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, also published as ITU-T Recommendation Z.140–Z.146, 2005.

[Fow99] M. Fowler. Refactoring – Improving the Design of Existing Code. Addison-Wesley, 1999.

[FP97] N. E. Fenton and S. L. Pfleeger. Software Metrics. PWS Publishing Company, 1997.

[GEF06] Eclipse Foundation. Eclipse Graphical Editing Framework. http://www.eclipse.org/gef, 2006.

[GHR+03] J. Grabowski, D. Hogrefe, G. Réthy, I. Schieferdecker, A. Wiles, and C. Willcock. An Introduction into the Testing and Test Control Notation (TTCN-3). Computer Networks, 42(3), June 2003.

[McC76] T. J. McCabe. A Complexity Measure. IEEE Transactions of Software Engineering, 2(4):308–320, 1976.

[Ope06] OpenTTCN Oy. OpenTTCN Tester for TTCN-3. http://www.openttcn.com/Sections/Products/OpenTTCN3, 2006.

[Sne04] H. M. Sneed. Measuring the Effectiveness of Software Testing. In S. Beydeda, V. Gruhn, J. Mayer, R. Reussner, and F. Schweiggert, editors, Proceedings of SOQUA 2004 (First International Workshop on Software Quality) and TECOS 2004 (Workshop Testing Component-Based Systems), volume 58 of Lecture Notes in Informatics (LNI). Gesellschaft für Informatik, 2004.

[Tel06] Telelogic AB. Tau/Tester. http://www.telelogic.com/corp/products/tau/tester/index.cfm, 2006.

[Tes06] TestingTechnologies. TTworkbench. http://www.testingtech.de/products/ttwb_intro.php, 2006.

[TRe06] TRex Website. http://www.trex.informatik.uni-goettingen.de, 2006.

[VSD06] D. Vega, I. Schieferdecker, and G. Din. Towards Quality of TTCN-3 Tests. In Proceedings of SAM'06: FifthWorkshop on System Analysis and Modelling, May 31–June 2, 2006, University of Kaiserslautern, Germany, 2006.

[WM96] A. H. Watson and T. J. McCabe. Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric. NIST Special Publication 500-235, National Institute of Standards and Technology, Computer Systems Laboratory, Gaithersburg, MD, United States of America, 1996.

[Zei06] B. Zeiss. A Refactoring Tool for TTCN-3. Master's thesis, Institute for Infommatics, University of Göttingen, Germany, ZFI-BM-2006-05, March 2006.

[ZNG+06] B. Zeiss, H. Neukirchen, J. Grabowski, D. Evans, and P. Baker. Refactoring for TTCN-3 Test Suites. In Proceedings of SAM'06: Fifth Workshop on System Analysis and Modelling, May 31–June 2, 2006, University of Kaiserslautern, Germany, 2006. (Extended version to appear as Refactoring and Metrics for TTCN-3 Test Suites in the Lecture Notes in Computer Science (LNCS) series published by Springer).