Georg-August-Universität
Göttingen
Institut für Informatik

ISSN 1611-1044
Nummer IFI-TB-2006-06

Technischer Bericht

Qualitätssicherung und Qualitätsverbesserung für TTCN-3 Testspezifikationen

–

Unterlagen zu einem Arbeitstreffen am Institut für Informatik
der Georg-August-Universität Göttingen vom 18. Oktober 2006

Jens Grabowski, Helmut Neukirchen, Benjamin Zeiß

Technische Berichte
des Instituts für Informatik
an der Georg-August-Universität Göttingen

07. November 2006

Georg-August-Universität Göttingen
Zentrum für Informatik

Lotzestraße 16-18
37083 Göttingen
Germany

Tel.	+49 (5 51) 39-1 44 14
Fax	+49 (5 51) 39-1 44 15
Email	office@informatik.uni-goettingen.de
WWW	www.informatik.uni-goettingen.de

Zusammenfassung

Am 18. Oktober 2006 fand am Institut für Informatik der Georg-August-Universität Göttingen ein Arbeitstreffen zu dem Thema „Qualitätssicherung und Qualitätsverbesserung für TTCN-3 Testspezifikationen“ statt. Die Teilnehmer an diesem Arbeitstreffen kamen von der Technischen Universität Berlin, der Universität Dortmund und der Universität Göttingen. In Form von Kurzvorträgen und Werkzeugvorführungen haben die Teilnehmer ihre Arbeiten an diesem Thema präsentiert und zur Diskussion gestellt. Der vorliegende technische Bericht dokumentiert das Arbeitstreffen. Er beinhaltet die Vortragsfolien der Teilnehmer und zusätzliches bisher unveröffentlichtes Informationsmaterial.

Abstract

On October 18th, 2006, the Institute for Informatics at the Georg-August-University in Göttingen organized a technical meeting with the subject „Quality assurance and quality improvement for TTCN-3 test specifications“. Participants from the Technical University of Berlin, the University of Dortmund and the University of Göttingen presented and discussed their work in this field. This technical report documents the technical meeting. It includes the slides of the presentations and further unpublished material.

Keywords:

Softwarequaltity, Metrics, Refactoring, Codesmells, Testspecification, TTCN-3

Vorwort

Für das Testen von modernen Kommunikationssystemen sind meist sehr umfangreiche und komplexe Testreihen erforderlich. Solche Testreihen werden mit Hilfe von speziellen Testspezifikations- und Testimplementierungssprachen, wie z.B. der „Testing and Test Control Notation“ (TTCN-3), entwickelt. Da sich moderne Kommunikationssysteme fortlaufend weiterentwickeln, müssen auch die zugehörigen Testreihen fortlaufend gepflegt und weiterentwickelt werden. Aus der Softwaretechnik ist bekannt, dass die Pflege und Weiterentwicklung von Software durch Verfahren zur Qualitätssicherung und Qualitätsverbesserung unterstützt und vereinfacht werden kann. Solche Verfahren sind z.B. der Einsatz von Metriken zur Qualitätsbewertung und der Einsatz von Refactoring zur Verbesserung der Struktur des Quelltextes. Da Testreihen Software mit speziellen Eigenschaften sind, stellt sich die Frage, wie die aus der Softwaretechnik bekannten Verfahren zur Qualitätssicherung und Qualitätsverbesserung an die Bedürfnisse der Testentwicklung angepasst werden können. Ein Bedarf an solchen Verfahren wurde insbesondere im Bereich der auf TTCN-3 basierenden Testentwicklung festgestellt.

Aus diesem Grund fand am 18. Oktober 2006 im Institut für Informatik der Georg-August-Universität Göttingen ein Arbeitstreffen zu dem Thema „Qualitätssicherung und Qualitätsverbesserung für TTCN-3 Testspezifikationen“ statt. Die Teilnehmer an diesem Arbeitstreffen kamen von der Technischen Universität Berlin, der Universität Dortmund und der Universität Göttingen. In Form von Kurzvorträgen und Werkzeugvorführungen haben die Teilnehmer ihre Arbeiten an diesem Thema präsentiert und zur Diskussion gestellt. Der vorliegende technische Bericht dokumentiert das Arbeitstreffen. Er beinhaltet die Vortragsfolien der Teilnehmer und zusätzliches bisher unveröffentlichtes Informationsmaterial.

Für uns war das Arbeitstreffen ein voller Erfolg. Die Vorträge und Diskussionen haben gezeigt, dass das Thema „Qualitätssicherung und Qualitätsverbesserung in der Testentwicklung“ Potential hat und dass die Arbeiten an diesem Thema erst ganz am Anfang stehen. Wir danken allen Teilnehmern für Ihre Vorträge und Diskussionsbeiträge.

Göttingen, November 2006

Jens Grabowski, Helmut Neukirchen
und Benjamin Zeiss

Inhalt/Contents

TTCN-3 Qualitätssicherung mit TRex

Martin Bisanz, Jens Grabowski,
Helmut Neukirchen, Benjamin Zeiss

Software Engineering for Distributed Systems Group,
University of Göttingen

Inhalt

- **TRex - Überblick**
- TRex - Alte Funktionalitäten
- TRex - Neue Funktionalitäten
- TTCN-3 Metriken
- Zusammenfassung / Offene Punkte

TRex - Überblick

- TTCN-3 Refactoring and Metrics Tool
- TTCN-3 Eclipse Entwicklungsumgebung
- Plattform für Forschungsarbeiten
- Als Open-Source veröffentlicht seit Mai 2006 (Eclipse Public License)

Inhalt

- TRex - Überblick
- **TRex - Alte Funktionalitäten**
- TRex - Neue Funktionalitäten
- TTCN-3 Metriken
- Zusammenfassung / Offene Punkte

TRex - Alte Funktionalitäten 1/2

- TTCN-3 Entwicklungsumgebung
  - Syntax Highlighting
  - Syntaktische Analyse (ANTLR Lexer/Parser)
  - Text Hover
  - Code Completion
  - Refactoring

TRex - Alte Funktionalitäten 2/2

- Code Formatierer
- Einfache Zählmetriken
- Durch Metriken getriggerte Refactorings (Quickfixes)
- Referenzen Sucher
- Outline
- Einfache Telelogic Tau Integration

Inhalt

- TRex - Überblick
- TRex - Alte Funktionalitäten
- **TRex - Neue Funktionalitäten**
- TTCN-3 Metriken
- Zusammenfassung / Offene Punkte

TRex - Neue Funktionalitäten 1/4

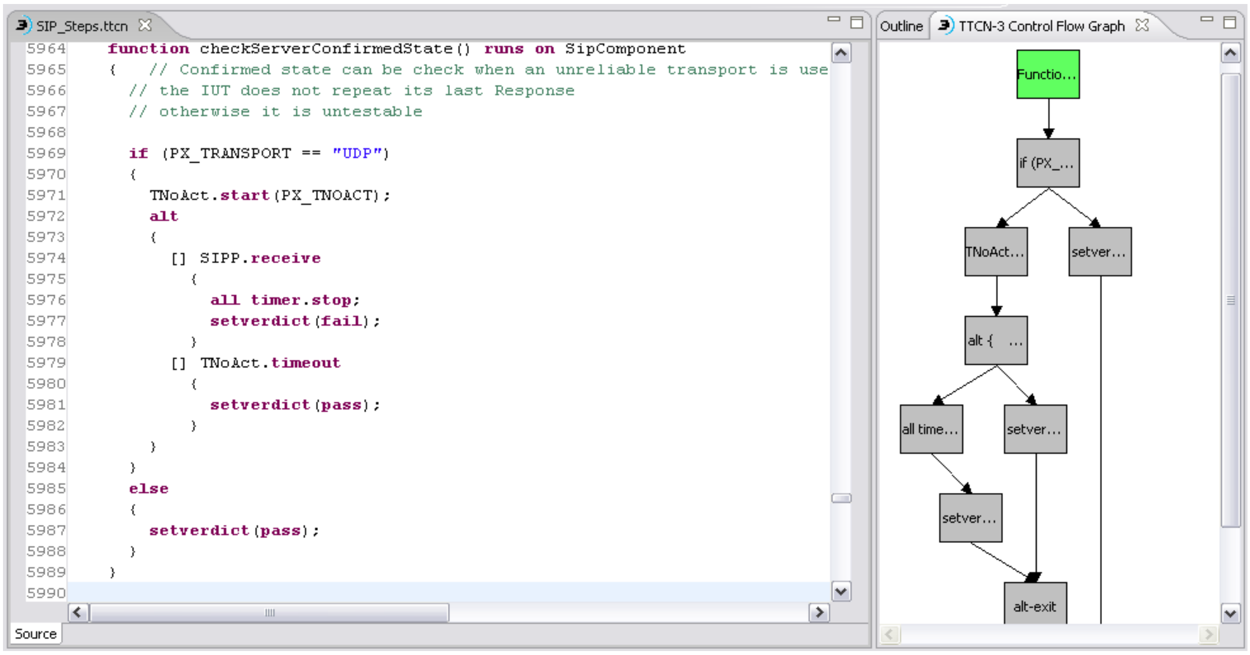
- Compiler-Integration
  - bisher:
    - Hartkodierte Telelogic Tau Integration
    - Aufruf durch Toolbar-Klick
  - Neuimplementierung:
    - Implementiert als erweiterbarer Eclipse Builder
    - Verhalten wird von Compiler-Plugins bestimmt
    - Erstes Compiler-Plugin: Danet TTCN-3 Compiler
    - Integration von anderen externen Compiler **sehr** einfach

TRex - Neue Funktionalitäten 2/4

- Pattern-basierte Smell-Analyse
- Erzeugung und Visualisierung von Kontrollflussgraphen
- Erzeugung und Visualisierung von Call-Graphen
- Auf Basis von Kontrollflussgraphen und Call-Graphen: Komplexitätsmetriken
- Zählmetriken

TRex - Neue Funktionalitäten 3/4

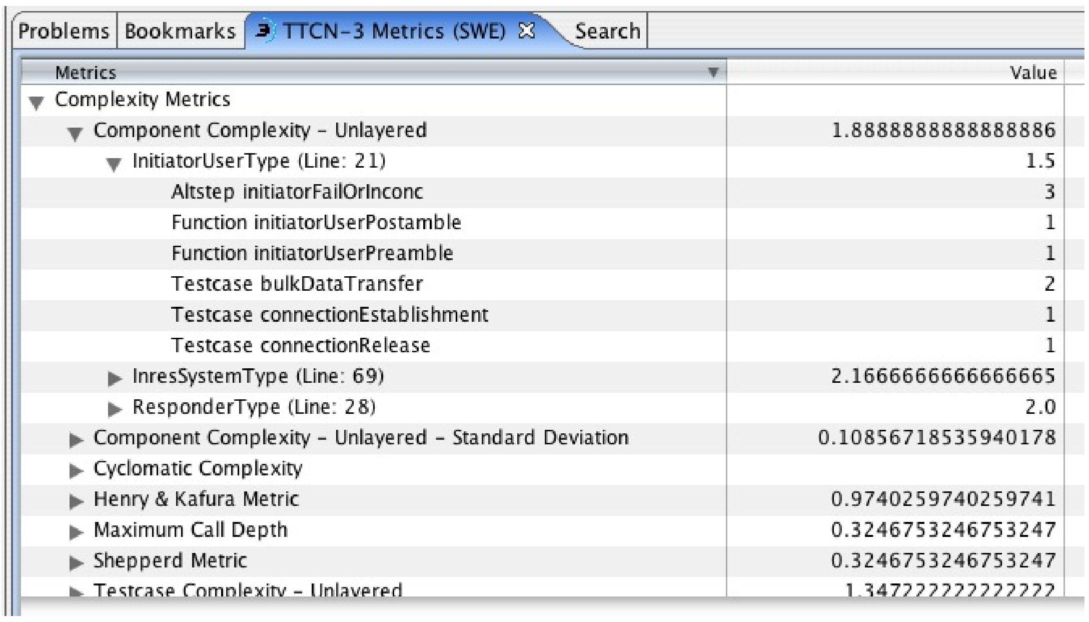
- Kontrollflussgraph Visualisierung:



```
    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);
      }
    }
```

TRex - Neue Funktionalitäten 4/4

- Metrics View:



Problems | Bookmarks | TTCN-3 Metrics (SWE) | Search

Metrics	Value
▼ Complexity Metrics	
▼ Component Complexity – Unlayered	1.8888888888888886
▼ InitiatorUserType (Line: 21)	1.5
Altstep initiatorFailOrInconc	3
Function initiatorUserPostamble	1
Function initiatorUserPreamble	1
Testcase bulkDataTransfer	2
Testcase connectionEstablishment	1
Testcase connectionRelease	1
▶ InresSystemType (Line: 69)	2.1666666666666665
▶ ResponderType (Line: 28)	2.0
▶ Component Complexity – Unlayered – Standard Deviation	0.10856718535940178
▶ Cyclomatic Complexity	
▶ Henry & Kafura Metric	0.9740259740259741
▶ Maximum Call Depth	0.3246753246753247
▶ Shepperd Metric	0.3246753246753247
▶ Testcase Complexity – Unlayered	1.3472222222222222

TRex - Geplante Funktionalitäten

- Semantische Analyse
- (Type-Checking, Datenflussanalyse, ...)

- Mehr Refactorings

- Metriken Fine-Tuning

Inhalt

TTCN-3 Metriken: Beispiel

- Goal, Question metric (GQM) approach.
  - Basili, Weiss: A Methodology for Collecting Valid Software Engineering Data. IEEE Transactions on SE, 1984.

- Ziel: Lesbarkeit verbessern
- Studium von existierenden
- TTCN-3 Testsuiten

- Frage: Gibt es tief verschachtelte Zuweisungen in Komponenten?

- Metrik: maximale
- Verschachtelungstiefe bei
- Zuweisungen in Komponenten

```
type component SipComponent {
   ...
   var From v_From := { // value of From header
        fieldName := FROM_E,
        addressField := {nameAddr :=
          {displayName := omit,
            addrSpec := {
              scheme  := SIP_SCHEME,
              userInfo := omit ,
              hostPort := {host:="",
                         portField:=DEFAULT_SIP_PORT},
                         urlParameters := omit,
                         headers := omit
                       }
          }
        },
        fromParams := omit
      };
   ...
}
```

TTCN-3 Metriken: Beispiel

- Ziel: fehlerträchtiges Verhalten identifizieren
- Frage: Gibt es viele Kontrollflussverzweigungen?
- Metrik: McCabe zyklomatische Komplexität v(G):=e-n+2

Zyklomatische Komplexität

- Kontrollflussgraph:
  - Einfach: if, for, while etc.
  - TTCN-3 spezifisch: alt, altstep, defaults
- Berechnung für verschiedene Verhaltens-Gruppen: Testcases, Funktionen, Altsteps, Control Part
- Erlaubt Interpretation von Unterschieden zwischen den Verhaltensgruppen
- Testcase Komplexität mittels Call-Graph
  - Arithmetisches Mittel über die zyklomatische Zahl jeder Verhaltensgruppe, die potentiell von einem Testcase aufgerufen werden kann.

Komplexitäts-Messungen 1/2

- Messungen der SIP und IPv6 Testsuiten:

Metric	SIP	$\int_v$	IPv6	$\int_v$
Lines of code	42397		46163	
Number of control parts	1		3	
Number of functions	785		643	
Number of testcases	528		295	
Number of altsteps	10		11	
Number of components	2		10	
v(All behaviour)	2.43	0.41	2.11	0.29
v(Control part)	542.00	n/a	90.33	89.3
v(All behaviour except control part)	2.01	0.05	1.83	0.06
v(All behaviour except control part and linear behaviour)	4.12	0.11	3.44	0.14
v(All functions)	1.80	0.07	2.13	0.08
v(Testcases)	2.32	0.09	1.03	0.03
v(Altsteps)	2.80	0.25	5.82	1.17

Komplexitäts-Messungen 2/2

- McCabes Richtlinie von V(G) <= 10 gilt anscheinend auch für TTCN-3 Testcases
- v(G) > 10 in den ETSI Testsuiten:
  - Einige wenig Funktionen, Testcases, die tatsächlich relativ komplex sind
  - Fast alle Control Parts
    - Selektion von Testcases gesteuert durch Modulparameter

```
control {
   if (runRGRT()) {
    if (runRGRTV001()) {
      execute(SIP_RG_RT_V_001());
    };
    if (runRGRTV002()) {
      execute(SIP_RG_RT_V_002());
    };
    …
```

Inhalt

- TRex - Überblick
- TRex - Alte Funktionalitäten
- TRex - Neue Funktionalitäten
- TTCN-3 Metriken
- **Zusammenfassung / Offene Punkte**

Zusammenfassung

- TRex: alte, neue und geplante Funktionalitäten
- TTCN-3 Metriken
- Zyklomatische Komplexität von ETSI TTCN-3 Testsuiten

Offene Punkte

- Kontrollflussgraph für „Reading Complexity", „Semantical Complexity" unterschiedlich:
  - Interleave Statement
  - Shortcut-Evaluation von Ausdrücken
  - Snapshot-Semantik
- Defaults:
  - Handhabung im Callgraphen „naiv" (worst-case)
  - Analysen bzgl. Defaults:
    - Wird jeder aktivierte Default auch wieder deaktiviert?
- Qualitätsmodell für TTCN-3

Literatur

- Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, Paul Baker.
- **Refactoring and Metrics for TTCN-3 Test Suites**. Accepted for LNCS proceedings of selected revised papers of SAM'06 -- Fifth Workshop on System Analysis and Modelling (formerly SDL and MSC Workshop), to appear in Lecture Notes in Computer Science (LNCS) 4320, Springer, Oktober 2006.

- Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, Paul Baker.
- **TRex - An Open-Source Tool for Quality Assurance of TTCN-3 Test Suites**. Proceedings of CONQUEST 2006 – 9th International Conference on Quality Engineering in Software Technology, September 27–29, Berlin, Germany, dpunkt.Verlag, Heidelberg, September 2006.

- Benjamin Zeiß, Helmut Neukirchen, Jens Grabowski, Dominic Evans, Paul Baker.
- **Refactoring for TTCN-3 Test Suites**. Proceedings of SAM'06 -- Fifth Workshop on System Analysis and Modelling (formerly SDL and MSC Workshop), May 31st-June 2nd 2006, University of Kaiserslautern, Kaiserslautern, Germany, 2006.

- Benjamin Zeiß.
- **A Refactoring Tool for TTCN-3**. Masterarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-BM-2006-05, ISSN 1612-6793 (Tippfehlerbereinigte Version), Zentrum für Informatik, Georg-August-Universität Göttingen, März 2006.

- Jochen Kemnade.
- **Development of a Semantics-aware Editor for TTCN-3 as an Eclipse Plug-in**. Bachelorarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-BM-2005-19, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, September 2005.

TTCN-3 Code Smells

Martin Bisanz

Institut für Informatik, Georg-August-Universität Göttingen
Gruppe Softwaretechnik für Verteilte Systeme

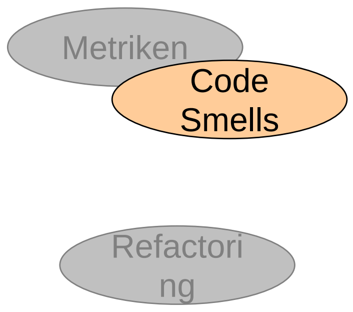
Inhalt

Einordnung

- Qualitätssicherung für TTCN-3:
  - qualitative **Bewertung** von Testsuiten
  - Auffinden **problematischer Stellen**

⬇

  - **Restrukturierung** von Testsuiten
- Stichwörter:
  - Statische Analyse
  - Code Smells
  - (Anti-)Patterns



Bad Smells in Code

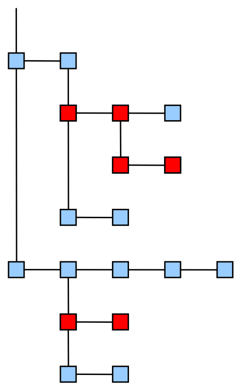
"...certain structures in the code that suggest (sometimes they scream for) the possibility of refactoring."

Fowler: Refactoring – Improving the Design of Existing Code. Addison-Wesley (1999)

- Code-Muster, die auf schlechtem Design oder schlechten Programmierpraktiken beruhen
- Nicht zwingend fehlerhafte Stellen, aber ***potentiell fehlerhaft*** bzw. ***fehleranfällig***
- ***Auch als Anti-Patterns bezeichnet***

Smells vs. Metriken

- ***Smell: Suche bestimmter Muster***
  - ***Duplizierter Code***
  - ***Daten- und Kontrollflussanomalien***
- ***Metrik: Ermittlung eines Wertes***
- ***Keine klare Abgrenzung:***
  - ***Smell x $\Rightarrow$ Metrik „Anzahl von x“***
  - ***Metrik y mit Grenzwert z $\Rightarrow$ Smell „y verletzt z“***



Inhalt

- Einordnung
- **TTCN-3 Code-Smell-Katalog**
- Beispiel
- Implementierung
- Zusammenfassung/Ausblick

TTCN-3 Code-Smell-Katalog

- Systematische Auflistung von Code-Smells für TTCN-3
- Quellen/Ideengeber u.a.:
  - Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley (1999)
  - Zeiss, B.: A Refactoring Tool for TTCN-3. Master's thesis, Göttingen (2006)
  - Code-Smell-Tools für Java
- 8 Kategorien
- Schema: Beschreibung, Motivation, Gegenmaßnahme, Beispiel

TTCN-3 Code Smells (1)

- Duplicated Code:
  - *Duplicate Statements*
  - *Duplicated Code in Conditional*
  - Duplicate Alt Branches
  - Duplicate Inline Templates
  - Duplicate Template Fields
  - Duplicate Component Definition
  - Duplicate Local VCT *

* Variable/Constant/Timer

TTCN-3 Code Smells (2)

- References:
  - Singular Template Reference
  - Singular Component VCT * Reference
  - Unneeded Parameter
  - Unused Definition (global/local)
  - Unused Parameter
  - Unused Imports
  - Unrestricted Imports

* Variable/Constant/Timer

TTCN-3 Code Smells (3)

- Complexity:
  - Long Statement Block
  - Long Parameter List
  - Complex/Nested Conditional
- Default Anomalies:
  - Activation Asymmetry
  - Unreachable Default

TTCN-3 Code Smells (4)

- Test Execution
  - Missing Verdict
  - Missing *log*-Statement
  - *stop* in Function
  - PTC not connected

TTCN-3 Code Smells (5)

- Coding-Standards:
  - Magic Values (Numbers/Strings)
  - Violation of Naming Convention
  - Disorder
  - Unsufficient Grouping
  - Comments
  - Goto

TTCN-3 Code Smells (6)

- Possible Bugs:
  - Name-clashing Imports
  - Data Flow Anomalies
- Other:
  - *Over-specific runs on*
  - *Out-of-bound Metric*

Inhalt

- Einordnung
- TTCN-3 Code-Smell-Katalog
- **Beispiel**
- Implementierung
- Zusammenfassung/Ausblick

Activation Asymmetry (1)

- **Beschreibung**:
  - Default-Aktivierung und -Deaktivierung finden nicht im selben Statement-Block statt.
- **Motivation**:
  - Um die Lesbarkeit zu erhöhen, wird empfohlen, Defaults am Anfang eines Statement-Blocks zu aktivieren und am Ende wieder zu deaktivieren.
- **Gegenmaßnahme(n)**:
  - Fehlende Aktivierung/Deaktivierung hinzufügen
  - Aktivierung und Deaktivierung in den selben Statement-Block verschieben

Activation Asymmetry (2)

```
testcase myTestcase1() runs on MyComponent {
    var default myDefaultVar := activate(myAltstep);
    // ...
    deactivate(myDefaultVar);
}

testcase myTestcase2() runs on MyComponent {
    var default myDefaultVar := activate(myAltstep);
    // ...
}

testcase myTestcase3() runs on MyComponent {
    var default myDefaultVar := activate(myAltstep);
    // ...
    if (checkSomething()) {
        // ...
        deactivateDefault(myDefaultVar);
    }
}

testcase myTestcase4() runs on MyComponent {
    activate(myAltstep);
    // ...
    deactivate;
}
```

Duplicate Alt Branches (1)

- **Beschreibung**:
  - Duplizierte alt-Zweige in alt-Statements von Funktionen, Altsteps und Testcases.
- **Motivation**:
  - Duplizierter Code vermindert Lesbarkeit und Änderbarkeit.
- **Gegenmaßnahme(n)**:
  - *Extract Altstep*
  - *Split Altstep*
  - *Replace Altstep with Default*

Duplicate Alt Branches (2)

```
testcase tc1() runs on
        ExampleComponent {
    timer t1;
    //...
    t1.start(10.0);
    alt {
        [] pt.receive(message1) {
            pt.send(message2);
        }
        [] any port.receive {
            setverdict(fail);
            stop;
        }
        [] t1.timeout {
            setverdict(fail);
            stop;
        }
    }
}
```

```
testcase tc2() runs on
        ExampleComponent {
    timer t2;
    //...
    t2.start(10.0);
    alt {
        [] pt.receive(message3) {
            pt.send(message4);
        }
        [] any port.receive {
            setverdict(fail);
            stop;
        }
        [] t2.timeout {
            setverdict(fail);
            stop;
        }
    }
}
```

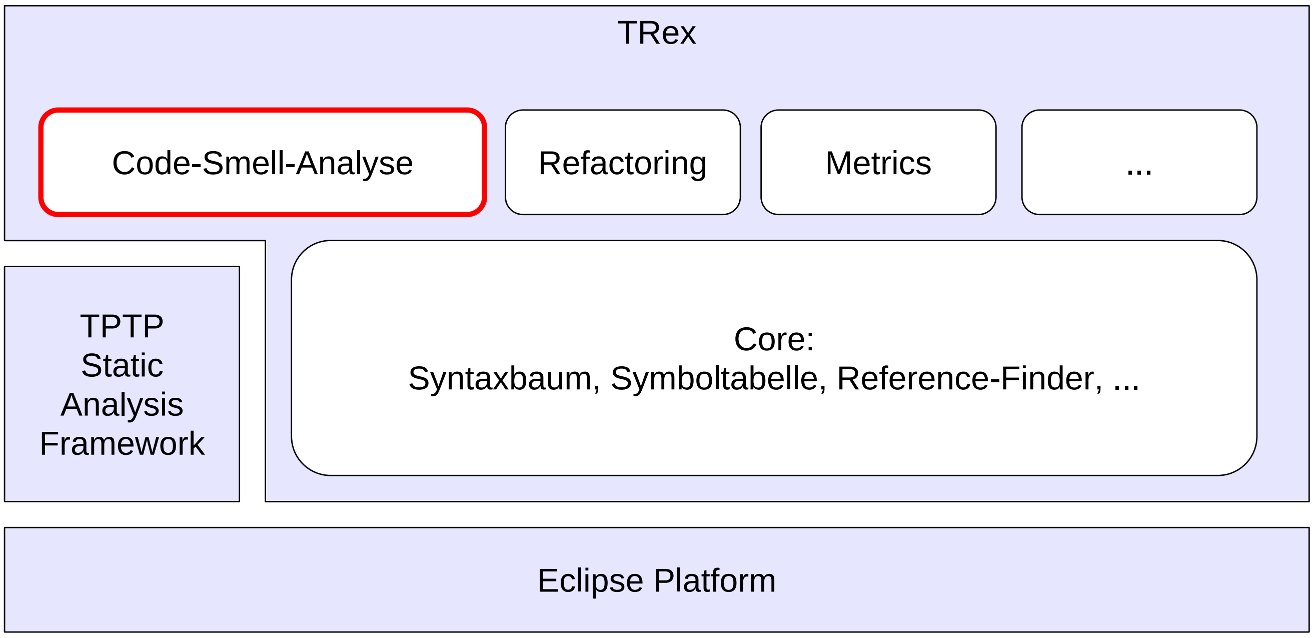
Inhalt

- Einordnung
- TTCN-3 Code-Smell-Katalog
- Beispiel
- **Implementierung**
- Zusammenfassung/Ausblick

TRex Code-Smell-Analyse

- Plug-In für **Eclipse-Platform, TRex**
- Nutzt das **Static Analysis Framework** aus der Test & Performace Tools Platform (TPTP):
  - Analyse-Konfiguration und Resourcenauswahl
  - Regel-Gruppierung, -Parametrisierung, Templates
  - Ergebnisverwaltung und -darstellung, Historie
  - QuickFix Support

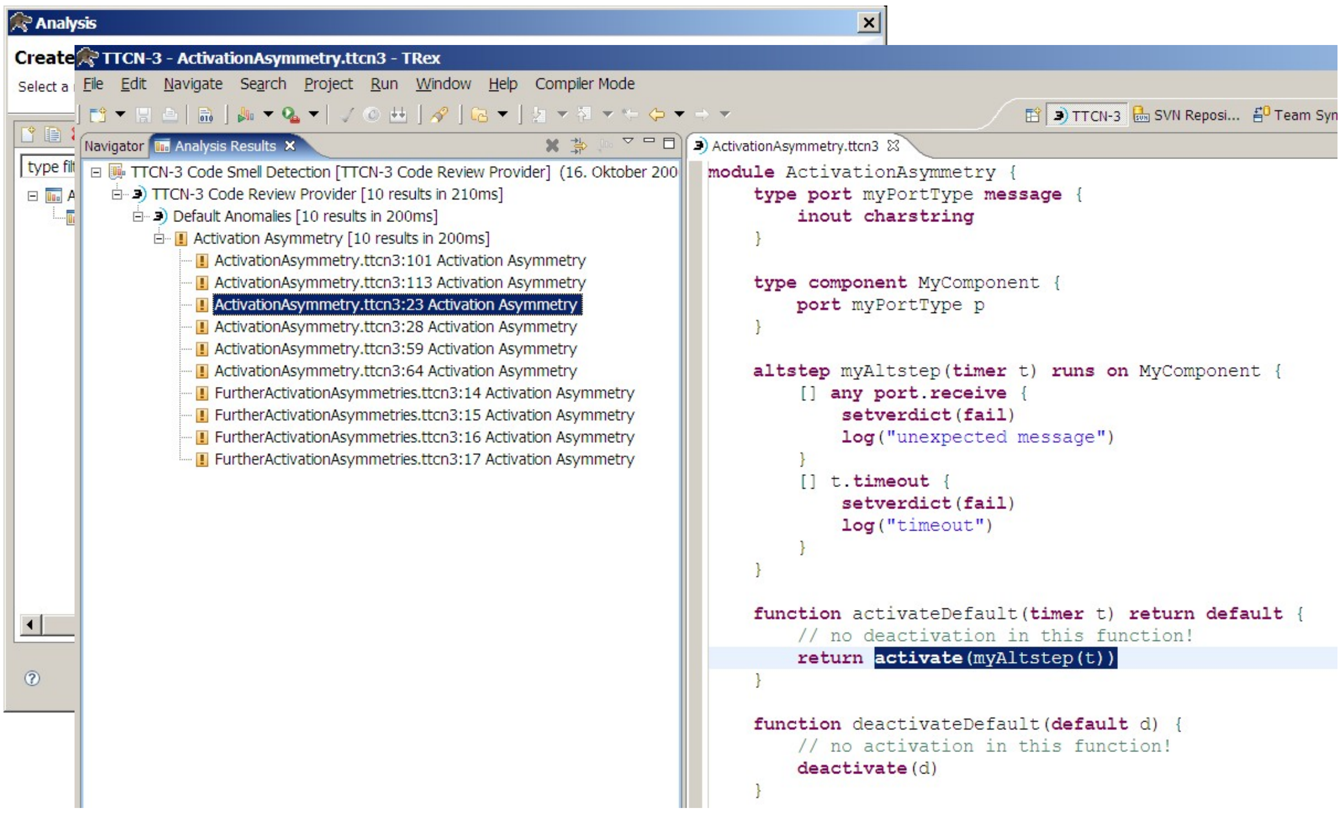
Architektur


TRex
Code-Smell-Analyse
Refactoring
Metrics
...
TPTP Static Analysis Framework
Core: Syntaxbaum, Symboltabelle, Reference-Finder, ...
Eclipse Platform

Implementierung: Magic Strings

```
public class MagicStringRule extends AbstractAnalysisRule {
  private final int[] NODE_TYPES = new int[] {
      TTCN3LexerTokenTypes.BitStringValue,
      TTCN3LexerTokenTypes.CharStringValue,
      TTCN3LexerTokenTypes.OctetStringValue,
      TTCN3LexerTokenTypes.HexStringValue};

  @Override
  public void analyze(AnalysisHistory history) {
    PatternDetectionResource resource = (PatternDetectionResource)
        getProvider().getProperty(history.getHistoryId(),
        PatternDetectionProvider.RESOURCE_PROPERTY);
    List<LocationAST> list = resource.getTypedNodeList(NODE_TYPES);
    for (LocationAST node : list) {
      if (LocationAST.resolveParentsUntilType(node,
          TTCN3LexerTokenTypes.ConstDef) == null)
        resource.generateResultsForASTNode(this,
            history.getHistoryId(), node);
    }
  }
  //...
}
```

Screenshots


Analysis
Create
Select a
TTCN-3 - ActivationAsymmetry.ttcn3 - TRex
File Edit Navigate Search Project Run Window Help Compiler Mode
TTCN-3
SVN Reposi...
Team Syn
Navigator
Analysis Results
ActivationAsymmetry.ttcn3
type filt

```
TTCN-3 Code Smell Detection [TTCN-3 Code Review Provider] (16. Oktober 200
  TTCN-3 Code Review Provider [10 results in 210ms]
    Default Anomalies [10 results in 200ms]
      Activation Asymmetry [10 results in 200ms]
        ActivationAsymmetry.ttcn3:101 Activation Asymmetry
        ActivationAsymmetry.ttcn3:113 Activation Asymmetry
        ActivationAsymmetry.ttcn3:23 Activation Asymmetry
        ActivationAsymmetry.ttcn3:28 Activation Asymmetry
        ActivationAsymmetry.ttcn3:59 Activation Asymmetry
        ActivationAsymmetry.ttcn3:64 Activation Asymmetry
        FurtherActivationAsymmetries.ttcn3:14 Activation Asymmetry
        FurtherActivationAsymmetries.ttcn3:15 Activation Asymmetry
        FurtherActivationAsymmetries.ttcn3:16 Activation Asymmetry
        FurtherActivationAsymmetries.ttcn3:17 Activation Asymmetry
```

```
module ActivationAsymmetry {
    type port myPortType message {
        inout charstring
    }

    type component MyComponent {
        port myPortType p
    }

    altstep myAltstep(timer t) runs on MyComponent {
        [] any port.receive {
            setverdict(fail)
            log("unexpected message")
        }
        [] t.timeout {
            setverdict(fail)
            log("timeout")
        }
    }

    function activateDefault(timer t) return default {
        // no deactivation in this function!
        return activate(myAltstep(t))
    }

    function deactivateDefault(default d) {
        // no activation in this function!
        deactivate(d)
    }
```

Resultate

SIP

- 543 Magic Numbers
- 602 Activation Asymmetries (73 ohne Testcases)
- 119 Duplicate Alt Branches
- 2 Singular Component VCT References
- 50 Unused Local Definitions

IPv6

- 368 Magic Numbers
- 801 Activation Asymmetries (317 ohne Testcases)
- 48 Duplicate Alt Branches
- 15 Singular Component VCT References
- 156 Unused Local Definitions

Inhalt

Zusammenfassung

- Code Smells: Muster für „schlechten“ Code
- Code-Smell-Katalog: >30 Code Smells für TTCN-3
- Implementierung als Plug-In für TRex

Ausblick

- Bibliothek/Framework für Code-Smell-Analyse:
  - Navigation/Selektion im AST
  - Referenzen
  - Duplikate
- Deklarative Smell-Beschreibung
  - XPath
  - Reguläre Ausdrücke
  - Eigene Mustersprache

Literatur

- Fowler, M.: Refactoring – Improving the Design of Existing Code. Addison-Wesley (1999)
- Zeiss, B.: A Refactoring Tool for TTCN-3. Masterarbeit, Institut für Informatik, Universität Göttingen (2006)
- W. J. Brown u.a.: AntiPatterns: Refactoring Software, Architectures and Projects in Crisis. Wiley (1998)
- TRex, http://www.trex.informatik.uni-goettingen.de
- TPTP, http://www.eclipse.org/tptp/
- PMD, http://pmd.sourceforge.net
- Checkstyle, http://checkstyle.sourceforge.net

Quality metrics applied to TTCN-3 test suites

Diana Vega, Ina Schieferdecker

Technical University of Berlin

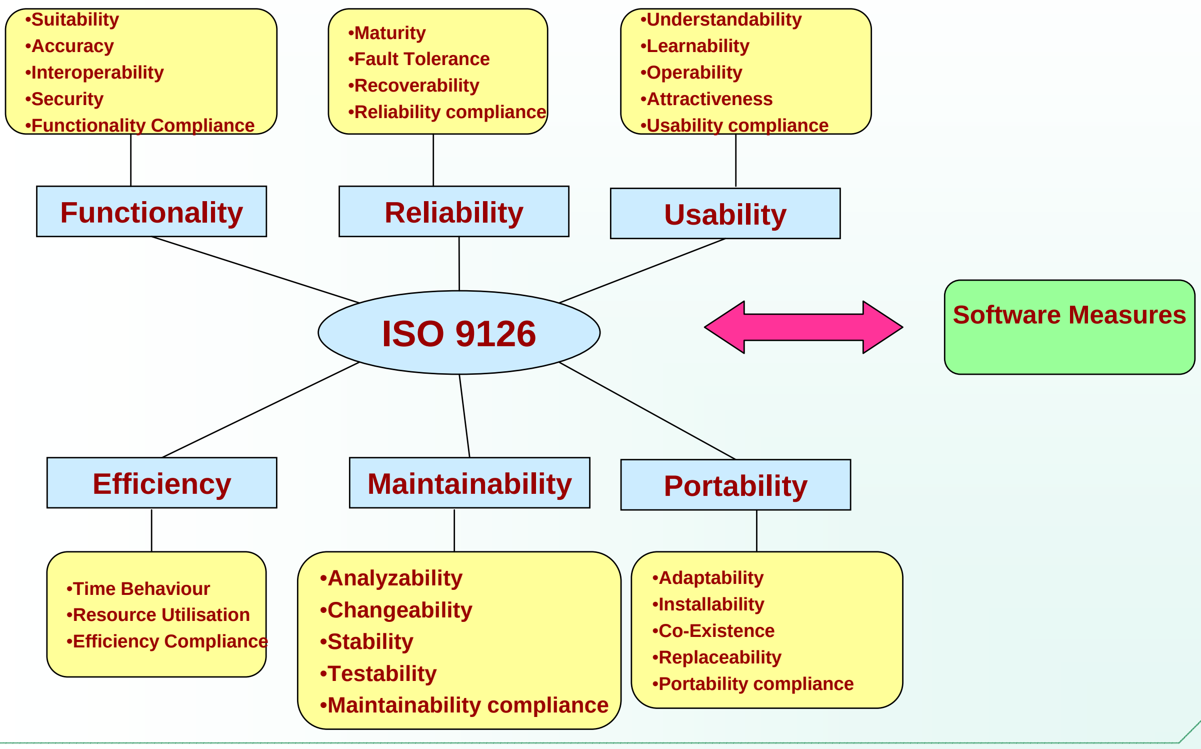
Göttingen 2006

Contents

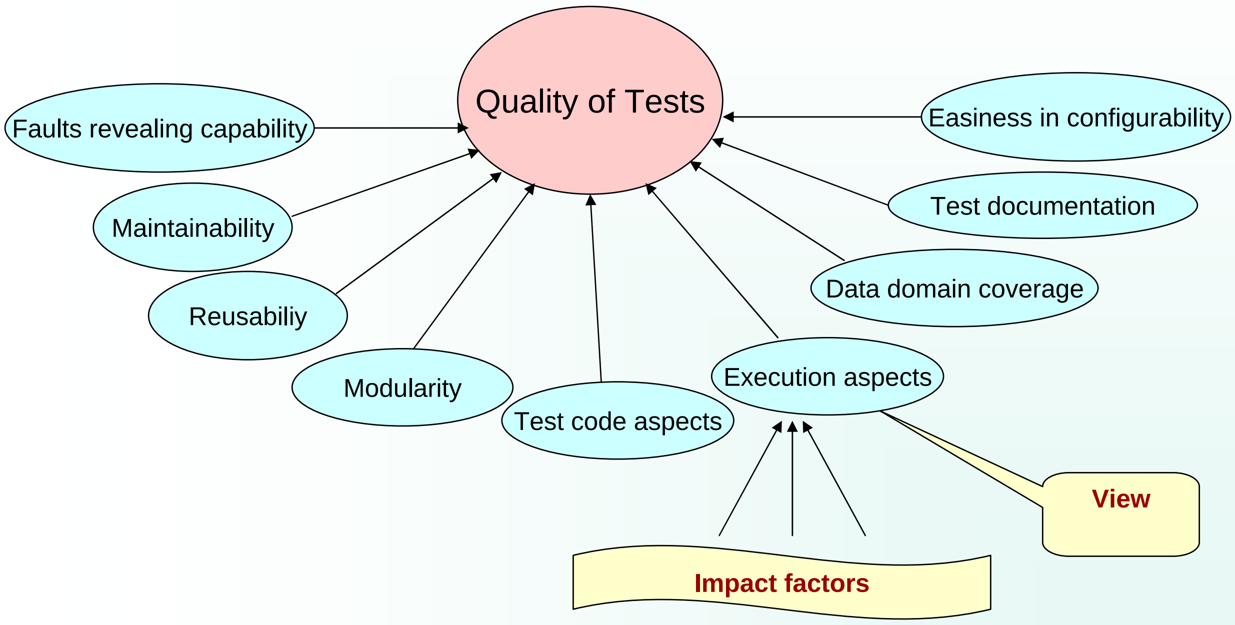
Measurements

- derive a basis for estimates
- track project progress
- determine complexity
- managerial decision-making during the software life-cycle
- help us to make better decisions
  - effort distribution
  - time schedule
  - complexity degree

ISO 9126

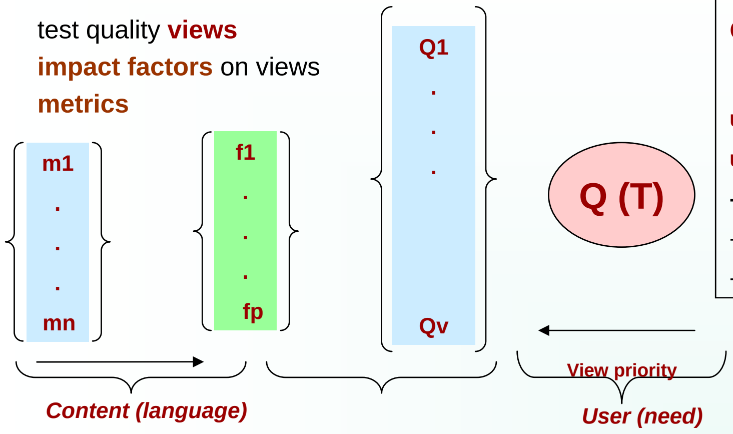


TTCN-3 test quality - a model ?



TTCN-3 test quality - a number ?

- test quality **views**
- **impact factors** on views
- **metrics**



Q = (Q1 u1 + Q2u2 + …+Qv uv) / (100v)

u1 .. uv in [0..100] → Qi in [0..1]

ui :

- User coefficients

- user needs dependent

- given by the user

fi = (c1 m1 + c2m2 + …+cn mn) / n ;

c1, ..cn in [0..1] → fi in [0..1]

ci :

- metrics coefficients

- test language dependent

- correlation

Qi = (p1 f1 + p2f2 + …+pp fp) / p ;

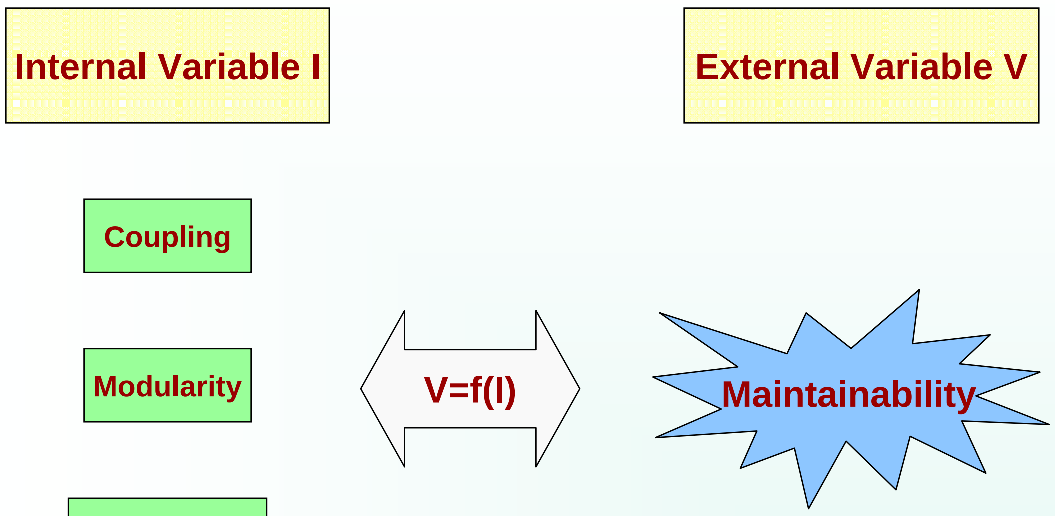
p1, ..pp in [0..1] → Qi in [0..1]

pi :

- factors coefficients

- established

Variable correlation - Prediction



Pearson's Correlation

- degree to which the variables are related

- degree of *linear relationship* between two variables

- from +1 to -1

Sneed's test metrics classification

- metrics for assessing the testability of the software
- metrics for evaluating test cases
- metrics for calculating test costs
- metrics for measuring test coverage
- metrics for assessing test efectiveness

Assessment of a TTCN-3 Test Suite

- Two points of view:
  - (1) Abstract Test Suite (ATS)
    - SUT interface representation is exhibited to TTCN-3 test infrastructure by means of the TSI (test system interface), test data type descriptions, etc.
  - (2)Executable Test Suite (ETS)
    - communication with SUT
- Analysis of test suites
  - Statically
    - inspecting the ATS without running the test against SUT
    - TTCN-3 static metrics
  - Dynamically
    - verdicts establishment
    - execution artifacts (e.g. time)
    - TTCN-3 dynamic metrics

General TTCN-3 Static Metrics

- Static metrics similar to other programming languages

- **lines of code**:
- **number of words**
- **test suite size**
- **min, max, average lines of code for a function**
- **min, max, average number of parameters for a function**
- **number of unused variables**
- min, max, average **cyclomatic complexity for a function**
- min, max, average **function Fan-in**
- min, max, average **function Fan-out**

Specific TTCN-3 Static Metrics

- Static metrics TTCN-3 specific : Volume , Relational
  - min, max, average **lines of code for a test case**
  - min, max, average **number of parameters for a test case**
  - min, max, average **altstep Fan-in**
  - min, max, average **altstep Fan-out**
  - min, max, average **number of parameters for a template**
  - number of **inline templates**
  - min, max, average **number of ports for a component**
  - min, max **depth of a user-type definition**
  - max **spreading of user-defined type**: maximum number of fields of a TTCN-3 user-defined type
  - max **depth of a type definition of a template sent via a port**
  - max **depth of inheritance of templates** by means of **modifiers**
  - min, max, average **test case call for execution depth**
  - max **depth of modules dependency**

TTCN-3 Dynamic Metrics

- Decisive factor in analyzing the quality of a test
- Based on the information recorded during or after test execution
  - number of passed/failed test cases after test suite execution
  - min, max, average time for the execution of a test case in a test suite
  - min, max, average number of parallel test components (PTCs) employed for the execution of a test case
- Time evolution perspective
  - statistics related to test cases that changed their verdicts

Example

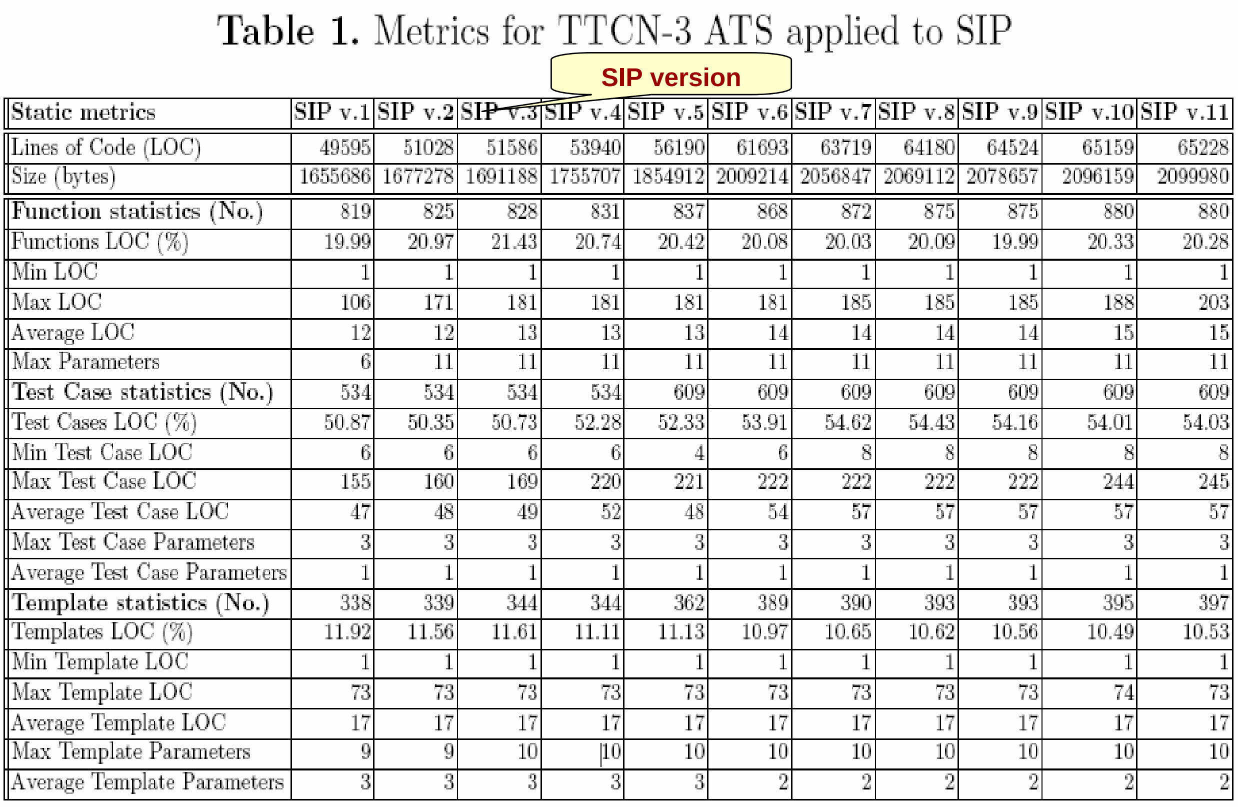


Table 1. Metrics for TTCN-3 ATS applied to SIP

SIP version

Static metrics	SIP v.1	SIP v.2	SIP v.3	SIP v.4	SIP v.5	SIP v.6	SIP v.7	SIP v.8	SIP v.9	SIP v.10	SIP v.11
Lines of Code (LOC)	49595	51028	51586	53940	56190	61693	63719	64180	64524	65159	65228
Size (bytes)	1655686	1677278	1691188	1755707	1854912	2009214	2056847	2069112	2078657	2096159	2099980
Function statistics (No.)	819	825	828	831	837	868	872	875	875	880	880
Functions LOC (%)	19.99	20.97	21.43	20.74	20.42	20.08	20.03	20.09	19.99	20.33	20.28
Min LOC	1	1	1	1	1	1	1	1	1	1	1
Max LOC	106	171	181	181	181	181	185	185	185	188	203
Average LOC	12	12	13	13	13	14	14	14	14	15	15
Max Parameters	6	11	11	11	11	11	11	11	11	11	11
Test Case statistics (No.)	534	534	534	534	609	609	609	609	609	609	609
Test Cases LOC (%)	50.87	50.35	50.73	52.28	52.33	53.91	54.62	54.43	54.16	54.01	54.03
Min Test Case LOC	6	6	6	6	4	6	8	8	8	8	8
Max Test Case LOC	155	160	169	220	221	222	222	222	222	244	245
Average Test Case LOC	47	48	49	52	48	54	57	57	57	57	57
Max Test Case Parameters	3	3	3	3	3	3	3	3	3	3	3
Average Test Case Parameters	1	1	1	1	1	1	1	1	1	1	1
Template statistics (No.)	338	339	344	344	362	389	390	393	393	395	397
Templates LOC (%)	11.92	11.56	11.61	11.11	11.13	10.97	10.65	10.62	10.56	10.49	10.53
Min Template LOC	1	1	1	1	1	1	1	1	1	1	1
Max Template LOC	73	73	73	73	73	73	73	73	73	74	73
Average Template LOC	17	17	17	17	17	17	17	17	17	17	17
Max Template Parameters	9	9	10	10	10	10	10	10	10	10	10
Average Template Parameters	3	3	3	3	3	2	2	2	2	2	2

Example indicators

- increasing testing effort in time
  - 49595 LOC to 65228 LOC
- code reusability indicators (large number of parameters for a function)
- modularity indicators (average 15 LOC for a function)
- effort distribution indicators
  - 20% - functions,
  - approx. 50% test cases 20%
  - approx 11% templates definition

TTCN-3 test quality impact factors examples (1)

Test Quality Views and their Impact Factors	*Impact Factors Measures*
Faults revealing capability, Domain coverage **-** I-O data domain partition **-** SUT functionalities coverage (there is at least one test case that covers a functionality of the SUT specified in the requirements - adequacy criterion)	- template variance (no. of templates directly derived from the same data type or indirectly by means of modifiers) - degree of permissibility for receive templates (how many fields are complete specified and not filled in with wildcards)

TTCN-3 test quality impact factors examples (1)

Test Quality Views and their Impact Factors	*Measures*
Maintainability - easiness in understanding (significant identifiers name, code indentation, etc.) - grouping, module libraries design - history of tests configurations - link to the version of requirements	- effort size distribution in terms of LOC - identifiers naming checking (e.g. all identifiers for variables start with v, for functions with f, etc.) - checks for formatting style (usage of automated formatting features as part of the testing tool employed for testing)

TTCN-3 test quality impact factors examples (2)

Test Quality Views and their Impact Factors		*Measures*
Reusability, Modularity, Code aspects - libraries of modules for core components - parameterized behavior functions (that can be called from multiple test scenarios) - stimuli data variance - links to versions of requirements - grouping, module libraries design - importing of data types from other languages - design of test system interfaces according to SUT interfaces		- how many times a function, altstep or default was called (fan-in, fan-out metrics) - degree of usage of template modifiers for templates that are based on the same type and varies only a few fields values - number of parameters for a behavior entity (function, altstep, test case) - number of "import from nonTTCN3Module language" statements - number of groups - module coupling (depth of import module recursiveness for a module calling a specific test case)

TTCN-3 test quality impact factors examples (2)

Test Quality Views and their Impact Factors		*Measures*
Execution aspects, Configurability - easiness in configurability - compilation speed - degree of automation/user intervention for executing test scenario - overload of test system nodes		- presence of default values for module parameters in configuration files - time metrics (execution time for a test case, time for SUT answer processing, delays, etc) - coupling between behavior entities - number of "map" statements (interaction between test components and TSI) - number of "connect" statements (interaction between test components)
Test documentation - links to requirements document (clearly specify the page, section, etc.) - document for installation, set-up, parameter setting, clear instructions for running		- checks for test documentation format for specific entities (ongoing standard proposed by ETSI)

Conclusion

- qualitative models behind the measures
- measurement and statistical theory
  - to interpret the meaning of numbers and
  - discover correlations
- identified three levels for TTCN-3 quality estimates:
  - Metrics
  - Impact factors
  - Views

References

- ***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.
- 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.
- 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, also published as ITU-T Recommendation Z.140, 2005.

Towards Quality of TTCN-3 Tests

Diana-Elena Vega[2], Ina Schieferdecker[1,2], George Din[1]

[1] Fraunhofer FOKUS, Kaiserin-Augusta-Allee 31, 10589 Berlin, Germany
[2] ETS, TU Berlin, Franklinstr. 28-29, 10587 Berlin, Germany **

Abstract. The estimation of software quality is achieved by using software testing. A challenging task is to establish how good the tests are. In white-box testing techniques, an assessment of quality of tests is given in terms of coverage acquired by analyzing specific values corresponding to standardized software metrics.
TTCN-3 is a test description language oriented towards black-box testing and serves as a good candidate technology for analyzing the quality of this type of tests. This paper will investigate various aspects related to the capability of a test to uncover software faults. The quality analysis of a TTCN-3 based test system is studied on the basis of a) a static analysis of the abstract test suite and b) a dynamic analysis of the interaction with the system under test (SUT). A TTCN-3 test specification includes also a model of the SUT described as a collection of interfaces or operations the test may call from SUT. This information is related then to the rest of the test to determine if the test behaviors cover entirely the SUT specification.
Means for empirical and formal evaluation of test effectiveness, in particular test system interface coverage at the TTCN-3 ATS (abstract test suite) level, have been investigated. We also present the possible advantages in the static-based effectiveness analysis when a TTCN-3 metamodel is accessible behind the textual representation of the test suite.

1 Towards Quality Assessment of Tests

Measurement is critical for software quality assurance and assessment because there is always a need for a numerical data related to software development. Therefore a set of software metrics was introduced to support quantitative managerial decision-making during the software life-cycle as a basis of cost estimation, resource planning and scheduling. Many papers investigated this topic ([3], [4]), even a standard was proposed where quality models, internal metrics, external metrics, quality in use metrics were presented (ISO 9126 [7]). A considerable contribution in software quality assessment is given by software testing. However the main idea behind software metrics lies in using relatively simple metrics combined with various test results for building tools that would be a managerial-decisions help in different kinds of predictions and assessment during the life-cycle.

** This work has been supported by the *Krupp von Bohlen und Halbach - Stiftung*

If the quality estimation of software product and development became a natural part of software development process, a much more research oriented theme is to assess the quality of testing. Next, we will give the significant ideas accepted to have a good impact on quality evaluation by means of fault detection capability, maintainability, readability of a test system.

1.1 Quality Estimation of White-box Testing

First aspects are concerned with giving a measure of *quality of structural testing* (glass-box, white-box) mainly in terms of *coverage*. When the system under test (SUT) is transparent to the test system with respect to the availability of system sources, acquiring coverage data gives valuable information with respect to the measure of test completeness. The concepts of coverage testing are well-described in the literature ([8], [9], [10], [11], [13]), commercial and free tools are already developed for implementing these concepts for standard programming languages such as C, C++, Java, etc. Basically, a code coverage analyzer is the process of finding areas of a program not exercised by a set of test cases, areas that may hide software faults. These techniques determine a quantitative measure of code coverage that is an indirect measure of quality. A large variety of coverage measures exist such as: statement coverage, decision coverage, condition coverage, multiple condition coverage, condition/decision coverage, path coverage, function coverage, call coverage, linear code sequence and jump coverage, data flow coverage, code branch coverage, loop coverage, race coverage, table coverage, line coverage, predicate coverage, etc. All these metrics related to coverage monitoring are structural testing techniques that help eliminating drawbacks in a test suite. The fundamental assumption is: to better uncover faults, control flows should be varied in a test, various elements in the code should be exercised. Unfortunately, applying these techniques does not tell anything about achievable specifications of the system, does not reveal faults of omission. Consequently, coverage analysis has certain strengths and weaknesses, it exposes some plausible faults but does not come close to exposing all classes of faults.

1.2 Quality Estimation of Black-box Testing

In *functional testing* (black-box) where program behavior is compared against a requirements specification, the internals of the SUT cannot be seen. Hence, white-box like coverage measurements cannot be used in order to asses the completeness of testing. To evaluate somehow the quality of black-box testing from the confidence point of view, one must have some kind of failure model or hypothesis, and then has to acquire specification-based testing coverage. It is also assumed that the implementation and the specification resemble each other. When the specification is expressed in a state machine, commonly employed coverage criteria are: *state coverage* (the test suite should check every state in the specification by exercising this state at least once), *transition coverage* (the same but for transitions), *transition sequences coverage* (applying the same idea for consecutive transitions or sequences). This idea may be a good one since a typical error is the omission of a specific transition from the implementation.

Thinking of fault-detection capability of the test suite in black-box testing, another problem has been researched with respect to test suite quality: how to reduce black-box testing efforts while maintaining the quality of the testing process. In [6] an approach of reducing the test suite size while maintaining its fault-detection capability has been introduced. This approach is based on the concept of *Input-Output (IO) Analysis*. The method is mainly used for testing data-driven programs which typically have multiple inputs and outputs; the employed testing techniques are equivalence-class partitioning and boundary value analysis. On the basis of *combinatorial testing techniques*, i.e. the tester must then consider how to test combinations of the selected test data values, IO analysis approach is used to determine the relationship between a programs inputs and outputs in order to reduce a combinatorial test suite by removing tests that are repetitive from the perspective of the program outputs. Further combination and heuristic strategies testing such as Category Partition (CP), Equivalence Partitioning, etc. are surveyed in [14].

Investigating in the direction of quality field of black-box tests is a complex task that touches different knowledge areas: reliable interpretation of a formal representation of a system, various heuristics and artificial intelligence algorithms applied for test data generation, statistical analysis employed in rendering levels of quality estimation. TTCN-3 language, through its textual representation and testing language artifacts offers the desired testing environment where formal and statistical analysis meet. Our motivation in investigating this direction is the not yet proven belief that some correlations between static analysis metrics of TTCN-3 suites, the way that a SUT is modeled in terms of Abstract Test System interface standardized in TTCN-3 and the density of exposed failures have not yet been discovered. Therefore, in the following sections we examine possible numerical data that can be collected during TTCN-3 test design and execution that could give indications for the test quality.

This paper is structured as follows: the following section introduces the TTCN-3 testing language followed by a formal quality definition of a test suite expressed in TTCN-3 core notation format. Next, a set of proposed static and dynamic metrics targeting quality investigation of a TTCN-3 test suite are presented. The fourth section gives insights in a TTCN-3 metrics tool design and its application upon two already developed test suites. Eventually our conclusions are shown and further possible investigation directions are outlined.

2 Metrics Definition for TTCN-3

2.1 TTCN-3 Overview

TTCN-3 [1] is the standard testing language designed to address testing needs of modern telecom and datacom technologies and widen the applicability to many kinds of tests including interoperability, system, integration and performance testing.

It is a textual test specification that looks similar to a traditional programming language, but provides test specific concepts. It offers different constructs to describe the test data: types, templates, variables, procedure signatures etc., and allows an easy and efficient description of complex test behaviors in terms of sequences, alternatives, and loops of stimuli and responses [2]. A TTCN-3 test specification, referred to as test suite or ATS is usually represented by a set of TTCN-3 modules. A TTCN-3 module consists of four main parts: type definitions for test data structures, templates definitions for concrete test data, function and test case definitions for test behaviors and control definitions for the execution of test cases. Templates are data structures used to define message patterns for the data sent or received.

The test specific concepts include e.g. test verdicts, matching mechanisms to compare the reactions of the SUT with an expected range of values, timer handling, distributed test components, ability to specify encoding information, synchronous and asynchronous communication, and monitoring. The communication between test components as well as the one between test components and test system interface (TSI) are realized over ports. Templates are used either to describe distinct values that are to be transmitted over ports or to evaluate if a received value matches a template specification. Data transmission directions can be defined for each port: *in* (the received data), *out* (the sent data), *inout* (the data can be both sent or received).

TTCN-3 allows the specification of dynamic and concurrent test systems by means of dynamically created test components at runtime. A system configuration is the specification of all test components, ports, connections and test system interface involved in the test system. A *system* component is generally used in each test case to specify an abstract test system interface. This abstract test system interface provides a means for test components to communicate with the system under test via the real test system interface.

In a real test system, besides abstract specification, a TTCN-3 execution environment has to be integrated so to get a complete TTCN-3 test system implementation. Concepts for the implementation of such test systems have been addressed in the context of the TTCN-3 Runtime Interface (TRI) standard [24], TTCN-3 Control Interface (TCI) [26] and outlined in [25].

2.2 Static Metrics

As previously presented, TTCN-3 core language has a lot of similarities with advanced programming languages concerning its structure. Thus, we propose a set of metrics that may be obtained starting with the language structure and particular elements described in a TTCN-3 test suite.

The idea of collecting metrics is often encountered in software engineering for estimating work volume and efforts in the time schedule. In the testing area the use of metrics is correlated with the test design and test execution results or test stopping decisions. The research challenge is determined by the interpretation of the collected values since the interpretation usually must provide as conclusion if the test meet a certain degree of quality.

The interpretation of a set of values in order to derive a global conclusion over the whole set, is achieved using *statistical theory*. The statistical theory offers mathematical instruments, methods and models that, having as input some sampling from a finite population, is able to predict the whole behavior of the entire population or draw some major conclusions about one of its attribute.

With respect to testing, the major conclusions should be related to the quality field. We foresee these estimates in terms of statistical correlations results between two or more sets of variables.

Since the quality of a test suite has to be assessed not only based on the revealed failure-density criterion, aspects of code structure are taken into consideration. The intent is the application of *statistical analysis* on the computed metrics with the aim of discovering dependency relations between two sets of values of apparently independent variables (e.g. the number of lines of code in a test case and the test case execution time). *Non linear regression correlations* [22], *Spearman and Pearson correlations* [23], etc. may be considered on the collected numerical data. This is a work that we want to investigate deeper.

However, having static metrics on the TTCN-3 code provides us also with the tester profile information and reflects his capability to organize the testing effort which influences also the testing process.

Related to the quality field, we intend to apply a statistical-based process to sets of metrics gathered from test suites corresponding to the same SUT. Following this direction, next we will introduce *static metrics for a TTCN-3 test suite* by highlighting the ones that are used also in other programming languages and the ones that are TTCN-3 specific. They can be computed only based on a single module or in combination with the other modules contained in a TTCN-3 test suite.

Static metrics similar to other programming languages

Without following a special selection method, we chose a list of metrics often employed in software development but also related to TTCN-3 code. The list is however open and we consider that later on, when more quality assumptions will be made, the list will considerably grow up.

These metrics can be subdivided into: volume metrics (in terms of LOCs, min/max number of parameters, etc.) and relational metrics (in terms of relation between two interacting entities).

The volume metrics are global indicators for the distribution effort upon the test definitions. They cannot be directly used to derive quality estimates of the test itself but they can offer useful information about the design practice (too many LOCs per test case, disproportional LOCs per test cases, etc.). Some examples are:

– lines of code: the number of all lines of code contained in a TTCN-3 test suite

- commented lines of code: the number of all commented lines of code contained in a TTCN-3 solution
- number of words: number of words contained in a test suite obtained by summing the words contained in each module
- test suite size: number of bytes for a test suite obtained by summing the size of each module
- min, max, average lines of code for a function
- min, max, average number of parameters for a function
- number of unused variables: counted with respect to their closer visibility domain (scope) e.g module, function, test case, etc.
- min, max, average cyclomatic complexity for a function: number of branches in the function

The relational metrics reflect coupling between various test behaviors (i.e. functions). Two examples are:

- min, max, average function *Fan-in* (how many entities call the function) for a test suite
- min, max, average function *Fan-out* (how many entities are called from the function) for a test suite

Static metrics TTCN-3 specific

Next, we try to identify also metrics specific to TTCN-3 based on the TTCN-3 artifacts, keywords, constructs, etc. They are in particular important because they help derive quality at a deeper level of granularity as for example: refactoring indicators, reusability, test system interface coverage etc.

We group again these metrics into volume metrics and relation metrics. The volume metrics are now more detailed since we can define them separately for each TTCN-3 structural element (test case, altstep, function, types, etc.). Some of these metrics are:

- min, max, average lines of code for a test case
- min, max, average number of parameters for a test case
- max size of modules dependency: the maximum number of imported modules contained in a module from the test suite (non-recursive computation)
- max depth of modules dependency: the max depth level of the nested modules (recursively computed depth of imported modules related to a module); this corresponds to the depth level of the module nesting tree associated to the test suite
- max depth of a type definition of a template sent via a port
- min, max, average number of parameters for a template
- min, max, average number of ports for a component
- min, max, average test case call for execution depth in a test suite: the nesting level of calling for execution in control part of a test case

Relation metrics are very important in order to extract the dependencies between behavior elements of a test suite (i.e. how many test cases reference a given function). A sample list is:

- min, max, average altstep *Fan-in* (how many entities call the altstep) for a test suite
- min, max, average altstep *Fan-out* (how many entities are called from the altstep) for a test suite
- min, max TSI ports covered by a test case (TSI ports coverage for a test case)
- min, max depth of a user-type definition: the maximum level of nesting for a TTCN-3 user-defined type (e.g. RecordType, SetOfType)
- max spreading of user-defined type: maximum number of fields of a TTCN-3 user-defined type (e.g. the direct number of fields for a RecordType, SetOfType)
- PCO (Point of Control and Observation - all interfaces with the SUT), PC (Point of Control - interfaces between Parallel Test Components) related to a test case: this reflects the interaction power of the test case to the TSI; the metric is close to the TSI port coverage metric
- min, max, average test case call for execution depth in a test suite: the nesting level of calling for execution in control part of a test case

2.3 Dynamic Metrics

The most decisive factor in establishing and analyzing the quality of a test is its capacity to reveal failures during the execution of SUT and, therefore, to expose faults that the system may hide. Based on the information recorded during or after test execution and verdict establishment, *dynamic metrics* are introduced. They provide quite useful information in order to assess the quality of testing with respect to its effectiveness of failures detection criterion.

To be noted that these indicators calculated during and post-execution time reflect quality aspects from the entire TTCN-3 test solution: test system implementation as well as ATS design. We propose the following dynamic metrics to be applied on a TTCN-3 solution (ATS and test execution implementation):

- number of passed/failed test cases after test suite execution
- number of failed verdicts divided by number of lines of codes for all test cases in a test solution
- number of failed verdicts divided by number of lines of codes for those test cases that established the fail verdicts in a test solution
- min, max, average time for the execution of a test case in a test suite (this could be done by inspecting the time stamps registered in the TTCN-3 log information)
- min, max, average number of parallel test components (PTCs) employed for the execution of a test case (the creation of PTCs is dynamically realized at runtime and depends on various additional information such as value of module parameters known only at execution time, consequently, their real number can be computed only at runtime)

Other forms of dynamic metrics should be defined for specific type of applications to be tested by carefully inspecting particular aspects such as real *inter-failure time* computation when the precision is very rigorous (real-time applications, embedded systems).

In a time evolution perspective of a test suite, statistics related to the test cases that have changed their verdict could be taken into consideration.

3 The Quality of a Test Suite

3.1 Definition

Quality should be always associated to a criteria upon it is observed. All coverage tools are designed with specific adequacy criteria in mind.

We believe that quality of a test suite should be seen as a function $f(S, I, T, C)$, where:

- S is the SUT specification given either in natural language or formal languages (UML, SDL, state machines, Markov chains, etc.)
- I is the implementation of the specification S which offers an interface to interact with
- T is the test suite whose quality is to be assessed and that will be run against the implementation and
- C is criterion upon which the quality is evaluated. A criterion is a property (most of the times also quantifiable) whose value has to be observed during test design.

We think of the specification S and implementation I as parameters of the function, and the test suite T and the criterion C as its independent variables. The mean for f(S, I, T, C) is that T satisfies C for I and S; otherwise T does not satisfy C.

An important concept is what it means for a criterion to detect a failure. A test criterion C detects a failure of an implementation I with respect to a specification S if every test suite T contains at least one test capable of identifying the failure.

Simply inspecting the test design expressed in TTCN-3 does not imply the communication with the SUT. In this case, the test quality definition introduced before may be reformulated as a function that depends only of S, T and C.

3.2 Relative vs. Absolute Quality Estimation

The analysis of the effectiveness of a test can be stated in a *relative* or *absolute* manner with respect to the selected criterion. Obviously, the *relative evaluations* have a greater significance, the evident criterion being the number of failures uncovered by means of failed verdicts establishment: the quality of a test suite is greater than the quality of another, both designed to be run against the same system, when the first test suite is capable to reveal more failures. By definition, relative means the existence of more elements to be compared, and, since the goal is to have to assess the test's quality, the test suite should be varied, that is, the reference system is the same system under test and the variants are the tests. As the Figure 1 indicates, for each version of test suite T (T1, T2, ...Tn), a specific quality of SUT is observed.

Close to the relative method for evaluation of tests quality is the idea of mutant testing where versions of T are obtained by applying mutant operators to an initial test suite [12]. Concerning the *absolute criterion*, a test could touch one percentage level between an established lower and upper threshold value. For example, if we consider the *usage of all defined variables* criterion in a TTCN-3 test suite, we state that the test has maximum quality with respect to this test efficiency criterion when all defined variables are used at least once in their visibility domain definition.

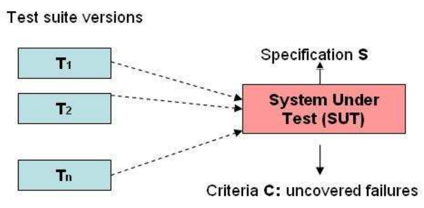


Fig. 1. Estimation of the quality in a relative manner

3.3 Test System Interface Coverage

In the particular case of a test system built upon TTCN-3 standardized test system architecture, the quality of a test has to be considered from two points of view: *abstract test suite* (ATS) and *executable test suite* (ETS) - the resulting TTCN-3 compiled code combined with specific hand-written components that make the test suite executable against a specific SUT. The invariant in this changing perspective is the SUT by means of specification S and implementation I.

Following the TTCN-3 language structure, every test suite has to represent a model of the SUT internally at the ATS level. This SUT representation is exhibited to TTCN-3 test infrastructure by means of the TSI (test system interface) definition. Therefore, from the technical point of view, the ATS comprehends basically two main parts which are considered in our work:

– SUT interface description - here are defined the types of messages and instances of messages that the SUT accepts as stimuli and the messages the SUT might replay as reaction to stimuli.

– test logic - this part expresses the logic for the communication between the test system and SUT. The test logic is here combined with data variation methods in order to apply for the same type of message various combinations of inputs.

In this test design perspective, the two parts can be correlated together in order to identify if the test logic part creates good data combinations to be applied as stimuli. It is obvious, if the whole ports and messages of the interface description are not covered, it is very unlikely that the whole behavior of the SUT is covered. Consequently, the quality check efforts should first verify if the test logic covers entirely the interface description of the SUT.

An analysis of quality of a TTCN-3 test suite can be done in a *statical manner* by inspecting the ATS without running the test against the system under test. A good correspondence between specification S and TSI, a full coverage of the interface by the test cases contained in the test suite may have a good impact in global quality estimation. The second component, ETS reflects the interaction with the system under test by means of verdicts establishment, therefore it implies a *dynamical method* in quality estimation.

3.4 Criterias

The quality estimation function is always associated with a criterion and cannot be globally evaluated but can be only compared upon a comparison criterion. Therefore, the meaning of a quality function depends on the significance of the criterion itself. In this section we discuss several criteria examples.

Test documentation. Having a well documented source code is always a sign of a good quality test. Obviously, the contrary is not necessary true. But according to particular quality views (maintainability, reusability, etc.) the documentation factor plays a major role. The more documentation we have the more maintainable the code is. As general documentation criterion we should say that a test is good if the most important structural parts (test cases, functions, defaults, components, ports, etc.) are documented. However, this criterion is relative since it bases on the presence of documentation and not on the meaning of the documentation itself.

Naming convention. Another test source code related criterion can be the naming convention. Many software design techniques recommend to follow the same naming rules in order to ease understanding of the source code among a developers team. A poor quality according to this criterion has a significant impact on the maintenance (in terms of process time) of the test suite.

Test data selection. This criterion refers to the data selection for creation of stimuli for the SUT. There are many well known methods to coupe with the data coverage (data domain partitioning, equivalence class, etc.). A good quality test in this context can be achieved if the test produces stimuli from each SUT input data domain and good combinations of data.

The time factor. This criterion refers to the development time for a test. We believe that the time factor must be a balance between the SUT complexity, test language and human resources involved in the development. It is obvious that many times a forced fast development (under pressure) may negatively influence the quality of the test with respect to other criteria (documentation, test data, naming convention, etc.).

The depth of inheritance type definition. This criterion says that a good quality is achieved when the nesting level is not deeper than a given value selected on user experience.

User satisfaction. This criterion involves the client perspective on the software product and implies that the test focuses on the aspects that offer the greatest client satisfaction.

As a general remark, we observe that one quality criterion may drive to the conclusion that a test is bad, while for the same test, but evaluated with another criterion, the quality might be good. For example, a well documented test might be a poor test in terms of number of failures found. These are only a few criteria, much more can be derived on specific domains or particular software engineering aspects.

4 Environment Metrics

According to [29], test metrics are classified into:

- metrics for assessing the testability of the software
- metrics for evaluating test cases
- metrics for calculating test costs
- metrics for measuring test coverage
- metrics for assessing test effectiveness

We concentrate in this paper on the metric categories 2, 4 and 5. This section is intended to extend the idea of test metrics in a more general panoramas and tries to introduce views from which the quality has to be regarded or is influenced.

4.1 Internal Metrics

So far, the static metrics (based on the TTCN-3 code structure) and dynamic metrics (based on test execution outcomes) were presented. They can be regarded as *internal metrics* since they are gather directly from the TTCN-3 test suite. Beside this category, we recognize further types of metrics which influence the quality of the whole test system, but these metrics are related to human factors (development efforts, testing efforts) or physical environment factors (hardware resources).

Due to the TTCN-3 format and to the extensively usage in conformance testing of telecommunication protocols, the problem of TTCN-3 tests reusability appeared. The concept of reusability applies mainly to software development area, but aspects related to how well the structure and features of TTCN-3 conform to producing reusable test software have already been investigated [27].

A form of reuse in test development process is represented by test patterns. In [28], in the context of mainly used approaches for test reuse (vertical, horizontal, historical), various categories of TTCN-3 test patterns are introduced (test pattern template, test pattern reuse, patterns in test execution, etc.).

4.2 External Metrics

The *external factors* have a very important impact on test's quality. When the system under test requires a special degree of performance such as real-time systems or long-live systems, then the test system has to assure also a special level of performance. Even if the logic and design of the test is good, the execution parameters may have a negative impact on the test quality. Test platform capabilities (operation system upon which the test system runs, memory, CPU), network overload that may affect the connectivity with the SUT, number of users, etc. are only o few examples of external factors.

4.3 Effort Metrics

A special factor is the time factor. Writing tests under pressure or trying to acquire domain-knowledge in record time is often practised, but, for sure, a deeper understanding of SUT concepts and functionality comes from a long time experience in that specific area.

Seeing testing as a part of the whole development process, one may speak not only about quality of tests but also to quality of testing process. The way the testing was planned, developed and executed (and here we mention as examples: communication between testers and developers, different programming styles and modularization, lack of synchronization of the work) may have negative impact on the whole testing process.

Some examples of effort metrics are:

- number of test cases from the entire test suite developed by a tester
- number of failures found per tester
- cost for testing process
- test efficiency defined as the number of failures found during testing reported to the number of failures found after releasing
- the time necessary to acquire domain-knowledge

5 Design ideas for a TTCN-3 metrics tool

In order to compute all above-defined metrics, there is a clear requirement for a TTCN-3 tool designated both for the abstract tests definition as well as for their execution against the SUT.

For this purpose, we have chosen the TTworkbench [16] product. The main reason behind this selection was the Eclipse-based plug-in structure of the product. The advantage of a relative easy extensibility of the test platform with new features and the employment of EMF [17] as a central repository for TTCN-3 elements, led us to the idea of providing a TTCN-3 metrics plug-in built upon TTworkbench infrastructure. This is a work-in-progress that will focus mainly on a general mechanism for computing *static metrics* starting with TTCN-3 code.

Our implementation is build upon the TTCN-3 meta-model [15]. The meta-model does not only directly reflect the structure of a TTCN-3 module but also the semantical structure of the TTCN-3 language definition. That enables our metrics collector tool to traverse TTCN-3 language elements and access the desired information.

Behind Eclipse specific aspects such as presenting the collected metrics in user-friendly way, in this case a new view, the computation process comprises two steps:

- identify all the TTCN-3 modules contained in a test suite having as input the main module of a TTCN-3 Eclipse project
- calculate the list of metrics by traversing each module

For the moment, when multiple sequential requirements for static metrics computation upon an ATS appear, there is no memory or history of the calculated values. It is possible only to see the differences between metrics gathered upon two test suites versions when these test suites are defined as separated folders in the same TTCN-3 project.

Since the execution of a test requires, besides compiled ATS, also the existence of the additional pieces in the TTCN-3 test system architecture such as Platform Adapter (PA), CoDec and SUT, we envision the implementation of *dynamic metrics analyzer* as a tool built into TTCN-3 runtime and logging interfaces. Another option would be an off-line investigation of a very detailed log file produced after test execution whose structure would have counterparts into ATS elements but also into execution-specific artifacts (e.g. time stamps).

Although the static metrics monitor is a work-in-progress tool, we have already used it for gathering basic TTCN-3 metrics applied to two test solutions: SIP test suite developed by ETSI and IMS conformance TTCN-3 test suite.

5.1 Static metrics applied to SIP test suite

The IETF Session Initiation Protocol [19] or SIP, has been developed as a control protocol for establishing, changing and tearing down multimedia sessions such as video, voice, chat, gaming between one or more endpoints in an IP network. SIP has a client-server architecture and is a request-response protocol dealing with requests from clients and responses from servers and whose implied participants are identified by SIP URLs.

ETSI STF176 is producing in the context of TIPHON (Telecommunications and Internet Protocol Harmonization Over Networks) also conformance test suites for SIP.

Starting with the available SIP-specific TTCN-3 test suite [30], we collected basic metrics that can be observed in Table 1. The first column denotes the name

Table 1. Metrics for ETSI TTCN-3 ATS applied to SIP (RFC3261)

Static metrics	Values
Number of Modules	9
Size (bytes)	2056808
Words	239115
Lines of Code	63723
Number of Functions	11
Min Lines of Code for a Function	2
Max Lines of Code for a Function	14
Average Lines of Code for a Function	3,67
Number of Test Cases	609
Min Lines of Code for a Test Case	10
Max Lines of Code for a Test Case	223
Average Lines of Code for a Test Case	66,82
Max Size of Modules Dependency	6
Min number of parameters for a function	0
Max number of parameters for a function	11
Average number of parameters for a function	1,3
Min number of parameters for a test case	0
Max number of parameters for a test case	3
Average number of parameters for a test case	1,23
Number of Component Types	2
Min number of ports for a component	1
Max number of ports for a component	5
Average number of ports for a component	3,0

of the static metric analyzed while the second one represents its value cumulated for the entire test solution.

The existence of two component types suggests the design of only one test system component. This fact shows that the test suite may uncover failures exposed only by the user interface of the SUT and it would not be able to detect problems in interfaces that may relate the SUT's subcomponents. The case of mapping multiple SUT interfaces into only one test system interface component denotes a rigid test design in terms of extendability and maintenance.

The relatively large number of parameters for a function defines a high level of test code reusability, and, therefore, increase its quality in this context. Additionally the eleven functions are not bigger than 14 LOCs which means that the test behavior was well split into smaller pieces of code.

Combining metrics offers another possibility to deduce further conclusions about the test. For example just multiplying the number of test cases with the average number of LOCs per test case gives about 10% of the total number of LOCs. That means that the coding effort focused on the data types and instances definition.

5.2 Static metrics applied to IMS test suite

The second input for our static metrics analyzer represented the conformance TTCN-3 test suite for IP Multimedia Subsystem (IMS) infrastructure provided by FOKUS Fraunhofer Institut [21]. IMS defines a generic architecture for offering Voice over IP (VoIP) and multimedia services. The VoIP implementation is based on a 3GPP standardized implementation of SIP, and runs over the standard Internet Protocol (IP). The same set of metrics have been summarized upon three different versions of IMS-specific TTCN-3 test suites as Table 2 indicates.

The test suite versions are the result of an evolution of the ATS in a interval of about 3 months. The latest three columns correspond to metrics values computed upon the three selected versions of the test suite. After a simple look into the table it is obvious that, during time, almost all metrics values increased. The increased amount of code (lines of code, bytes, words) may be explained as a natural result of increasing the number of test cases.

On the other side, the abrupt variation of the number of modules (from 5 to 18) may reflect a much better organization and structure of code as a consequence of the fact that the testers gained much more experience and knowledge of the domain. How much domain-knowledge has a tester is a factor that affects the quality of a test.

Based on the obtained values we observe (e.g by multiplying the number of test cases and the average number of lines of code for a test case) that most of the testing coding effort, expressed in terms of code size (bytes, line of codes), was done in area of test cases design.

6 Outlook

This paper proposes a set of static and dynamic metrics for TTCN-3 test suites targeting the analysis and estimation of the test's quality. A method of how to realize the static metrics analyzer upon an existing TTCN-3 tool architecture has been introduced.

Furthermore, our investigation will go into the direction of collecting dynamic metrics upon selected case studies. Independently studied, the collected numbers do not bring a consistent contribution to quality estimation. Therefore a cornerstone in our research would be to find a correlation between statically and dynamically collected values on a statistical theory basis that may promise to enable the failure prediction capacity of a test suite.

We are also thinking to extend our ideas around domain specific quality profiling since quality, in our opinion, depends very much also on categories of SUTs (automotive, telecommunication protocols, real time application, embedded systems, etc.).

One could also compare the testing systems upon the degree of tool configurability and easiness to use. The more flexibility in the configuration the test has, the less manual adaptation is required. Such factors can be also taken into account in a global view quality estimation.

Table 2. Metrics for IMS TTCN-3 test suite

Static metrics	ATS v.1	ATS v.2	ATS v.3
Number of modules	5	5	18
Size (bytes)	161464	187240	195132
Words	16381	19324	21391
Lines of Code	5801	6762	7002
Number of Functions	4	4	7
Min Lines of Code for a Function	3	3	2
Max Lines of Code for a Function	5	6	4
Average Lines of Code for a Function	3,5	4	2,83
Number of Test Cases	2	4	5
Min Lines of Code for a Test Case	91	32	48
Max Lines of Code for a Test Case	111	120	158
Average Lines of Code for a Test Case	101	82,5	119,13
Max Size of Modules Dependency	3	4	7
Min number of parameters for a function	0	0	0
Max number of parameters for a function	2	2	2
Average number of parameters for a function	1	1	0,89
Min number of parameters for a test case	0	0	0
Max number of parameters for a test case	0	0	0
Average number of parameters for a test case	0	0	0
Number of Component Types	4	6	5
Min number of ports for a component	1	1	1
Max number of ports for a component	3	3	2
Average number of ports for a component	1,75	2,17	1,6

The presented ideas are only a starting point, future work will attempt to determine in which measure the combination between the proposed metrics better assess the testing quality.

References

1. ETSI, TTCN-3 Standard Part 1: ES 201 873-1 V3.1.1 - TTCN-3 Core Language
2. J. Grabowski, D. Hogrefe, G. Rethy, I. Schieferdecker, A. Wiles, C. Willcock: "*An Introduction into the Testing and Test Control Notation (TTCN-3)*", Computer Networks, Volume 42, Issue 3 (2003), 375-403
3. Everald E. Mills: "*Software metrics*", SEI Curriculum Module SEI-CM-12-1.1, December 1988
4. Norman Fenton, Martin Nell: "*Software Metrics: Roadmap*", The Future of Software Engineering, Anthony Finkelstein (Ed.), ACM Press 2000
5. S. H. Kan, J. Parrish, D. Manlove: "*In-process metrics for software testing*", IBM Systems Journal, Vol 40, No 1, (2001)
6. Patrick J. Schroeder, Dr. Bogdan Korel: "*Maintaining the Quality of Black-Box Testing*", `http://www.stsc.hill.af.mil/crosstalk/2001/05/korel.html`
7. *The ISO 9126 Standard*: `http://www.issco.unige.ch/projects/ewg96/node13.html`

8. Steve Cornett: *Code Coverage Analysis*, http://www.bullseye.com/coverage.html, December 2005
9. Joseph R. Horgan, Saul London, Michael R. Lyu: "*Achieving software quality with testing coverage measures*", IEEE Computer Society Press, September 1994 (Vol. 27, No. 9) pp. 60-69, http://dx.doi.org/10.1109/2.312032
10. A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, M. E. Delamaro: "*Coverage testing of Java programs and components*", Elsevier North-Holland, Inc., Sci. Comput. Program., 2005 (Vol. 56, No. 1-2) pp. 60-69, http://dx.doi.org/10.1016/j.scico.2004.11.013
11. Ponrudee Netisopakul, Lee J. White, John Morris, Daniel Hoffman: "*Data Coverage Testing of Programs for Container Classes*", IEEE Computer Society, ISSRE 2002, 13th International Symposium on Software Reliability Engineering, pp 183-194, http://csdl.computer.org/comp/proceedings/issre/2002/1763/00/17630183abs.htm
12. A. Jefferson Offutt, Gregg Rothermel, Roland H. Untch, Christian Zapf: "*An Experimental Determination of Sufficient Mutant Operators*", ACM Transactions on Software Engineering and Methodology (TOSEM) 1996, Volume 5, Issue 2, pp 99 - 118,
13. Roger T. Alexander, Jeff Offutt, James M. Bieman: "*Fault Detection Capabilities of Coupling-based OO Testing*", IEEE Computer Society, ISSRE 2002, 13th International Symposium on Software Reliability Engineering, pp 207-220, http://csdl.computer.org/comp/proceedings/issre/2002/1763/00/17630207abs.htm
14. Mats Grindal, Jeff Offutt, Sten F. Andle: "*Combination testing strategies: a survey*", Software Testing, Verification and Reliability, 2005 (Vol. 15, Issue 3) pp.167-199
15. Ina Schieferdecker, George Din: "*A Meta-model for TTCN-3*", FORTE Workshops 2004, pp.366-379
16. Testing Technologies, www.testingtech.de/products/
17. www.eclipse.org/emf/
18. www.geocities.com/model_based_testing
19. Draft IETF RFC 3261, *Session Initiation Protocol (SIP)*
20. Anthony Wiles, Theofanis Vassiliou-Gioles, Scott Moseley, Sebastian Mueller: "*Experiences of Using TTCN-3 for Testing SIP and OSP*", portal.etsi.org/ptcc/downloads/TTCN3SIPOSP.pdf
21. http://www.fokus.fraunhofer.de/home/
22. en.wikipedia.org/wiki/Regression_analysis
23. en.wikipedia.org/wiki/Spearman_rank_correlation
24. ETSI: "*TTCN-3 Standard Part 5: ES 201 873-5 V3.1.1 - TTCN-3 Runtime Interface (TRI)*",
25. S. Schulz, T. Vassiliou-Gioles: "*Implementation of TTCN-3 Test Systems using the TRI*", www.testingtech.de/download/TestCom2002-TRI.pdf
26. ETSI: "*TTCN-3 Standard Part 6: ES 201 873-6 V3.1.1 - TTCN-3 Control Interface (TCI)*",
27. Pekka Ruuska, Matti Kärki: "*TTCN-3 Language Characteristics in Producing Reusable Test Software*", Springer, ICSR 2004, (Vol. 3107) pp 49-58
28. Alain Vouffo-Feudjio, Ina Schieferdecker: "*Test Patterns with TTCN-3*", Springer, FATES 2004, (Vol. 3395) pp 170-179
29. Harry M. Sneed: "*Measuring the Effectiveness of Software Testing*", SOQUA/TECOS 2004, pp 109
30. ETSI: "*RTS/MTS-00097-3r1' Work Item*", Publication (2005-07-13), Version 3.2.1

TTCP

Test and Testing Control Platform

Harald Raffelt
Oliver Rüthing
Bernhard Steffen

Ausgangspunkt

- Motivation: Popularität von TTCN-3 erhöhen durch eine freie Entwicklungsplattform
- Angeregt durch Nokia-Research Bochum 2004, insbesondere Thomas Deiß
- Rahmen: Projektgruppe
- Ziele:
  - Plattform zum Experimentieren mit der TTCN-3
  - Zunächst nur Teilsprache (message-based comm.,..)
  - Keine industrial-strength-quality

Projektgruppen

- Bestandteil der Informatikausbildung der Universität Dortmund
- Ziel: Erlernen teamorientierter, praxisnaher Projektarbeit
- Seit 1972 über 500 Projektgruppen
- Projektgruppe besteht aus 8-12 Studenten
  - Umfang: 2 Semester a 8 SWS
  - Bestandteile: Seminarphase, Projektarbeit, Zwischen- und Endbericht, Abschlusspäsentation
  - 2 Betreuer

Architektur

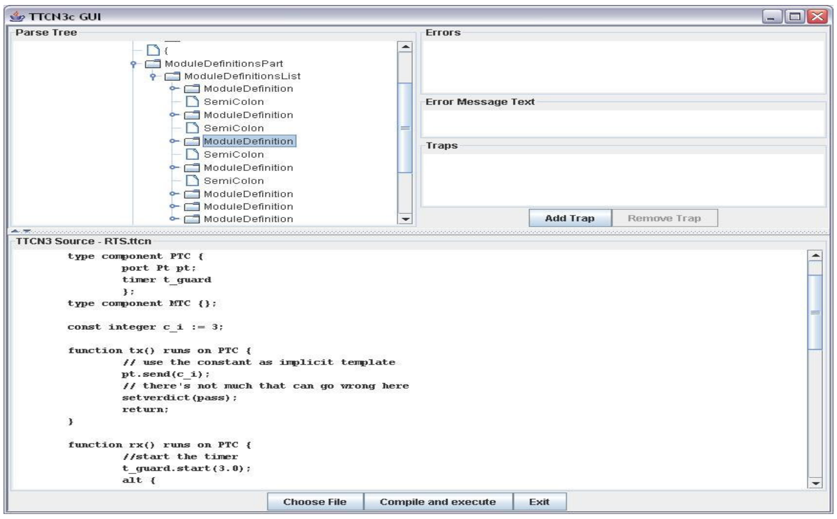
- Frontend:
  - Parser
    - Generiert durch ANTLR
    - Direkte Umsetzung der 630 EBNF-Regeln des Sprachstandards
    - Erzeugung des Sytaxbaumes und geeigneter Fehlermeldungen
  - Static Semantic Checking
    - Basierend auf Sytaxbaum des Parser
    - Überprüfung des Scopings
    - Typchecking

Architektur

- Backend
  - Grundsatzfrage: Übersetzer oder Interpretierer mit eigener VM?
  - Entscheidung der Projektgruppe
    - Übersetzung nach Java
    - Gründe:
      - Vertrautheit von Java
      - Threads
      - Effizienzgesichtspunkte
      - Vorbildcharakter TTWorkbench

Architektur

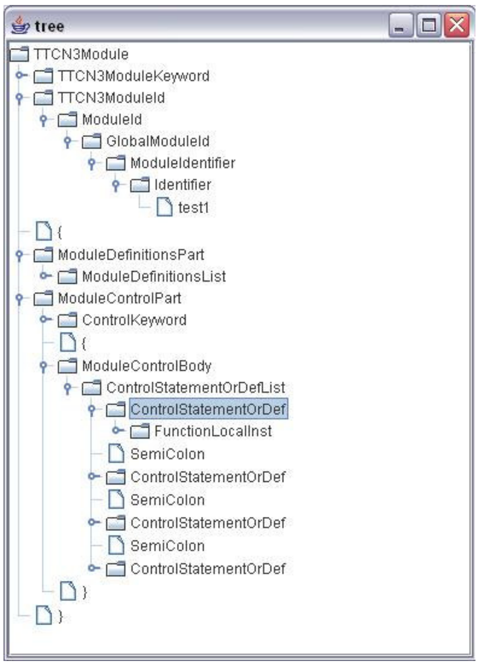
- GUI
  - Entwickelt unter Visual Editor und SWT



```
type component PTC {
        port Pt pt;
        timer t_guard
        };
type component MTC {};

const integer c_i := 3;

function tx() runs on PTC {
        // use the constant as implicit template
        pt.send(c_i);
        // there's not much that can go wrong here
        setverdict(pass);
        return;
}

function rx() runs on PTC {
        //start the timer
        t_guard.start(3.0);
        alt {
```

Parser

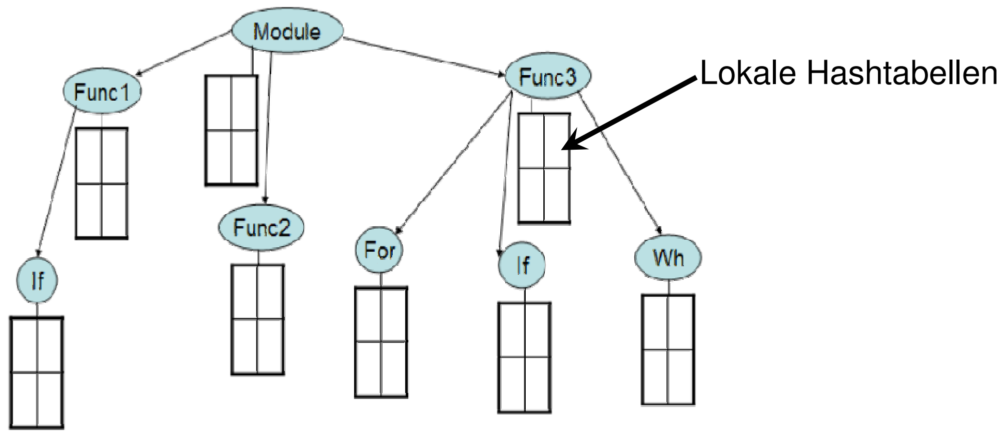
- Ergebnis der Syntaxanalyse
  - Im Erfolgsfall: Abstrakter Syntaxbaum
  - Im Fehlerfall:
    - Detailierte Fehlermeldungen
    - Wiederaufsetzen nach Fehlern



```
tree
TTCN3Module
  TTCN3ModuleKeyword
  TTCN3ModuleId
    ModuleId
      GlobalModuleId
        ModuleIdentifier
          Identifier
            test1
  {
  ModuleDefinitionsPart
    ModuleDefinitionsList
  ModuleControlPart
    ControlKeyword
    {
    ModuleControlBody
      ControlStatementOrDefList
        ControlStatementOrDef
          FunctionLocalInst
        SemiColon
        ControlStatementOrDef
        SemiColon
        ControlStatementOrDef
        SemiColon
        ControlStatementOrDef
    }
  }
```

Statische Semantik

- Hierachische Symboltabelle



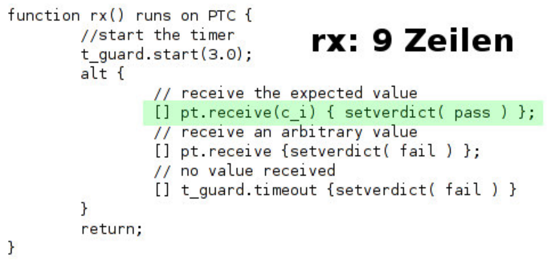
- ``Lochliste´´ für Benutzungen von später deklarierten Identifikatoren

Statische Semantik

- Weitere Herausforderungen
  - Sehr umfangreiches Typsystem
  - (Rekursive) Modulimporte
- Integration in die GUI
  - Fehlermeldungen

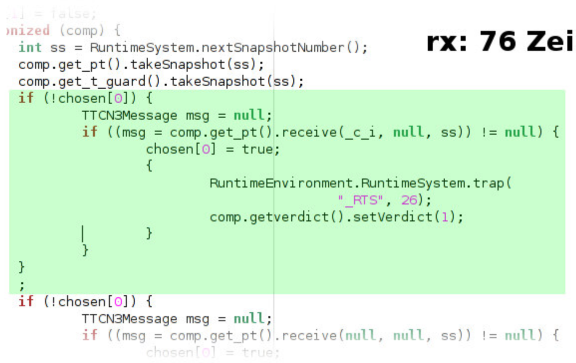
Codeerzeugung

- Syntaxorientierte Übersetzung (entlang annotiertem Syntaxbaum)



```
function rx() runs on PTC {
        //start the timer
        t_guard.start(3.0);
        alt {
                // receive the expected value
                [] pt.receive(c_i) { setverdict( pass ) };
                // receive an arbitrary value
                [] pt.receive {setverdict( fail ) };
                // no value received
                [] t_guard.timeout {setverdict( fail ) }
        }
        return;
}
```

rx: 9 Zeilen



```
nized (comp) {
  int ss = RuntimeSystem.nextSnapshotNumber();
  comp.get_pt().takeSnapshot(ss);
  comp.get_t_guard().takeSnapshot(ss);
  if (!chosen[0]) {
          TTCN3Message msg = null;
          if ((msg = comp.get_pt().receive(_c_i, null, ss)) != null) {
                  chosen[0] = true;
                  {
                          RuntimeEnvironment.RuntimeSystem.trap(
                                          "_RTS", 26);
                          comp.getverdict().setVerdict(1);
          |       }
          }
  }
  ;
  if (!chosen[0]) {
          TTCN3Message msg = null;
          if ((msg = comp.get_pt().receive(null, null, ss)) != null) {
                  chosen[0] = true;
```

rx: 76 Zeilen

Laufzeitsystem

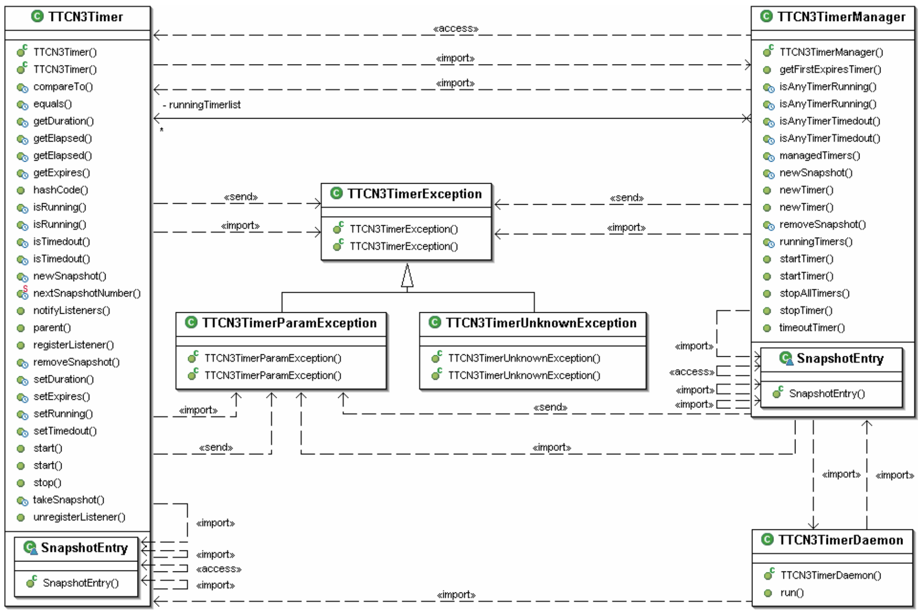
- Herausforderungen
  - Nebenläufigkeit
  - Laufzeitüberprüfungen
  - Timer
  - Ports
  - Alt-Statements
  - …

Laufzeitsystem

- Timer
  - Verwaltet durch TimerManager
  - Notify/Listener-Konzept:
    - Registrierung blockierter Alt-Statements als Listener bei nicht abgelaufenen Timern
    - Benachrichtigung aller Listener-Objekte nach Ablauf eines Timers
    - Java: Monitor für Alt-Statements von wait/notify-Operationen benutzt
  - Snapshots
    - lokal für einzelne Timer
    - global für TimerManager

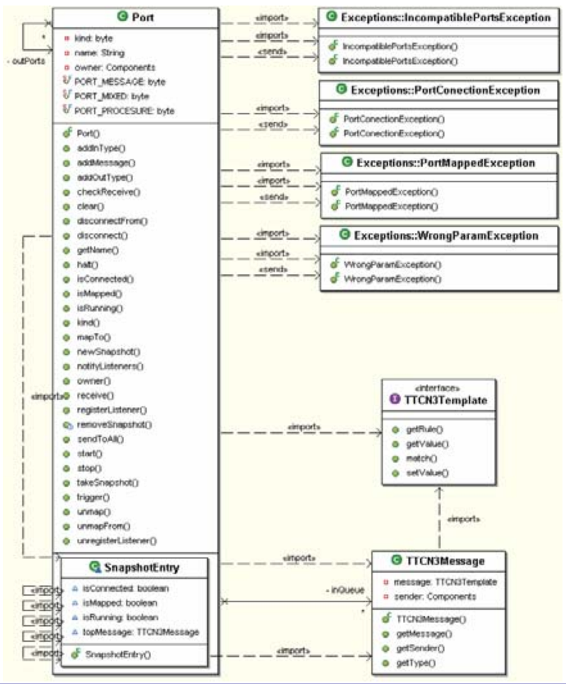
Laufzeitsystem

- Timerklassen



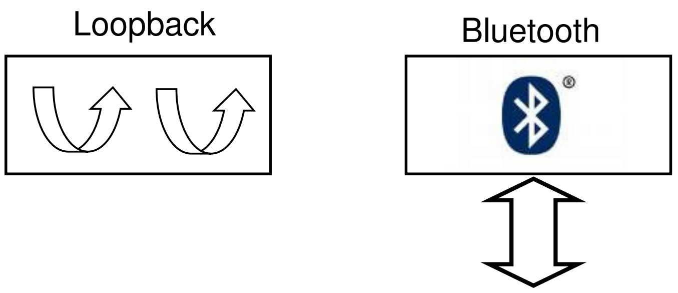
Laufzeitsystem

- Ports
  - Integriert in das Notify/Listener-Konzept
  - Snapshots



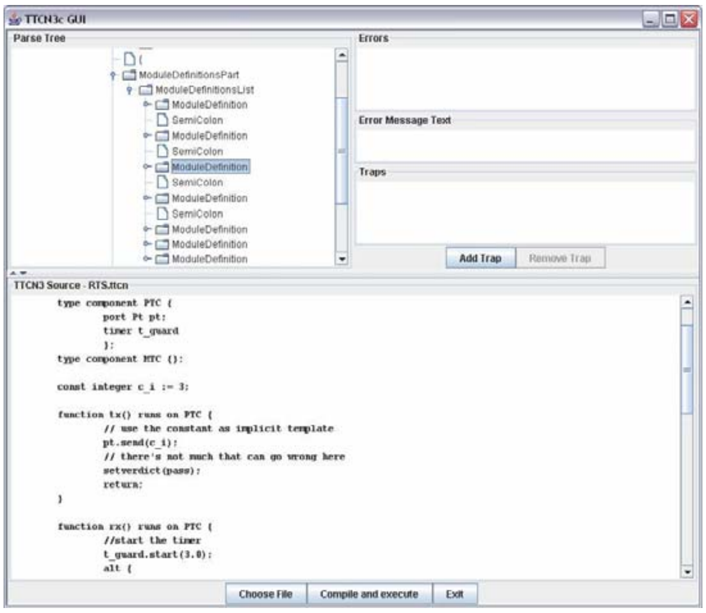
Weitere Komponenten

- SUT Adapter



Weitere Komponenten

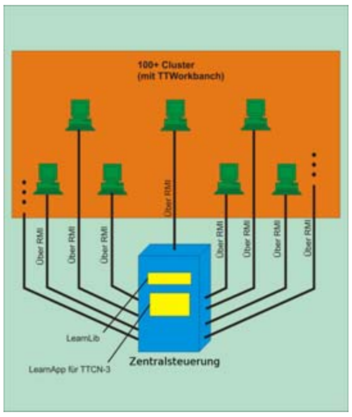
- GUI
  - Quellcode
  - Synatxbaum
  - Fehleranzeige
  - Traps



```
type component PTC {
        port Pt pt:
        timer t_guard
        }:
type component MTC {}:

const integer c_i := 3;

function tx() runs on PTC {
        // use the constant as implicit template
        pt.send(c_i):
        // there's not much that can go wrong here
        setverdict(pass):
        return;
}

function rx() runs on PTC {
        //start the timer
        t_guard.start(3.0):
        alt {
```

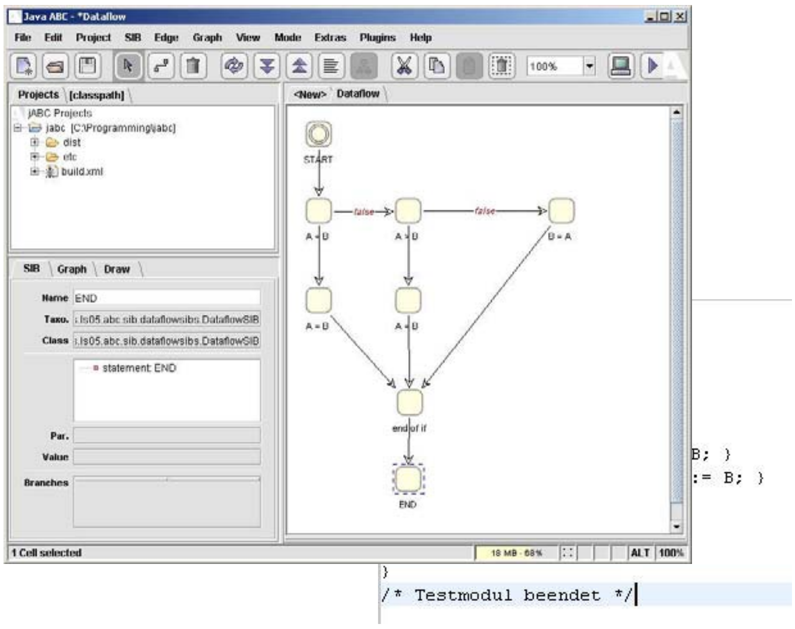
Weitere Komponenten

- Automatische Testfallgenerierung
  - Zum Testen von TTCP
  - Basierend auf LearnLib (UniDo)
  - Tool lernt automatisch reguläre Menge von Testfällen
    - Referenzimplementierung: TTWorkbech
    - Parallele Ausführung auf Rechnercluster
  - Entdeckt wurden auch Fehler in TTWorkbench



Weitere Komponenten

- Kontrollflussvisualisierer
  - JABC plugin
  - derzeit nur intraprozedural



Name END
Taxo. ...ls05.abc.sib.dataflowsibs.DataflowSIB
Class ...ls05.abc.sib.dataflowsibs.DataflowSIB
statement: END
Par.
Value
Branches

```
B; }
:= B; }
}
/* Testmodul beendet */
```

Ausblick

- Erreicht
  - Alpha-Version einer freien TTCN-3 Plattfom
- Zukunft
  - Basis für Open-Source Compiler
  - weitere JABC-Anbindung
    - Visualisierug interprozeduraler/paralleler Kontrollfluss
    - Generierung von TTCN3-Code aus SIB-basierten Workflows
  - Unbehandelte Sprachkonstrukte
  - Statische Analysen
  - Verbesserung GUI
    - Symbolischer Debugger