Georg-August-Universität
Göttingen
Zentrum für Informatik

ISSN 1612-6793
Nummer ZFI-BSC-2010-05

Bachelorarbeit

im Studiengang "Angewandte Informatik"

Detection and Analysis of Dependencies Between TTCN-3 Modules

Kathrin Becker

am Lehrstuhl für

Softwaretechnik für Verteilte Systeme

Bachelor- und Masterarbeiten
des Zentrums für Informatik
an der Georg-August-Universität Göttingen

15. April 2010

Georg-August-Universität Göttingen
Zentrum für Informatik

Goldschmidtstraße 7
37077 Göttingen
Germany

Tel. +49 (5 51) 39-17 20 10
Fax +49 (5 51) 39-1 46 93
Email office@informatik.uni-goettingen.de
WWW www.informatik.uni-goettingen.de

Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Göttingen, den 15. April 2010

Bachelor's Thesis

Detection and Analysis of Dependencies Between TTCN-3 Modules

Kathrin Becker

April 15th, 2010

Supervised by Prof. Dr. Jens Grabowski
Software Engineering for Distributed Systems Group
Institute for Computer Science
University of Göttingen, Germany

Abstract

Dependency analysis is a popular technique in software engineering to keep projects maintainable. As coupling also occurs in test suites of the testing language TTCN-3 and as size and complexity of several TTCN-3 projects have increased in the last decade, strategies for dependency detection and coupling metric generation in TTCN-3 modules had to be developed.

This thesis presents a methodology for the detection and analysis of dependencies between modules of the TTCN-3. The key characteristics of this methodology are import and reference resolution in TTCN-3 test suites combined with the building of a coupling graph detailing the gathered information. It is shown that the presented methodology allows for calculation of coupling metrics as well as further analysis based on the information contained in the coupling graph.

A *Java* implementation is provided as a proof-of-concept as well as to demonstrate the methodology's effectiveness. The implementation provides visualization of the coupling graph and calculation of several coupling metrics derived from the internal graph representation. Thus it yields useful data for the maintenance of TTCN-3 test suites.

To validate this methodology, the tool is applied to a real TTCN-3 test suite consisting of 20101 Lines of Code and 53 modules. Analysis of the tool's metrics and observation of the tool's graph visualizations provide various insights into the test suite's structure. In addition, it is demonstrated how to utilize the metrics and visualizations to decrease coupling, in this particular case by more than thirty percent.

Contents

Acronyms

TTCN-3 Test and Test Control Notation Version 3
ETSI European Telecommunications Standards Institute
ITU-T International Telecommunication Union
3GPP Third Generation Partnership Project
ITU-T International Telecommunication Union
SUT System Under Test
AST Abstract Syntax Tree
TRex TTCN-3 Refactoring and Metrics Tool
TLE Top Level Element
TLEcand Top Level Element Candidate

1 Introduction

In software engineering analysis of source code with respect to coupling metrics is a popular technique. Without making use of such metrics many projects would be hard to maintain and finally minor issues such as misplaced code elements may cause the loss of a project's former design properties.

In the telecommunication industry, standards have to be implemented correctly in order to provide interoperability between products from different vendors. For testing and verifying implementations, testing languages such as the widely used Test and Test Control Notation Version 3 (TTCN-3) have been developed (7). As the importance of information exchange has continuously increased in the last decades, so has the number of standards belonging to this domain, thus requiring a larger amount of tests for verification issues. This is reflected in the increasing size and importance of TTCN-3 projects. Hence TTCN-3 engineers have to handle coupling in test suites nowadays, but despite this, no strategies for the detection of dependencies between TTCN-3 modules have been developed so far. One may claim that platforms providing features to test a testing language would result in an endless circle. Nonetheless, even testing language projects require maintainability in order to facilitate efficient and effective working. Therefore strategies dealing with this issue are needed.

This thesis presents a methodology for the detection of dependencies between TTCN-3 modules as well as between their elements. In order to assess its effectiveness, the methodology has been transferred into a Java tool. In a further step, coupling metrics are defined and included into the tool, facilitating an effective analysis of the detected dependencies. Application of the tool to a real TTCN-3 project that consists of 20101 lines of code and 53 modules identifies several useless dependencies as well as elements that would have been better located within another module. Elimination of these useless dependencies results an decrease of coupling, in this particular case by more than thirty percent.

The remainder of this thesis is as follows: The second chapter introduces the foundations essential for the methodology and the tool's design including the TTCN-3 language, coupling metrics and abstract syntax trees. In chapter 3 the methodology will be presented and discussed. Implementation of the Java tool will be explained in chapter 4, followed by a description of its applications in chapter 5. In chapter 6 the results of a case study based on a real project, the Third Generation Partnership Project (3GPP) test suite devel-

opment project, are presented. The last chapter concludes and points at avenues for future research.

2 Foundations

For understanding how dependencies in TTCN-3 test suites can be resolved it is essential to study some basics. First, the TTCN-3 syntax and semantics will be introduced focusing on structures that are relevant for referencing. Afterwards, the Abstract Syntax Tree (AST) data structure, providing a tree representation of the TTCN-3 modules, is presented. Finally, software metrics are introduced and defined while maintaining a focus on coupling metrics applicable to analysis of dependencies between TTCN-3 modules as well as their elements.

2.1 Test and Control Notation Version 3 (TTCN-3)

In this section the TTCN-3 language will be introduced. For illustrating purposes it is supported by development of a test suite for a fictitious coffee ordering machine whose key features are described in section 2.1.1. The syntax introduced in this chapter refers to the core language standard v4.1.1 (1).

2.1.1 A Fictitious Test Suite

The fictitious test suite deals with testing the correct working of the coffee ordering machine illustrated in figure 2.1. Its intended use is facilitating the placement of orders for either Espresso or Cappuccino. Additionally, it has to inform waiters when orders are received by reporting the chosen drink and the room number from which the coffee has been ordered. The waiters will then serve the chosen drink. For circumventing unauthorized coffee consumption a special coin, which is called `caesar`, has to be inserted before orders can be placed.

The machine has two states: When it is ready for a coin to be inserted the red lamp blinks and its state is *wait*. When a coin has been inserted its state switches to *ready* causing the green lamp to blink. In this case the machine is ready to receive orders for either espresso or cappuccino. After an order has been placed by touching the corresponding lettering, its state switches back to *wait*.

This obligates testing the correct implementation of the following properties: In a first step it has to be verified that the coffee ordering machine changes its states correctly. Since

Figure 2.1: Coffee Ordering Machine

no one would wait eternally, the test only passes if the correct answer was received within 10 seconds. In a second step it has to be verified that the waiters receive the correct order and the correct location. Since we only want to verify that waiters receive the correct information, it suffices to wait for another 10 seconds whether any information has been sent to them.

2.1.2 Basic Information about the TTCN-3

TTCN-3 is an internationally standardized language for testing purposes mainly applied in the Telecommunication and Computer industry, e.g. for Telecommunication systems (ISDN, ATM) and Internet protocols (IPv6, SIP). It is currently maintained by the European Telecommunications Standards Institute (ETSI) and the International Telecommunication Union (ITU-T) and provides testing exact specifications of technical requirements. Regarding to its core functionality it contains particular features such as test cases, timers and templates. Additionally, it provides adapter functionality in order to be applied to implementations of different programming languages.

In TTCN-3 the principal building blocks, which contain the actual source code, are denoted by the term *modules*. Executable collections of modules are referred to as *test suites*.

2.1.3 Modules

Modules are the basic building units of TTCN-3 test suites and thus they contain the actual TTCN-3 code. They must have a unique name and consist of an optional definitions part as well as an optional control part.

If specified, the definitions part contains groups of TTCN-3 definitions and it may also specify their visibility to other modules (see section 2.1.6), while the control part handles the actual execution of the module. For this reason, the control part can call each element of the definitions part, thus they are all visible to the control part. Therefore, if no control part is specified, the module will be called a *library*(1).

Basic data types which can be either contained in the control part or in the definitions part are denoted by constants, identifiers, variables and comments. The only restriction is not allowing for direct placement of variables in the definitions part. Otherwise entities from the definitions- and the control part could access and change them concurrently which would cause inconsistency.

A module definition for the coffee ordering machine is specified in listing 2.1. It can be observed that it consists of a definitions part and a control part, and due to visibility properties the control part is always enclosed in the definitions part.

```
1 module coffeeOrderingTester {
2
3 // ... (some definitions, functions, etc.)
4
5  control{
6  // ... (contains calls to definitions of the definitions
7  // part and further statements specifying the execution path)
8   }
9 }
```

Listing 2.1: The module definition for the coffee ordering machine.

2.1.4 Definitions Part

The definitions part contains definitions which may be grouped for structuring purposes. Accordingly, grouping of definitions has no consequences for the actual execution of a test suite. An overview of definitions, allowed for direct placement in the definitions part, can be obtained in table 2.1. Within this thesis, a definition which can directly be located within the definitions part will be denoted by the term Top Level Element Candidate (TLEcand). However, TLEcands may be nested. Therefore, an element which is in fact directly located within the definitions part will be termed Top Level Element (TLE).

```
1  module coffeeOrderingTester {
2
3  // Since the coin type is specified in another module
4  // we have to import it
5  import from CoinSpec all;
6
7  // The coffee ordering machine accepts coins of type caesar
8  // thus we have to specify this by employing the imported coinType
9  const coinType myCoin='caesar';
10
11 // we chose not to wait for more than 10 seconds
12 const float TMAX=10*second;
13
14 // ... (further definitions such as test cases, port types
15 // applicable for message exchange, test components, etc.)
16
17  control{
18 // ...
19   }
20 }
```

Listing 2.2: Specification of the definitions part for the coffee ordering machine.

The definitions part for the coffee ordering machine is specified in listing 2.2. The `coinType` required for placing orders is imported in line 5 and a constant specifying the type accepted by the machine is located in line 9. Additionally, a constant specifying the duration of test execution is specified in line 12. This assignment reflects the testing purpose of the TTCN-3. Later, this constant will serve as a parameter for a timer that avoids test execution durations of more than 10 seconds.

2.1.5 Control Part

The control part handles the execution path of a module. Hence, it contains calls to TLEs as well as loops, conditional clauses, and data types such as constants, variables and identifiers. When calling functions, alt steps or test cases belonging to the definitions part, parameters can be assigned. Additionally, the control part can receive and store values returned from entities such as functions or test cases. In the case of test cases this will allow for supervising if they have finished successfully.

In case of the coffee ordering machine the control part (see listing 2.3) needs two variables of verdict type, which can store return values from test cases. Additionally, the control part has to start execution of the test cases. By making use of log statements notifications can be given if something went wrong. The value `pass` is a test verdict returned by a test case. Verdicts will be specified in section 2.1.8.

Language Element	Keyword	Definitions Part	Control Part
Import of definitions from other modules	import	Yes	-
Grouping of definitions	group	Yes	-
Data type definitions	type	Yes	-
Communication port definitions	port	Yes	-
Test component definitions	component	Yes	-
Constant definitions	const	Yes	Yes
Data/signature template definitions	template	Yes	Yes
Function definitions	function	Yes	-
Altstep definitions	altstep	Yes	-
Test case definitions	testcase	Yes	-

Table 2.1: Overview of specification options for TTCN-3 language elements in module definitions part and module control part.

2.1.6 Scope Rules

As it is convention in programming languages, scope rules specify ranges of visibility. In TTCN-3 there are seven different units of scope. Two of them belong to the usual structure of a TTCN-3 module, i.e. the module definitions part and the module control part. Four of them denote TLEcands which can contain further definitions, i.e. component types, functions, altsteps and testcases. They will be introduced in the next sections. The seventh unit of scope are statement blocks, which are either contained by the control part or included anywhere within TLEcands.

2.1.7 Visibility Rules

Visibility rules are a new feature, which has been included to the TTCN-3 core standard in 2009 (1). There are three different values available, i.e. public, friend, or private, assignable to TLEs. Public and private rules work as usual, but friend definitions are a particular feature of the TTCN-3. A module can be declared to be a friend of another module. In such a case, the other module sees elements having a friend visibility, too. In addition, friendships can be cyclic.

Usually, the default visibility rule is public, but for import definitions the default value is private. For group definitions a visibility rule cannot be specified thus they are always public.

In case of the coffee ordering machine the module `coffeeOrderingTester` imported `all` from `CoinSpec`. Despite this, only the `coinType` definition was needed. Thus `coinType` can be set public and the residual elements could be private.

```
1  control{
2  // define variables of verdict type, which will store
3  // return values from test cases
4    var verdicttype test1,test2,test3;
5  // start first test case (tests if coin type is accepted by the machine)
6    test1 := execute(tc_coinsPass(myCoin),TMAX);
7    if(test1==pass){
8      log("Coin was accepted.");
9      // start second test case (tests if espresso can be ordered)
10     test2 := execute(tc_coffeeOrdering('espresso'),TMAX);
11     if (test2==pass){
12       log("espresso ordering works.")
13       // start third test (tests if cappuccino can be ordered as well)
14       test3 := execute(tc_coffeeOrdering('cappuccino'),TMAX);
15     }
16   }
17   // if test3 is passed, test2 and test1 must have passed, too
18   if (test3 == pass){
19     log("Cappuccino could also be ordered. Coffee ordering machine works!")
20   }
21   // some output for the case that at least one test did not pass
22 }
```

Listing 2.3: The control part for the coffee.

Alternatively, `coffeeOrderingTester` could be declared to be friend of `CoinSpec`.

2.1.8 Top Level Elements

In the last sections elements, which can directly be placed within module scope, i.e. TLE-cands, have been mentioned. In the following they will be introduced.

Import Definitions

By making use of import definitions modules can import other modules. This provides reusability and structuring of code, e.g. by placing elements with much interdependence within a common module.

In TTCN-3, there are different options to specify import definitions. Like in other programming languages, it is possible to import a whole module. Additionally, it is possible to import only „module parameters, user defined types, signatures, constants, data templates, signature templates, functions, external functions, altsteps, [or] test cases“(7). An import can be placed anywhere in the module definitions part, but in order to let test suites be well-arranged, imported definitions can not be imported for a second time. Hence imported definitions are automatically private. For the coffee ordering machine tester an import definition has already been defined in listing 2.2.

Functions

Functions can only be placed within the module definitions part. Their keyword is `function` and within a module its name has to be unique. As conventional, a function may return a (template) value, which is either indicated by usage of the optional keyword `return` (if a non-template value is returned) or by the keyword `return template`. Additionally, functions can be called either from the control part statements or from elements of the definitions part. Since functions are well known from common programming languages, an example specifying functions for the coffee ordering machine was spared.

Altsteps

Altsteps are a particular version of functions for specification of default behaviour and structuring the alternatives of alt-statements. An altstep's body always contains so-called top alternatives whose syntax is identical to alt statements (see (1)[page 54]). Since their semantics is out of scope, they will be treated as if they were functions within this thesis.

Port Types

Port types are interfaces for data exchange between a test suite and a system under test as well as within a test suite. They have to be defined directly within the definitions part and are not allowed to be defined within other TLEs. Ports are either uni- or bidirectional and can be either message based or procedure based depending on their data exchanging behaviour. If the direction is set to `in`, the port will only permit outgoing messages. If it is set to `out`, the port will serve for message reception thus it receives messages containing replies or exceptions from outside. If the keyword is `inout`, usage of both directions will be permitted.

```
1  // define port types for message exchange with the coffee ordering
2  // machine:
3  // define ports for submitting room number and coffee type
4  type port CharstringInputPortType message { in charstring }
5  type port CharstringOutputPortType message { out charstring }
6
7  //define ports for coin insertion
8  type port coinTypePortType message { in coinType }
9
10 //define port for LED-lamp
11 type port IntegerOutputPortType message {out integer}
```

Listing 2.4: Definition of port types for the coffee ordering machine.

In listing 2.4 a port type is specified for the coffee ordering machine test. The order and the status of the LED will be submitted via a standard port type accepting either character strings or integers. The coin insertion makes use of a port of `coinType` imported from `CoinSpec`.

Component Types

Component Types are environments, which allow for sharing of certain entities such as ports, variables, constants or timers. Since variables cannot be placed directly within the definitions part, component types provide an alternative for sharing them. Hence entities such as functions or test cases can be defined to *run on* a certain test component, if they need common access to ports or variables. Additionally, component types can employ ports for message exchange with other modules or even with a System Under Test (SUT), which may be an external implementation that has to be tested. Since test cases cannot utilize ports on their own, they always have to run on at least one component type. A component type for the coffee ordering machine test is specified in listing 2.5.

```
1 // we define a component type on which test cases will run
2 type component mc_coffeeTester{
3   port coinType coinsIn;
4   port CharstringOutputPortType orderOut;
5   port CharstringInputPortType orderIn;
6   port IntegerOutputPortType
7   // ... (definition of variables, timers, ...)
8 }
```

Listing 2.5: Specification of a component type for testing the coffee ordering machine.

Test Cases

Test cases are particular functions reflecting the testing issues of the TTCN-3. They have particular return values of `verdicttype` and with respect to the outcome of a test, the return value can have particular values: Per default, it is set to `none`. If a test passes, its value will be `pass`. If the test has finished without a definite result, its value will be `inconc`. This can happen, if a timer caused an abortion of the test case execution after a certain amount of time. If a test failed, the returned value will be `fail`, and if an error has occurred during test case execution, the verdict `error` will be returned.

Test cases have to be started via the `execute` statement, and as it has been mentioned in section 2.1.8, they always have to run on at least one component type. The test cases for the coffee ordering machine are specified in listing 2.6.

```
1  // test case for coin insertion
2  testcase tc_coinPass(coinType coin) runs on mc_coffeeTester{
3    // ... (test if coin is accepted and led state switches)
4  }
5
6  // test case for the actual ordering
7  testcase tc_coffeeOrdering(charstring coffeeType) runs on mc_coffeeTester{
8  timer t;
9  // for each available room number:
10 t.start(TMAX);
11       alt {
12         [] // ... test if drink of coffeeType can be ordered
13          }
14          {
15         [] // ... test if information for waiters is correct
16          }
17          [] t.timeout {
18          }
19       }
20       // ... (set verdicts)
21 }
```

Listing 2.6: Test cases and a component type for the coffee ordering machine.

Further Top Level Elements

Besides elements that have been mentioned so far, there are some other elements which can directly be located directly within the definitions part, too. Such elements are for instance constants, data definitions, signatures, and templates. However, their actual functioning is out of scope and thus they will be regarded as further *TLEcands* within this thesis.

2.2 Abstract Syntax Trees

In this section ASTs are introduced. At first, some basic information is given. Afterwards, a more precise description of ASTs for TTCN-3 modules is presented.

2.2.1 Basic Information About ASTs

An AST is a tree representation of the abstract syntactic structure of source code. It is built during compilation when syntax analysis is performed. It is less concrete than a parse tree due to not representing each detail of the source code (6). For instance, the location of (superfluous) parentheses in the source code does not have to be reflected within the tree. In addition, an AST can represent syntactic constructs such as `if-else` conditions as one node with two branches (6).

The nodes of an AST usually represent semantic attributes of the source code. For example, expressions could be represented by an *ExpressionNode*, declarations by a *DeclarationNode* and Identifiers by an *IdentifierNode* (3).

2.2.2 ASTs for TTCN-3 Modules

A parser providing creation of ASTs of TTCN-3 modules has been developed by Wei Zhao in 2005 (9). This parser was added to the TTCN-3 Refactoring and Metrics Tool (TRex)[1] which is an open source Eclipse Plugin[2] providing IDE functionality for the TTCN-3 core notation. This reflects, that ASTs also find application besides parsing. Due to containing essential information of the parsed source code, in TRex ASTs are employed for calculation of software metrics and refactoring purposes.

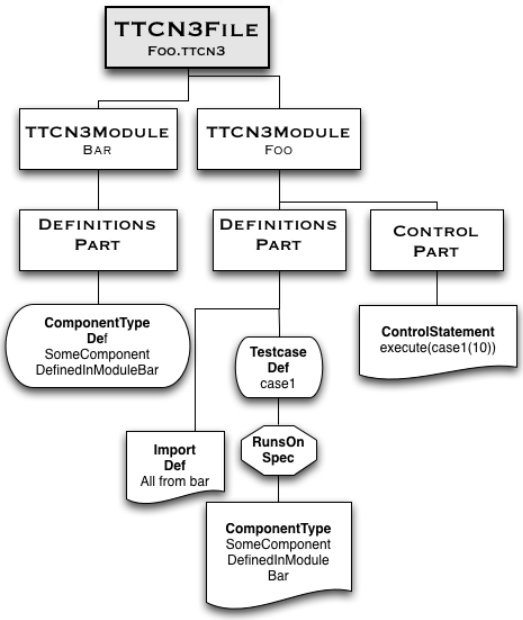

Figure 2.2: A simplified AST of a TTCN-3 file, which contains the two modules Foo and Bar.

A very simple example of an AST representing a TTCN-3 module is given in figure 2.2. Three node types can be observed: Rectangles represent structural elements such as the

[1] http://www.trex.informatik.uni-goettingen.de/trac
[2] http://www.eclipse.org

TTCN-3 module, its definitions part or its control part. Rounded boxes represent definitions and flow chart documents refer to references.

In fact, there are more than three node types. More precisely, for almost each semantic attribute of the TTCN-3 a particular node type exists. Accordingly, there are node types such as *Identifier Nodes*, *Function Definition Nodes*, *Altstep Definition Nodes*, *Control Statement Nodes*, *Function References Nodes* as well as several others. However, within this thesis only the existence of different node types will be a concern.

2.3 Software Metrics

In this section software metrics will be introduced. At first, some general information will be given thus explaining, why they are so important in software engineering and what can be achieved by employing them. Afterwards, the metrics utilized within the following chapters will be defined.

2.3.1 Source Code Measurements in Software Engineering

In general, metrics define a mapping of a „particular measurable entity to a numerical value“(4) thus helping to sum up particular aspects of things and to detect „outliers in large amounts of data “(4). Additionally, metrics can be used for development of visualizations such as diagrams or charts. Since humans are trained in understanding signs and pictures, this supports understanding and identifying of „hidden aspects of large software “(4).

In software engineering, maintenance of the product's source code is required as well as management of resources and processes associated with the development processes. Each of them needs to be controlled in order to provide efficient and effective working as well as to produce products not lacking quality. For each of those topics metrics can be defined thus they can be classified as *process metrics* , *product metrics* and *resource metrics* (2).

Within this thesis, process metrics and resource metrics will not be a concern thus focusing on product metrics. They usually describe attributes or entities of the product that will be finally delivered to the customer. Some properties, such as its size, are measurable purely in terms of the product. These metrics are referred to as *internal* product metrics. In contrast to this, attributes such as „usability, integrity, efficiency, testability, portability, and interoperability “are termed *external* product metrics because of requiring information not directly obtainable from the product (2). However, since this thesis deals with the analysis of source code, only internal product metrics will be a concern.

According to (4), there are three main topics internal product metrics can deal with when using object oriented designs: Size, coupling and inheritance. Since TTCN-3 is not an ob-

ject oriented language, the third one dealing with inheritance will not be a concern within this thesis. The first topic covers size and complexity measures, including metrics such as a project's *Lines of Code*, its *Number of Operations*, its *Number of Modules* or complexity measures such as the *Operation Complexity*. As this thesis deals with the detection and analysis of dependencies, size metrics will not be a concern as well. Hence only the second class of internal product metrics, i.e. coupling, will be relevant. Due to data encapsulation, different modules of the TTCN-3 collaborate during runtime. Measures can describe attributes regarding the question how many modules or classes have to exchange data or to which extent they reference each other. Additionally, measurements can focus on the question to which extent elements encapsulated within the same module exchange information. This aspect is called *Cohesion*.

Coupling Metrics

As already discussed, coupling specifies the interdependence between modules. It can be measured on different levels: On the one hand, the extent of interdependence in a whole system can be summed up. On the other hand, modules can be investigated or, since their extent of interdependence results from code elements referencing each other, coupling can also be measured for code elements. In figure 2.3 a code element is illustrated having incoming as well as outgoing references. This reflects that coupling can be classified to be either efferent or respectively afferent (8).

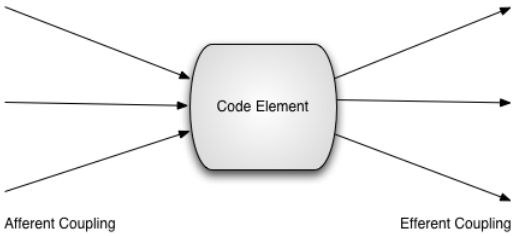

Figure 2.3: Visualization of the general coupling classifications of a code element, i.e. afferent and efferent coupling which is denoted by either incoming or outgoing arrows.

Coupling metrics are often visualized as a graph (4). In case of TTCN-3, its nodes can represent modules as well as TLEs, and the edges can visualize to which extent they are coupled.

Since elements usually reference elements within their own module as well as elements of other modules, the external usage of a module can be examined on three levels: Usage

inside the parent module, usage by elements from external modules and the whole external usage. Accordingly, it is sometimes not helpful to distinguish if references have their origin in foreign modules or within the own module and due to this there is no strict distinction between cohesion and coupling (8).

2.3.2 A Selection of Software Metrics

In this section metrics will be defined and introduced explicitly. Most of them will be employed in the following chapters. Accordingly, they will be defined with respect to the TTCN-3 syntax and semantics. As already discussed, each metrics discussed within this section can be classified as an internal product metric, or more precisely as a coupling metric.

Fan In and Fan Out Metrics

The metrics *Fan In* and *Fan Out* specify measures for afferent and efferent coupling. In this thesis, they will be defined for modules as well as for TLEs.

Fan In and Fan Out Metrics of Modules The Fan In of a module denotes the amount of modules by which a module is referenced thus it specifies afferent coupling and provides information about a module's external usage.

The Fan Out of a module denotes the number of modules a module makes use of via references thus it specifies efferent coupling and provides information about the extent to which a module depends on other modules.

Fan in and Fan Our of Top Level Elements The Fan In of a TLE denotes the number of TLEs it is referenced by. Consequently, it specifies a code element's external usage. The value of the Fan In may result either from TLEs located in other modules or from TLEs located within the same module. For this reason, this metric can be further partitioned into *Internal Fan In* and *External Fan In*.

The Fan Out of a TLE denotes the number of references a TLE has to other TLEs. Hence, it reflects a TLEs external dependency. Since the TLEs can either reference to TLEs within its own module or to TLEs from other modules, the metric can be partitioned into the metrics *Internal Fan Out* and *External Fan Out*.

Metrics with Respect to Import Relations

If references from one module to another exist, the referenced module must be imported. However, if a module is imported, references do not need to be there. In contrast to the Fan In of a module the *Number of Imports* specifies the number of imported modules. Additionally, the metric *Number of Useless Imports* denotes the difference of the Fan In and the Number of Imports, thus it specifies the number of imported modules to which no reference exists.

Further Metrics for Top Level Elements

For TLEs, it may be a concern to find out if they are located within a proper module. If they are badly located, encapsulation could be improved by moving them to the right place.

To identify elements, which are badly located with respect to coupling, the metric *External Usage Indicator* can be used. This metric is defined as the difference of two other metrics, i.e. the difference of the maximum amount of references to an external module minus the element's Internal Fan In. If this value is positive, one module exists whose elements are referenced more often than internal elements.

2.3.3 Visualizations of Metrics

As already mentioned, humans are trained to understand signs and pictures (4) and for this reason metrics are often used to create visualizations such as diagrams or charts, which allow for identification of outliers.

For visualization of coupling metrics, usage of graphs can be a good choice (4). Incoming edges could for instance represent a module's or TLE's Fan In and thus, the graph would give a visualization of the examined element's external usage. Accordingly, outgoing edges of a graph can also refer to Fan Out metrics. In this case the graph would illustrate its elements' external dependencies. In order to investigate particular attributes, partial graphs can also be defined. They could for instance only portray certain dependencies, such as incoming edges of modules containing references. In this thesis, graph visualizations will be utilized to allow for detailed visualization of dependencies of TTCN-3 modules.

3 Methodology and Design of the Tool

This chapter deals with the methodology for the detection and analysis of dependencies between TTCN-3 modules and their elements. First of all, a problem statement is given. Within this statement, related work is discussed and afterwards the importance of dependency detection in TTCN-3 test suites is emphasized. Afterwards, the actual methodology is presented.

3.1 Problem Statement

In the last decades, requirements on testing products belonging to the telecommunication or computer industry have increased (7). In the case of the testing language TTCN-3, this resulted an increase of a test suite's average size and complexity (5). Hence, problems such as handling of coupling and cohesion, which are well-known from common programming languages, also occurred in TTCN-3 test suites.

A first approach dealing with the increasing size and complexity of test suites is described in (5). In order to increase a test suite's „maintainability, changeability and analyseability "(5), in this work a methodology was developed for calculation of size and complexity metrics as well as for refactoring purposes. Additionally, the developed methodology was transferred into the open source Eclipse Plugin TRex and applied to real test suite development projects. In case of the ETSI IPv6 test suite development project version 1.1, they were for instance able to reduce the metric Lines of Code about thirteen percent and the metric Number of Templates about almost fifty percent. Even though this emphasizes the need of methodologies and tools to support maintenance TTCN-3 test suites, research did not focus on the detection of dependencies.

As it is convention in software engineering, TTCN-3 engineers encapsulate source code in different modules in order to provide reusability and scalability. Hence, at run time code elements of different modules have to access each other via references. For this reason, maintainability of source code can be increased by calculating metrics indicating to which extent code elements are used within a test suite. By employing coupling metrics to detect code elements encapsulated in a disadvantageous module, analyseability and thus maintainability can be increased. Additionally, through dependency detection the whole

structure of a project can be analysed. Therefore, dependency detection and coupling metrics are an important issue for maintenance of test suites.

3.2 Methodology for the Resolution of Dependencies

In this section the methodology for the resolution of dependencies is studied: The first matter is analysis of the kinds and origins of dependencies, followed by a discussion of the requirements for managing the detected information. Afterwards, it is presented how dependencies can be resolved in TTCN-3 modules.

3.2.1 Analysis of the Kinds and Origins of Dependencies

As studied in section 2.1, TTCN-3 test suites consist of modules. If a module references another one, this module has to be imported. Due to this, from the pure existence of an import it can not be concluded that the corresponding module is also referenced. This fact is studied more detailed later in this section. At this point, it suffices to note that imports are the weakest kind of dependency occurring between modules. Figure 3.1 illustrates dependencies resulting from imports. Modules are represented as blue *M*-labeled spheres having imports represented by arrows among one another.

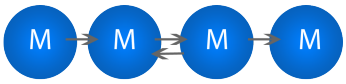

Figure 3.1: Illustration of modules having import dependencies.

A scenario depicting imports as well as references is shown in figure 3.2. In this figure, modules are illustrated together with TLEs dangling on the module spheres. References between the TLEs as well as imports between the modules are represented by arrows. In general, the TLEs do not reference each other directly. Instead, they cover referencing elements. However, due to nesting it would be hard to examine such elements directly. Therefore, references will be assumed to be directly between TLEs.

Letting the leftmost module be denoted as module number one and the others as the second, third and fourth, it can be obtained that no reference exists from a TLE of the second module to one of the third module. Despite this, an import exists between the modules. Hence, this import causes a useless dependency. Additionally, figure 3.2 illustrates references between TLEs located within module one and module four. Such references are

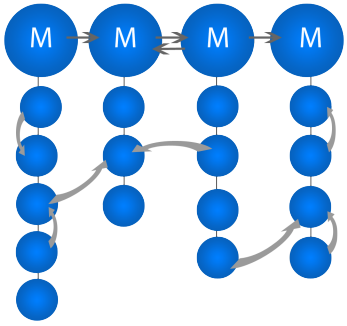

Figure 3.2: Illustration of imports between modules and references between their TLEs. References and imports are represented by arrows, modules and TLEs as blue spheres.

internal ones and reflect the cohesion of those modules. Since the TLEs of the third module, which reference either a TLE of the second or the fourth module, are not referenced internally, they could also be located within modules they reference. In this case, cohesion would be increased and coupling would be decreased.

3.2.2 Management of Detected Dependencies

In the last section it was analysed where dependencies may originate from and which kinds of dependencies exist. Before studying how to actually detect dependencies, it must be discussed how to classify and manage them.

Since the spheres, which represent modules in the two figures of the last section, can be referred to as nodes and as the arrows can be referred to as directed edges, a graph such as the one given in figure 3.3 can be used to maintain the detected information. For the remainder of this thesis, this graph will be termed *coupling graph*.

Requirements of the Graph Module nodes must be labeled in order to distinguish them from TLE nodes. Additionally, both node types have to be identifiable. Since modules have unique names within a test suite, their name can be assigned as an ID to the appropriate node. Despite this, TLEs do not have unique names within a test suite. Hence, before assigning the TLE name, a unique ID has to be created by appending the module name.

Since different kinds of dependencies may exist between the nodes, different kinds of edges are needed: Between module nodes, two kinds of directed edges will be needed to distinguish useless import from imports which are needed. Additionally, a third type of directed edges is needed to represent references. Finally, edges allowing for assignment of

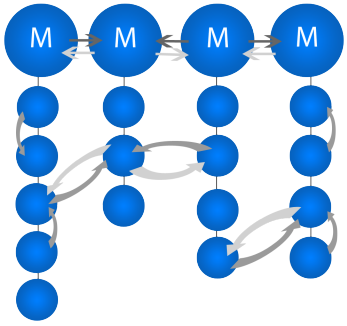

Figure 3.3: Illustration of a graph which can manage the detected dependencies. Dark arrows denote references between elements and accordingly needed imports between modules, light coloured arrows denote a imported by relation.

TLE nodes to module nodes are needed. Since the coupling graph shall be used for the definition of coupling metrics, for each edge an additional one with the inverse direction has to be added, too. While this allows easy and efficient calculation, it increases the demands on the actual implementation.

3.2.3 Detection of Dependencies

For the detection and resolution, dependencies resulting from import definitions as well as from references are considered. In both cases, the AST, which was introduced in chapter 2.2 and which is also used in TRex, utilized.

By examining a small test suite in combination with the appropriate AST, it is discussed how information can be gathered from the AST. In addition, it is pointed out how to create the coupling graph's nodes from information provided by the AST. Then, in a further step it is discussed how to resolve import definitions and how to resolve references.

How to Employ the AST

In order to investigate how to make use of an AST for the resolution of dependencies, the test suite given in listing 3.1 will be examined. It consists of two modules, i.e. module `Foo` and module `Bar`. Module `Foo` imports everything from module `Bar` and additionally, it references the two TLEs of module `Bar`. In contrast, module `Bar` does neither import module `Foo` nor does it reference its elements.

An AST representation of the two modules is given in figure 3.4, in which nodes representing references are highlighted by an orange background. Import definitions also have

```
1  module Bar {
2     function y()
3     {
4        /* ... */
5     }
6
7     type component SomeComponentDefinedInModuleBar {
8        /* ... */
9     }
10 }
11 module Foo {
12    import from bar all;
13
14    testcase case1() runs on SomeComponentDefinedInModuleBar
15    {
16       y();
17    }
18 }
```

Listing 3.1: A file containing the TTCN-3 modules Foo and Bar..

an orange background, as these definitions cause dependencies.

Before the analysis of the AST with respect to dependency-detection can start, the coupling graph's nodes have to be created from the information provided by the AST. Otherwise no edges could be added when imports or references are found.

To create nodes representing modules, the AST has to be traversed for module IDs. In order to create TLE nodes, the AST has to be traversed for nodes representing definitions. However, only few of them will eventually be relevant due to nesting.

Resolution of Import Definitions

In order to resolve and store import definitions within a test suite, two kinds of information are needed, i.e. the ID of the module which contains the import definition as well as the ID of the module, which is imported. Hence, for obtaining such information, each module's AST has to be traversed with respect to import definitions and module IDs. If an import definition is detected, the ID of the imported module can be retrieved by traversing its branch. The leaf node will then contain the appropriate information. After this, the detected module IDs have to be used in order to detect the appropriate nodes within the coupling graph. Finally, the two edges can be added.

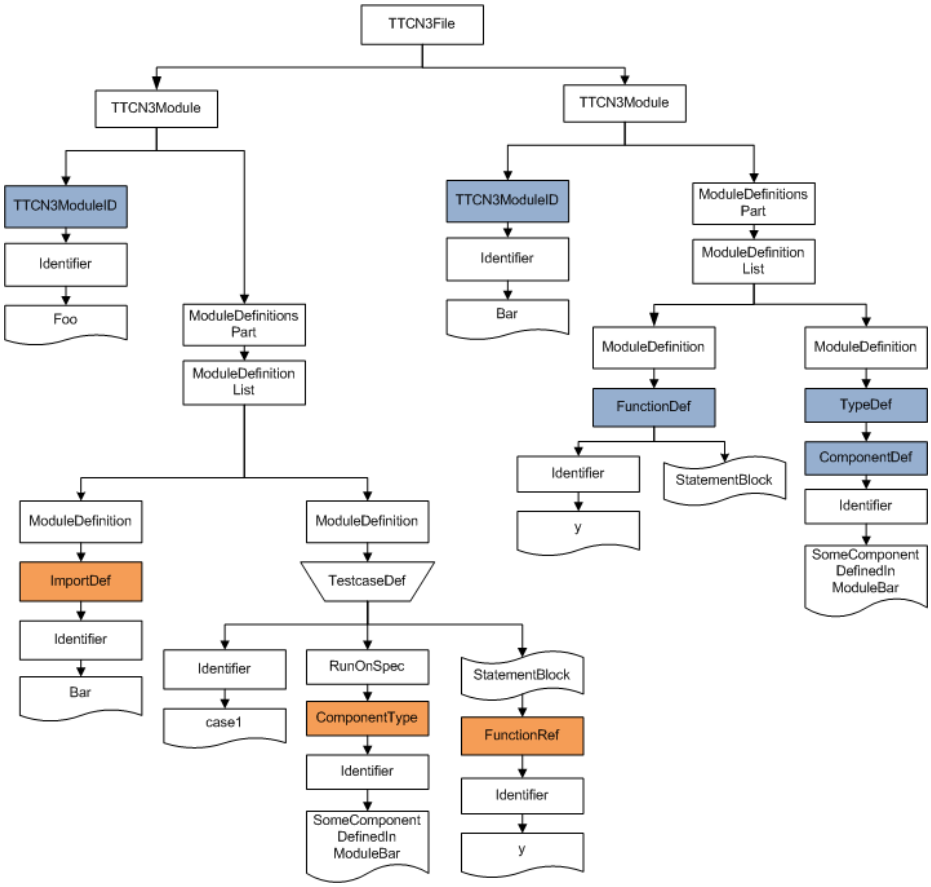

Figure 3.4: AST for module Foo and module Bar. Nodes representing reference calls have an orange background, nodes representing definitions have a blue background.

Resolution of References

In order to resolve references, several things have to be done. At first, each node representing references has to be found. Despite the fact, that some nodes representing references are already illustrated in figure 3.4, in a first step the relevant node types have to be discussed in detail. Then, in a second step it is studied, how the referenced element can be found. Finally, it is examined how the enclosing TLEs can be detected.

Node Types covering References AST nodes, which represent references, are highlighted by an appended *Ref*. For example, if a function is referenced, the AST will contain a node labeled as *FunctionRef*. Most elements, which can be referenced, are covered by the term

TLE thus requiring detection of *TLE~Ref* nodes, where *TLE~* has to be replaced by the actual TLE name. An exception are imports, which cannot be referenced. A further exception are variables, which cannot directly be placed in the definitions part. Since they can be referenced, nodes labeled as *VaraibleRef* will also be relevant. A further exception results from the manner in which component types are referenced. They are not referenced directly, instead, the *runs on* specification exists (see figure 3.4). Therefore, nodes representing functions, alt steps or test cases running on a component type will contain a child with the label *RunsOnSpec*. Since such a node also represents a reference call, it must be also considered.

Resolution and Storage of References Since the references will be added as edges between TLE nodes within the graph, solely detection of the elements, which contain references, would be insufficient. Instead, after the detection of a reference, it has to be ensured that the appropriate TLEs are found. As already discussed, TLE nodes of the coupling graph are identified by a combination of the TLE name and the corresponding module ID. Accordingly, for the referenced element both information have to be resolved. In contrast, for the referencing element it suffices to resolve the TLE and to traverse the AST to get the module ID.

To perform the actual resolutions, the first thing to do is traversing each module's AST with respect to nodes representing references. If such a node is found, the actual resolution starts. Hence, in figure 3.4 resolution would either start at the node referencing the component type or at the node representing the function reference. In both cases, the ID of the referenced element can be obtained by traversing to the leaves. However, since an ID of an element is not unique within the whole test suite, the detected ID does not suffice for storing resolutions in the coupling graph. In addition, the referenced element can also be nested and thus it may not represent a TLE. For this reason, particular functionality provided by the TRex AST implementation has to be used in order to identify the appropriate module ID and the TLE ID. A detailed description how to perform this resolution will be given in the next chapter. As soon as the referenced node's TLE and module ID of the two analysed elements are resolved, the TLE of the referencing element has to be found. Its module ID can be obtained in its AST by re-traversing to the top-most definition. In addition, the module ID can also be gathered by traversing the AST. Finally, both TLE nodes can be identified in the coupling graph and the appropriate edges can be added.

4 Implementation

As a proof-of-concept of the methodology presented in chapter 3 a draft implementation named *TTPendency* has been written and applied to real test suites. It is implemented in Java to facilitate usage of already existing functionality provided by the open source Eclipse Plugin TRex. Additionally, a command line interface for *TTPendency* is available.

This chapter starts giving an overview of the packages *TTPendency* consists of. After this, the execution path of the tool is examined, followed by a description of the tool's internal representation of the coupling graph, which has been introduced in the foregoing chapter. Then, the resolutions of imports as well as references are explained. The chapter closes with an explanation of the calculation and output of the metrics *TTPendency* provides.

4.1 Design of the Tool

In this section the design of *TTPendency* will be explained, starting by examining the UML diagram given in figure 4.1. By convention, import-dependencies are illustrated as dashed directed edges pointing to the imported package.

The responsibilities of the packages are as follows: The Input and Output package handle the user interaction. Both are imported by the main package, which controls the execution path of the tool. Additionally, the input package is imported by the metrics package to provide user specific calculation of metrics. The output package imports the input package to provide user specific output and additionally, it accesses the metrics and graph package to get the data for the outputs. The visitor packages handles traversals of the AST and contains methods to resolve imports and references of TTCN-3 modules. The graph package defines a class node, which is needed for representing the graph. Finally, the metrics package handles calculation of the coupling metrics.

4.2 Graph Representation and Management of the Observed Information

As discussed in the last chapter, the detected information is stored within a graph, which is termed coupling graph. In *TTPendency* this graph consists of instances of `Node` (from

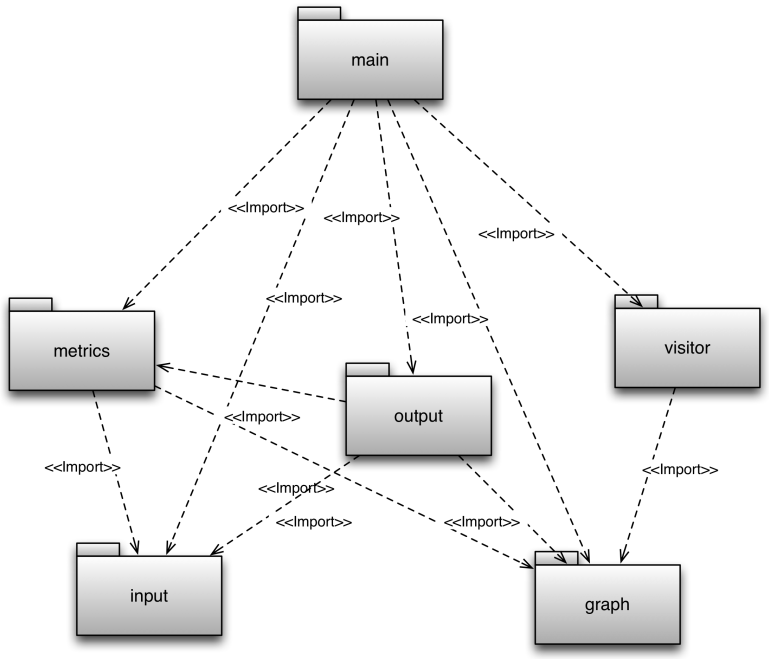

Figure 4.1: UML Package Diagram of TTependency.

the graph package). The module nodes are collected in a hash map, which contains tuples consisting of a String representation of the unique module ID and a reference to the respective node. In addition, each node contains a secondary Hash Map of the same data types. If a node represents a module, this map will contain the IDs of its TLEs paired with a reference to the respective node. In contrast, in case of TLEs this map is empty. Instead, each TLE node contains a reference to the node representing the module it is contained by. Additionally, the edges which represent the detected dependencies are also represented by references and stored in a list of `Node`.

4.3 Execution Path of the Tool

The execution path of the tool is illustrated in the sequence diagram given in figure 4.2. The `Main` class starts execution by requiring the user input from the `InputReader` (input package). Afterwards, the `Helper` class (main package) is called. It builds the coupling graph's nodes by requiring an AST for each module and in order to add the appropriate

edges (references), the `Helper` calls the `ImportVisitor` and the `ReferenceVisitor`, which then resolve and store the appropriate references. As soon as the internal graph is built, the `MetricsCalculator` (metrics package) is called to calculate the coupling metrics. Afterwards, the `Exporter` (output package) is called. It outputs the tool's metrics and the graph visualization into CSV- and DOT-files[1].

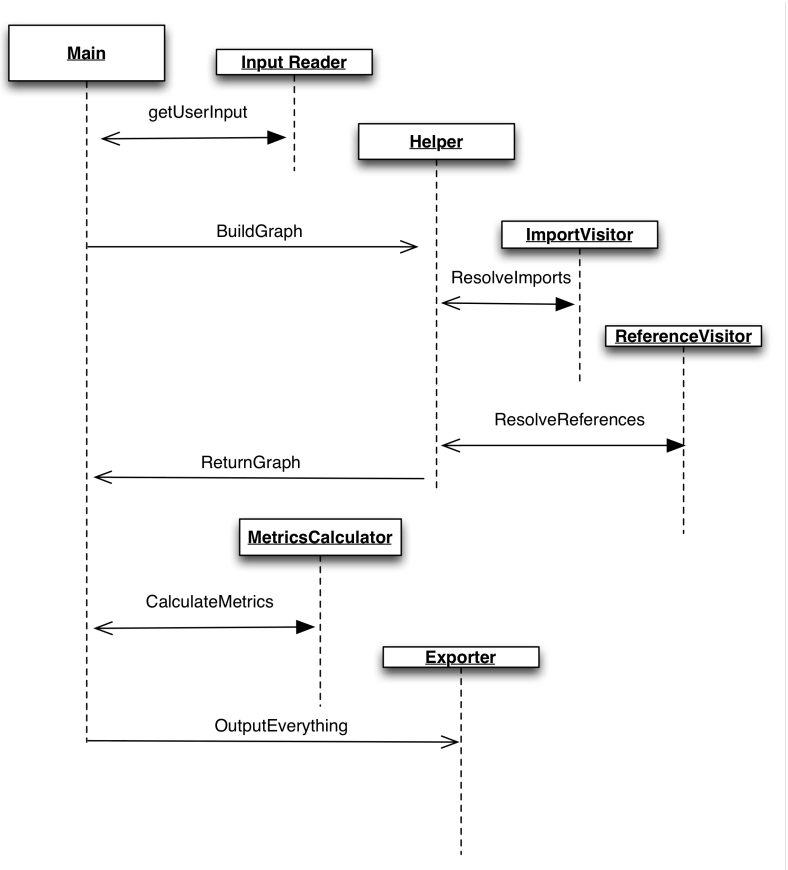

Figure 4.2: UML Sequence Diagram of TTependency.

4.3.1 Traversal of the AST

In order to traverse the AST, a visitor pattern can be used. In general, visitor patterns allow separating algorithms from the object structure on which they operate. Therefore, opera-

[1] `http://www.graphviz.org`

tions can be added to the object structure without modifying them. In case of *TTPendency*, usage of a visitor allows for performing actions at certain nodes without touching the AST implementation. For this issue, TRex offers an `AbstractVisitor` performing rudimental action when traversing the AST. To perform further actions at the nodes of the AST, it can be extended.

4.3.2 Resolution of Imports

The resolution of imports is done by the `ImportVisitor` class, which extends the TRex `AbstractVisitor` class. On each visit of a node representing an import definition a flag is set. Since the `AbstractVisitor` traverses the AST, the next identifier node to be visited will be the one containing the leaf with the ID of the imported module. For this reason, the visit-methods for the identifier nodes are overwritten to check on each visit of an identifier node if the flag is set. If set, an edge will be added from the module node whose AST is currently traversed to the module node of the graph whose ID equals the detected one. Additionally, the flag has to be removed.

4.3.3 Resolution of References

To resolve references, the `ReferenceVisitor` extends the `AbstractVisitor` by overwriting actions to be performed when visiting AST nodes representing references. In section 3.2.3 the relevant nodes of the AST have already been discussed. On each visit of such a node a function named `foundReference()` is called accepting a reference to the detected AST node as parameter. Within the `foundReference()` method this node will be termed `identNode`.

Then, two further steps have to be performed in order to retrieve a reference to the AST node containing the declaration of the referenced element. As shown in listing 4.1, at first an instance of the TRex `Symbol` class has to be retrieved (line 3 in the listing), which represents the declaration of the referenced element. In order to get it, particular methods from the TRex AST are applied, which are out of scope for this thesis. As soon as this symbol is retrieved, the actual reference to the respective AST node can be be resolved (see line 7 in the listing).

At this point, references to AST nodes representing the referencing element and the declaration of the referenced element have been resolved. Due to nesting, those elements may not equal the TLEs, between which the edges have to be added within the coupling graph. Hence, in a last step the corresponding TLEs have to be resolved. Instead of demanding re-traversals of the AST, TRex provides functionality to return parent nodes with particular properties. In this case, a *scope-level 2* parent representing a declaration can be required, which delivers a reference to the AST node representing the wanted TLE defini-

```
1 // identNode is a reference to the detected reference node
2 // first step of resolution is to retrieve the appropriate symbol
3 Symbol symbol = identNode.getScope().resolve(identNode);
4 // ... exception if symbol is null
5
6 // second step is to retrieve the AST node with the needed declaration
7 LocationAST declarationNode = symbol.getDeclarationNode();
8 // .. exception if declarationNode.getScope().getScopeSymbol() is null
```

Listing 4.1: Resolution of declaration from referenced elements.

tion. Additionally, the parent node which represents the respective module definition can be retrieved analogously.

As soon as the two TLE names as well as the two module IDs are known, the corresponding nodes of the coupling graph can be detected and the references, which represent the edges, can be stored.

4.3.4 Calculation of Metrics

To compute the metrics defined in 2.3.2 the internal graph is utilized. The calculations are performed by methods of the class `MetricsCalculator`. A list of the metrics measuring various properties of modules is given in table 4.1, another one containing metrics for TLEs is given in table 4.2. If necessary, a description is also given in the tables. Most of the metrics are calculated by counting the different kinds of edges of the tool's internal graph, for the others ratios of already obtained metrics are computed. Since the edges are represented by lists of references, in fact such calculations are as simple as getting a list's size.

4.3.5 Output

Three types of output can be performed by the class `Exporter`, i.e. printing the metrics into CSV-files, transferring the internal graph into a DOT-file and printing visibility suggestions into a CSV-file. Since test suites usually consist of several modules, the graph containing all modules is often very large. To improve readability, the `Exporter` also respects user inputs, which can for instance specify prints of sub-graphs. However, this functionality will be presented in the applications chapter. Since visibility rules are a relatively new feature of TTCN-3, the tool makes also use of its Fan In metrics in order to provide visibility suggestions for TLEs. This functionality will also be studied in chapter 5.

Name of Metric	**Description**
Module Fan In	Number of modules by which a module is referenced
Module Fan Out	Number of referenced modules
Number of References	-
Number of Referenced By Relations	-
Number of External References	Number of references to other modules
Number of Internal References	Number of references within a module
Number of External Referenced By Relations	-
Number of Imports	Number of imported modules
Number of Useless Imports	Number of Imports minus Fan Out
Number of References per Needed Import	Ratio of Number of References and Fan Out
Number of Useless Imported By Dependencies	-
Maximum Import Weight	Highest amount of references between two modules
Minimum Import Weight	Lowest amount of references between two modules
Maximum Imported By Weight	-
Minimum Imported By Weight	-
Average Import Weight	Ratio of Number of External References and Fan Out
Average Imported By Weight	-

Table 4.1: Overview of the tool's metrics for TTCN-3 modules.

Name of Metric	**Description**
Parent Module Fan In	Fan In of module which contains the TLE
Parent Module Fan Out	Fan Out of module which contains the TLE
Number of Internal References	Number of references within TLE's parent module
Maximal Number of References to an External Module	Maximum amount of references to one module
TLE Fan Out	Number of references to other TLEs located in other modules
TLE Fan In	Number of references to the TLE which belong to TLEs from other modules
External Usage Indicator	Difference of Maximal Number of References to an External Module and Number of Internal References
Module with Maximal Dependency	*Module to which an element has the most references* (actually not a metric, but treated so)

Table 4.2: Overview of the tool's metrics for TTCN-3 Top Level Elements.

5 Applications

In this chapter it is studied how to apply *TTPendency*. At first, its parameters are described, afterwards the output options are presented and discussed.

5.1 Parameters of the Tool

As it is convention of a command line tool, *TTPendency* accepts several parameters in a short form `-v` and a long form `--verbose`. A list of the tool's parameters is given in table 5.1. Listing 5.1 shows how to call *TTPendency*, if the folder `~/ttcnProject` has to be analysed and the file extension is `ttcn`.

```
1 java -jar TTpendency.jar -d "~/ttcnProject" -t "ttcn"
```

Listing 5.1: Start of the command line tool TTPendency.

5.2 Options for the Graph Output

Per default, *TTpendency* creates a graph representing the structure of the whole test suite. Arrows represent dependencies caused by import definitions and distances between the elements reflect to which extent modules depend on each other, which is measured by the Number of References between the modules. An example for such a graph is given in figure 5.1, where module `foo` is imported by `bar`.

The file name of the graph can be specified by the parameter `--dotFile`. Additionally, there are four different edge types available, which can be set via the parameter `--graph` (see listing 5.2).

```
1 java -jar TTpendency.jar -d "~/ourProject" --dotFile "myGraph.dot" --graph 0
```

Listing 5.2: Specification of the file name and edge type.

Short Form	Long Form	Description
-t	--ttcnType	Specification of file extension *Default:* `ttcn3`
-d	--ttcnDirPath	path to directory for analysis *Default:* ~
	--dotFile	Name of dot file for the graph on module level *Default:* `moduleGraph.dot`
	--refOfFile	Name of dot file related to the one node option *Default:* `referencesOfOneNode.dot`
	--refedByFile	Like `refOfFile`, but edges result from referenced-by relations. *Default:* `referencedByOneNode.dot`
	--visibFile	Name of csv file containing visibility suggestions *Default:* `visibilityFile.csv`
	--metricsFile	name of csv file with metrics *Default:* metricsFile.csv
	--precDot	Name of dot file for graph extended by top level elements
	--TLEmetrics	Graph for Top Level Elements *Default:* TLEMetrics.csv *Default:* `TLelementsGraph.dot`
	--oneNode	Enables one node operations for the specified module *Default:* unset
	--graph	specifies graph type, options are. 0, 1, 2, 3 *Default:* 0
	--help	prints the embedded help

Table 5.1: TTPendency's parameter list.

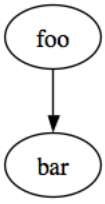

Figure 5.1: Graph of a test suite consisting of two modules, i.e. foo and bar. The arrow represents that bar is imported by foo.

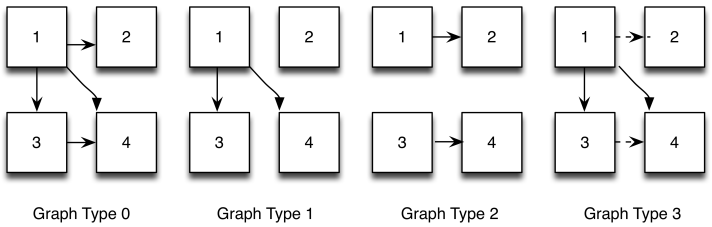

Figure 5.2: Illustration of the four graph types TTPendency provides.

An illustration of the different graph types is given in figure 5.2, and their meaning is as follows:

- Graph type 0: Edges represent imports
- Graph type 1: Edges represent needed imports
- Graph type 2: Edges represent useless imports
- Graph type 3: Edges represent needed imports and dashed represent edges useless imports

Sometimes it may be helpful to study only one module's dependencies. In this case, sub-graphs can be created by specifying the `--oneNode` parameter. If set, two graphs on module level will be created: The edges of the first one represent the imports of the analysed module. Its file name can be specified by the parameter `--refOfFile`. The edges of the second one represent imported by relations, and for this graph type the file name can be specified by the parameter `--refedByFile`. Additionally, the four graph types specifiable with the `--graph` parameter are also available for sub-graphs. In listing 5.3 the call parameters for creation of a sub-graph of module `commonDefs` are given.

```
1 java -jar TTpendency.jar -d "/ourProject" --oneNode typeDefs --graph 0
```

Listing 5.3: Utilization of the oneNode option to create a sub-graph which only contains modules imported by the module typeDefs.

Besides creation of graphs only containing modules, the `--oneNode` parameter also enforces creation of two graphs, which also contain the TLEs of the visualized modules. This option can be helpful for a more detailed analysis of the origins of dependencies. In the

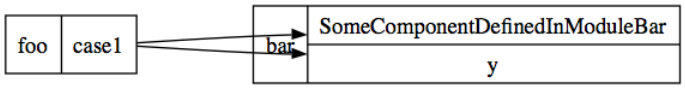

Figure 5.3: Visualization of references between top level elements of module foo and module bar.

example given in figure 5.3 it can be observed that the TLE case1 references both TLEs of module bar.

The parameters to specify the corresponding file names are `--precDot` and `--precDotRev`. As in the previous discussed graph type, the edges of the first graph with TLEs represent references, while the references of the second graph with TLEs represent referenced-by relations. In listing 5.4, an example for the specification of the call parameters is given.

```
1 java -jar TTpendency.jar -d ~/ourProject --oneNode foo --precDot modulescopegraph.dot
```

Listing 5.4: Specification of file name for graph whose node's depend on module foo.

5.3 Output of Metrics

The tool calculates metrics based on its internal graph. As already discussed in chapter 4.3.4, there are metrics available describing either attributes of modules or attributes of TLEs. The file name, in which the metrics for the modules will be stored, can be specified by the parameter `--metricsFile`, and the parameter to specify the metrics file for the TLE is `--TLEmetrics`. In the case study (see chapter 6) it will be discussed how to analyse and interpret those metrics.

5.4 Suggestions for Visibility Rules

Visibility rules have been added to the TTCN-3 core notation in 2009. Accordingly, they are quite new and most test suites do not use them. In order to declare elements as private, it must be ensured, that they are not referenced by TLEs from other modules. Hence, TLEs whose Fan In equals zero can be suggested to be private. Analogously, TLEs with a positive Fan In can be suggested to be public. Such visibility suggestions are provided by *TTPendency* and the name of the CSV-file containing the visibility suggestions can be specified by the parameter `--visibFile`.

6 Case Study

In order to validate the methodology, the tool was applied to version *IWD_09wk04* of the 3GPP test suite development project, which consists of 53 modules and 20101 Lines of Code. In this chapter the results are presented and discussed.

At first, an overview of the analyzed project is given, which summarizes the metrics of the test suite's modules. Afterwards, it is demonstrated how *TTPendency* can be applied to such a real test suite by interpreting the meaning of module and TLE metrics and visualizations.

6.1 Project Overview

A summary of the information gained from the module metrics of the analyzed project is given in table 6.1.

Metric	Value
Number of Imports	411
Number of References	3361
Number of External References	2457
Number of Internal References	904

Table 6.1: General information about the 3GPP test suite development project.

It can be observed that a lot of interdependence exists in the analyzed project: As the Number of External References are more than twice as high as the Number of Internal References, there must be a lot of coupling. This can also be obtained in figure 6.4, which commemorates the phrase *not to see the wood for the trees* due to numerous edges. Hence, the graph and the metrics of the overview underline the demand of a methodology for dependency detection and analysis in TTCN-3 test suites.

6.2 Analysis of Module Metrics

In this section metrics of modules will be discussed. For this purpose, a selection of modules and metrics is given in table 6.2, which will be used as example.

Module ID	FO	FI	NOI	NFI	R	ER.	NIB	FNIB	MIW
NasEmu	9	1	10	1	69	56	1	0	13
EPS_NAS_TypeDefs	2	9	2	0	425	332	14	12	285
EUTRA_CommonSteps	20	14	21	1	236	210	15	0	29
EUTRA_Security_Templ.	0	1	4	4	0	0	2	1	0
RLC_testcases	13	1	16	3	339	285	1	0	112
RLC_Templates	1	1	11	10	3	3	3	2	3

Table 6.2: A selection of module metrics. Abbreviations: FO=Fan Out, FI =Fan In, NOI=Number of Imports, NFI= Number of False Imports, R= Number of References, ER=Number of External References, NIB=Number of imported-by Dependencies, FNIB= False NIB, MIW=Maximum Import Weight

6.2.1 Detection of Useless Dependencies

In table 6.2 it can be observed that in five of the six given examples the Fan Out does not equal the Number of Imports and thus, useless imports must exist.

Not presented in the table is the Number of Useless Imports detected within the whole test suite, which is 124. Since the whole test suite covers 411 imports, more than thirty percent of the detected import definitions are useless.

In order to identify the useless imports, the sub-graph for the appropriate modules can be utilized. If the graph type 3 is selected, edges representing useless imports will be dashed. For the module NasEmu such a sub-graph is given in figure 6.1. By observing this graph it can be obtained that a useless import of module CommonDefs exists.

The detection of useless import can also be based on metrics dealing with imported-by and referenced-by relations. Accordingly, the metric Number of imported-by Dependencies and the metric Number of False imported-by Dependencies have to be analysed in combination with the sub-graphs whose edges denote imported- or referenced-by relations. This approach will be helpful, if modules are almost private. In such a case the graphs will help to analyze the dependencies to the few modules, which import the examined module.

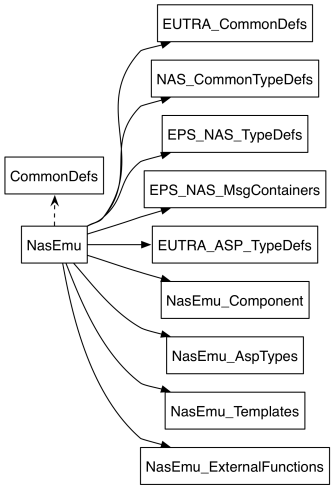

Figure 6.1: Sub-Graph of Module NasEmu (with graph type 3).

6.2.2 Analysis of Fan Out Metrics

Since the Fan Out metric is an indicator for external usage, it can be utilized to identify modules having high external dependency. If the metric Maximum Import Weight also suggests that the module with the high Fan Out is an outlier, this module may contain elements with a lot of dependency towards another module. Hence, the location of some of its code elements may be disadvantageous.

The graph of figure 6.2 displays the metrics Fan Out, Number of External References as well as their ratio for each module of the test suite development project, which is analyzed within this chapter. It can be observed that few outliers exist having either several references to other modules or a high Fan Out.

One of the outliers is module `PS_Nas_TypeDefs` having 332 references to other modules and a Maximum Import Weight of 285. Since it has 425 references, more than half of its references point to another module. In figure 6.3 its sub-graph is illustrated (graph type 0). Since module `Nas_CommonTypeDefs` is closer than `CommonDefs`, this module is the one referenced 285 times. Accordingly, there may be elements located in `EPS_Nas_TypeDefs`, which should be located in `Nas_CommonTypeDefs`. Since the goal of this thesis is development of a methodology to detect coupling, further analysis of the detected module's source code is not a concern.

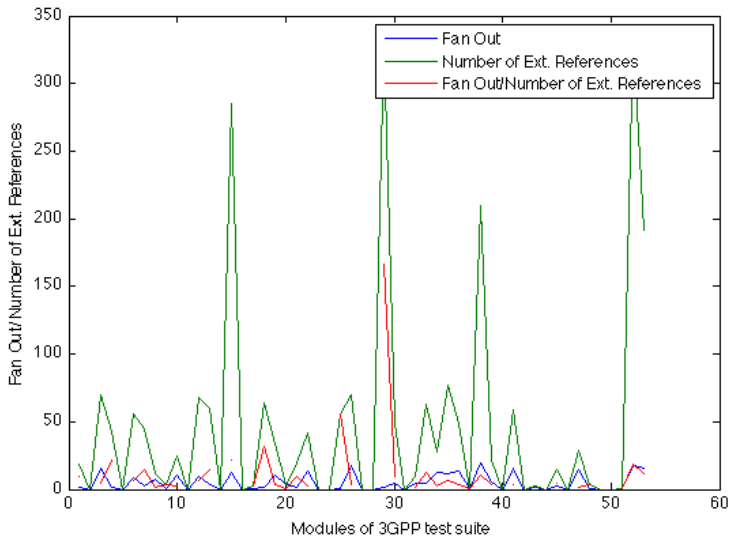

Figure 6.2: Graph displaying the metrics Fan Out, Number of External References and their ratio for each module of the analyzed 3GPP test suite development project.

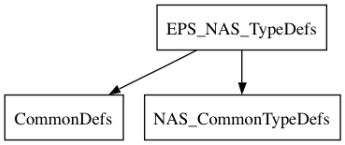

Figure 6.3: Sub-Graph of Module EPS_Nas_TypeDefs (with graph type 3).

6.3 Analysis of TLE Metrics

Instead of analyzing module metrics to detect outliers, TLE metrics can also be used to detect elements having a high amount of references to another module. In table 6.3 a selection of the TLE metrics is given.

It can be observed that both modules have an External Usage Indicator greater than zero thus having more references to an external module more than within their own module. However, for both elements the actual source code has to be examined in order to find out if the location of the elements is semantically acceptable. The analysis with *TTPendency* shows that moving the detected elements would decrease coupling.

TLE ID	PM FI	PM FO	IR	MREM	FO	EUI	MMD
LB_SetupDRB_IE_Type	2	2	0	3	3	3	CommonDefs
f_TC_8_5_4_1_EUTRA	1	16	5	10	46	41	EUTRA_ CommonSteps

Table 6.3: A selection of TLE metrics. LB_SetupDRB_IE_Type belongs to module EUTRA_LoopBack_TypeDefs and f_TC_8_5_4_1_EUTRA to module RRC_Others. Abbreviations: PM=Parent Module, FI = Fan In, FO= Fan Out, IR= Internal References, MREM= Maximal Number of References to External Module, EUI= External Usage Indicator, MMD= Module with Maximal Dependency.

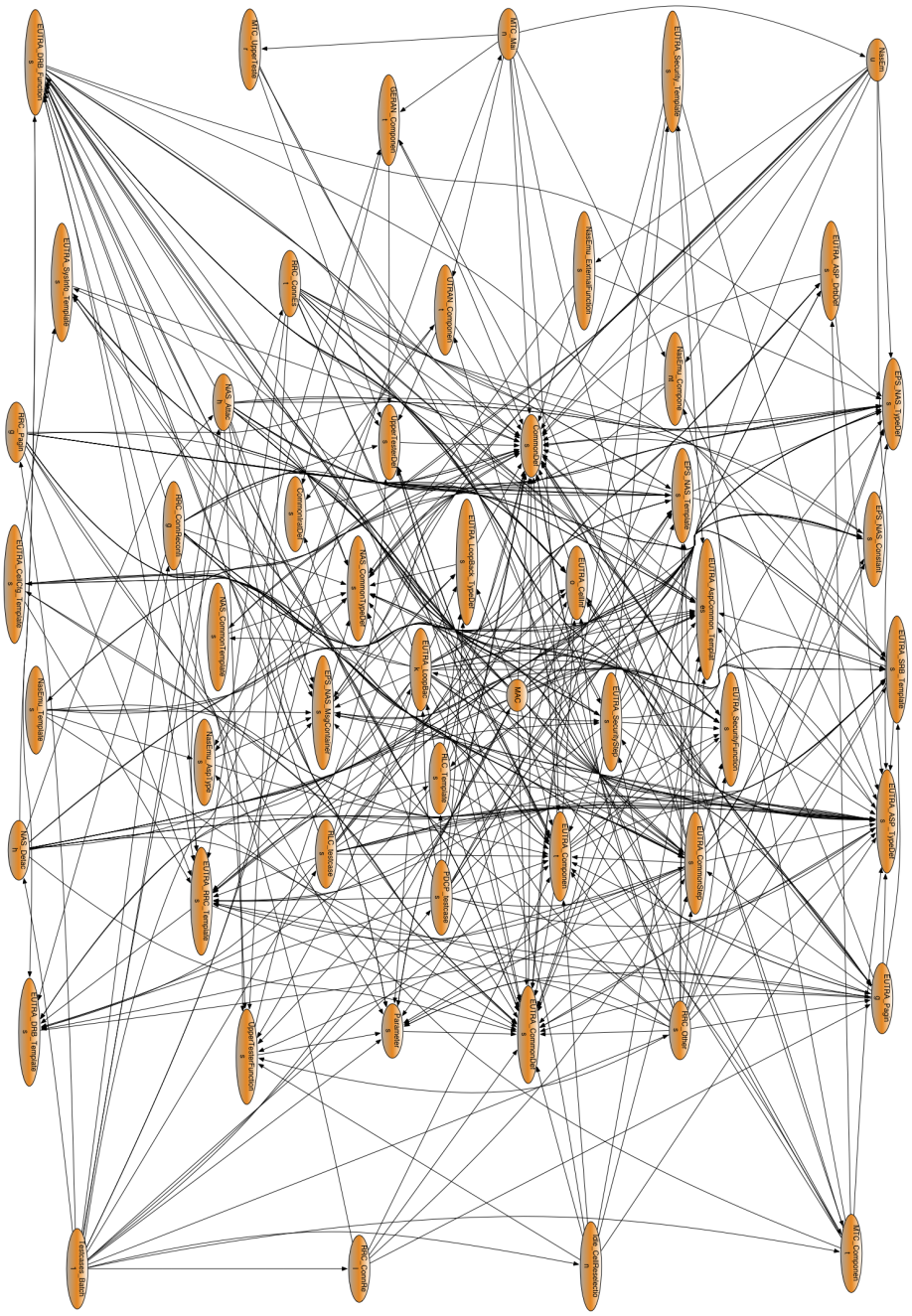

Figure 6.4: Graph visualizing imports in 3GPP project which are caused by import definitions.

7 Conclusion

This thesis presented a methodology for the detection and analysis of dependencies between TTCN-3 modules. In addition, a Java tool was developed as a proof-of-concept which provides calculation of coupling metrics and graph-visualizations of a test suite's internal structure. Coupling metrics and analysis were required due to increasing sizes and complexities of TTCN-3 test suites and thus due to coupling, which makes maintenance of test suites hard to ensure.

In order to validate the developed methodology, the tool was applied to a real test suite. The observed metrics and graph-visualizations underlined the existence of several dependencies and thus they highlighted the requirement of coupling detection in TTCN-3 test suites.

The application of the tool to the test suite also facilitated identification of modules that are highly affected by coupling. Analysis of the sub-graphs providing the visualization of dependencies of a particular module then pointed at the origins of the detected coupling. In addition, the detection of useless dependencies also identified easy targets to decrease coupling metrics, and analysis of metrics defined for elements of a module's definitions part, which are denoted by the term Top Level Elements (TLEs) within this thesis, provided identification of several elements supposedly located in a disadvantageous module. If such TLEs were located in the proper module, this would allow for a further decrease of coupling.

Hence, the results obtained by the case study provided an extensive analysis of the dependencies and allowed detailed insight into the test suite's structure. Therefore, it can be concluded that the developed methodology is effective in detecting and analysing dependencies between modules of the TTCN-3. Furthermore, it is even applicable for the detection of useless dependencies and can be utilized to make propositions for refactorings.

Future work can center on extending visualization options for the graph representing the whole test suite, which is currently hard to observe. Another topic of interest would be the development of strategies to provide suggestions for friend-declarations, which could be based on metrics describing the extent of dependency between modules. Furthermore, even methodologies for placement suggestions based on the tool's metrics could be developed. Integration in TRex could then facilitate coupling-based refactorings.

Bibliography

[1] ETSI. Standard ES 201 873-1 V4.1.1 (2009-03), Part 1: TTCN-3 Core Language. European Telecommunications Standards Institute (ETSI), Sophia-Antipolis, France, March 2009.

[2] Norman E. Fenton and Shari Lawrence Pfleeger. *Software metrics*. PWS Publishing Company, 20 Park Plaza, Boston, MA 02116-4324.

[3] Nicola Howarth. Abstract syntax tree design. Technical report, ANSA, 1995.

[4] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice*. 2006.

[5] Helmut Neukirchen, Benjamin Zeiß, Jens Grabowski, Paul Baker, and Dominic Evans. Quality assurance for TTCN-3 test specifications. *Software Testing, Verification and Reliability (STVR), Volume 18, Issue 2. (ISSN 0960-0833) DOI: http://dx.doi.org/10.1002/stvr.379*, pages 71–97, June 2008.

[6] Stefano Crespi Reghizzi. *Formal Languages and Compilation*. Springer Publishing Company, Incorporated, 2009.

[7] Colin Willock, Thomas Deiß, Stephan Tobies, Stefan Keil, Federico Engler, and Stephan Schulz. *An Introduction to TTCN-3*. John Wiey & Sons, Ltd.

[8] E. Yourdon and L.L. Constantine. *Structured design: fundamentals of a discipline of computer program and systems design*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1979.

[9] Wei Zhao. Entwicklung eines Parsers für TTCN-3 Version 3 unter Verwendung des Parsergenerators ANTLR. Bachelor's thesis, Bachelorarbeit im Studiengang Angewandte Informatik am Institut für Informatik, ZFI-BM-2005-15, ISSN 1612-6793, Zentrum für Informatik, Georg-August-Universität Göttingen, August 2005.